1/24 Lecture: Security basics: goals, threat models

Beili Wang (with minor edits by Rob Johnson)

*** (Black color is what the professor wrote on the board; Blue color is the notes from the class discussion.)**

General Information about the class:
The course focuses on: single system that multiple users can use but not interfere with each other.

Things on Paper Review:
Is it going to work?
What will be the cost?
What is the result?
How can I get access the system? Break the system?

Objectives in Security:

1. Protecting a resource
2. Private: files, folders, bank accounts, etc.
    a. From whom?
    b. From what done to them?
2. From copying (reading), deleting, corrupting.

General Goals:

1. Confidentiality (copying, reading)
2. Integrity (corrupting)
3. Availability (deleting) e.g. virus

Example: Buffer overflow Attack
Integrity – memory corrupting
Availability – gain total control
Confidentiality – can do anything

1. Confidentiality: (protecting the resource)

- Eavesdropper can’t read message in transit.
- Thief can’t read documents on stolen laptop.
- Other users can’t tell that I’m running a large computation.

  (ex: mostly in military setting, very expensive to achieve this goal.)
- Observer can’t tell that I’m talking to Bob.

  (ex: voting machine)

2. Integrity: (Does the system achieve what it should achieve?)

- Only authorized users can modify database.

- No one can modify a program's memory.
- Only authorized users can run programs.
- I only want to accept messages that come from Bob.

4. Availability:

- Attacker can’t prevent me from shopping on Amazon.
- Attacker can’t consume all of disk.
- Use can’t degrade performance to unusable levels.

Q: Why separate address space?
A: More reliable. (Mistakes happen that do not need to shut down the whole system.) Also, can swap to disk.)

Q: Difference between securities vs. correctness?

1. System may not be fully correct.
2. Correctness:
   Mistakes, accident do happen
   Ex: Raid 5

Main difference between accidents and security?
Adversary (who has a hammer?)
“Something goes wrong in worst possible way.” (security)
“Adversary has limited power.”

Two Questions:
What do we want to achieve?
Who do we want to achieve it against? (attacker).

Threat models:
Limitation of Attacker:

1. computational power
2. bandwidth (not a problem)
3. storage (not a problem)
4. time
5. money

1. Computational Power
   1 Pentium ~ 4GHz = 2 ^32 instructions/second
   1 Blue Gene ~ 2 ^16 CPUs
   So total 2^48 instructions/second
   1 year about 2^25 seconds/year
   So about 2^73 instructions/year
   100 years about 2^7 years

So total is about 2 ^80 instructions

It is a lot but not infinite amount. The message is time sensitive which limits attacker.
(not important after 100 years)
However: ex: Who kill JFK?
2^128 keys try 2^80 instructions, we left 2^-48 probabilities of recovery key.
Moore's law: computer power doubles, get faster exponentially.

2. Bandwidth:
   network flood with packets:
   1 attacker, dial up 56kb/second
   100,000, dial up 5.6Gb/second
   On today's Internet, bandwidth usually not a problem.
   An attack that "propagates from one machine to another" is called a worm.

3. Storage is not a problem today.

4. Time:
   "Attack at Dawn."
   a. Time sensitive: limits attacker.
   b. Rekeying, e.g. every 24 hours, limits attacker.

2. Money:
   Attacker limited in money.
- Poor attackers: script kiddies
- Resource isn't valuable. (example: Trade secret: $1 million)
- This also limits defender.
  (example: Biometric: fingerprint, Japanese researcher uses gum to break it. (even cheaper)).

---

1. Access (gain access)
   Outsider or Insider
   Local vs. Remote attackers
   a. Remote attackers
      - attackers can communicate with web server, ftp server.
      - Cannot log in
b. Local attackers
      - assume attacker can log in

2. Related to Access: knowledge
- knows OS version, Applications version
- configuration information
- source code

- vulnerabilities in OS/Apps

3. Doesn’t know:

- passwords
- private keys
- random numbers

  (example: ssh demon)

Summary:
Defender: Security Goals (What you try to do?)

Attacker: Threat Models

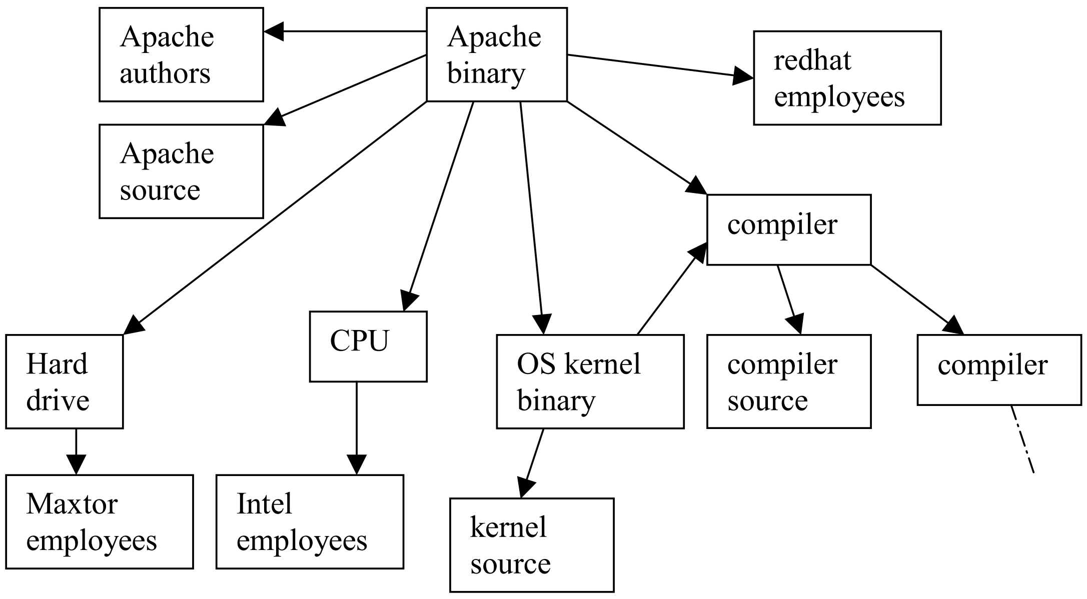
- Computation power
- Time
- Bandwidth
- Storage
- Money
- (What you try to against?)

1/26 Lecture: Trust, open design, principles of secure system design

*** (Black color is what the professor wrote on the board; Blue color is the notes from the class discussion. – Beili Wang)**

Trust
Trust == Dependence, BAD
Trust is transitive.
Trust in system security means dependence and dependence is bad.
Trust has transitive property. If A trusts B, B trusts C, it means A trusts C.

For Example: Apache enforce .htaccess. Apache enforce only .htaccess.



How to limit trust?

- verify everything
- multiple independent verifications (different people look at the stuffs)
- online testing (real time testing)
- build everything from scratch

Base case:
We lose if one system is compromised.

Independent checking:
Attacker must break all redundant checks.

For example: three organizations checking voting result or three compilers compiling the code.

Trusted Computing Base:

A resource is in the TCB if its correct operation is necessary for security.
You don't have to trust the code, if you don't install the code on the computer.

Rule: Keep TCB small.

Keep it small. Keep it simple.

Rule: Economy of mechanism.

- less code
- simple code

Clarity and conceptual simple

For example: Economy of mechanism

Exercise: passwords
Username unique
Hash password

- flat file – depends on OS
- encrypt
- table in database – depends on DB, OS

flat file vs. table in database
Storing in flat file is better but slow.

Rule: Failsafe Defaults.

- trust no one
- read only
- password protected
- default: DENY vs. ALLOW

if wrong:

break in (quietly)	ALLOW – easy (ex: Microsoft)
complaints (noisily)	DENY – forces administer to set policy (ex: DOS)

default is ALLOW: if wrong, administer knows when break in. The system failed quietly.
default is DENY: if wrong, administer know when complaints. The system failed noisily.

Rule: When fail, fail loudly possible.

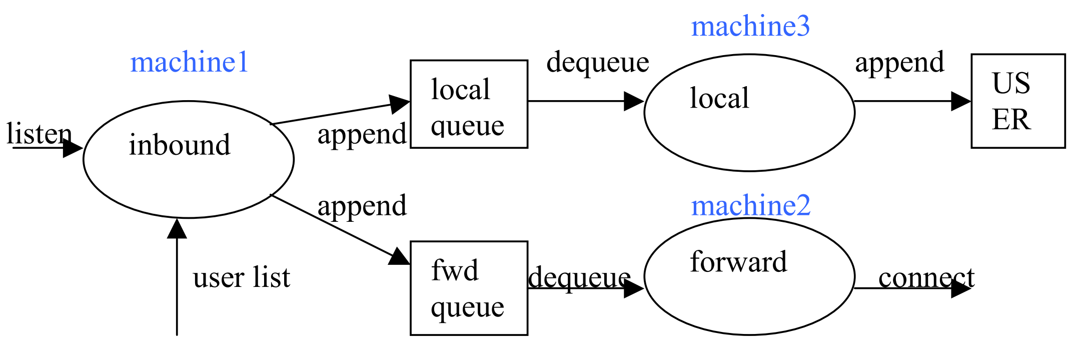
Rule: Least Privilege.

- Give each user/program least amount of power necessary to do its job.

For example: You are root or not.
Root: can do anything.

Not Root: only do the things can.
Unix does not give good support of least privilege.
Exercise: Mail delivery system.

- listen to network
- look up users
- write to user mail
- queuing
- connect



does not have privilege to connect other machine
it is hard for machine1 connect to machine 2

Rule: Separation of Privilege

- Harder to subvert multiple subsystems.

For example: Force attacker to do two things instead one thing to win.

Rule: Least Shared Mechanism (Least Shared State)

e.g. Slashdot user account is different from slashdot's server user account.
- separate database
- log into the account does not have privilege to access the server.

Rule: Complete Mediation

- open
- creat

protect every way can get it.
For example: have security check on open, must have security check on create.

- open( ) vs. read( )

For example: Unix enforce complete mediation on file system.

Process A	Process B
Open(f)	
	Chmod a –rf
Read(...)	

Yes and No complete mediation

Unix design consider efficiency. It caching open and use for read. However, the caching result maybe stale. Unix consider this as ok.

Rule: Open Design

"Full Disclosure"

1. Application released
2. Bug found by user
3. user tells world (explore the bug)   or   3'. user tells author in private

3.5. Hack, Hack, Hack

4. author fix bug
5. Users upgrade

For example: Paper Fuzz.

"Full Disclosure" is better. For example: Microsoft bugs will be fixed in average 50 days in full disclosure vs. 100 days without full disclosure.

CSE 509: Lecture Notes for 1/31/2006

Human Factors in Security:

Unusable security is unused security.

Sometimes security gets in the way, so users will circumvent it.
e.g. Configuring NFS is such a hassle, that its hard to get it right in the first few attempts.

With Cryptography, there have been sucesses and failures.
Sucesses: SSH, SSL
Failures: Email (Mozilla plugin for crypto. - Enigmail), WEP (Implementation issues. Crypto is not secure.)

The general principle is that the cost to secure should be less than the cost of what you are securing. This has been in favour of Crypto in SSH (securing the password) and SSL (securing money - online transactions) and against Email. So, most email is not secret.

e.g. Corporate N/W
It has a firewall and a policy that it doesn't allow any outside connection to a machine in its network and it doesn't allow access to websites on internet apart from CNN. To get around this security policy, a user could install a modem on a PC that directly connects to the internet. The security is thus compromised.

```
    ---------------------------------------------|
    |            ______                          |
 -------        |      |                         |
 -Modem---------| PC   |____                     |
 -------        |______|    |    ___________     |
                            |   |           |
                ______      |___| Firewall  |____  INTERNET
                |    |      |   |___________|
                | PC |______|
                |____|
```

Trend in Computer Security : Greater Sharing

```
Separate   -->  Multiprogramming Env  -->  RPC  -->  Mobile  -->      ?
Computers           like Unix              IPC        Code        Who knows what
          share                       share
          CPU                          data
                                  ---------------------
                                   We are here today
```

Here is an example of Good Security, but No Sharing

```
                    ____________________________________________________
                   |                                                    |
    Customers          Operator    Computer   Printer   Generator       |
    with disk      |                                                    |
    (& donuts!)    |____________________________________________________|
                                 Secure Room
```

Running System:
1. Users give disk to operators.
2. Operator inserts disk and turns on computer.
3. Computer runs for one hour.
4. Operator gives printout to the user.
5. Operator turns off the computer. (Complete Reset)

Goals of the System:
1. Confidentiality
2. Integrity
3. Availability

Ways in which security could be breached:
1. User gives malicious code on disk.
2. Operator can be bribed. (No trust).
3. Operator could (by mistake) give wrong / different output.
(Trusting the Operator to be honest, diligent and a robot for integrity and availability.)

4. Reading previous user's data from the memory / disk. (Lets assume / or the design is such that Poweroff is a total reset, and nothing belonging to the previous user is retained.) So confidentiality.
5. Electromagnetic attack. (Need the room to be sealed against this)
6. Stolen disk. (Need some kind of user authentication for this)

The major disadvantage of this system is its Inefficiency.

It seems so hard to get this simple a system work right and remain secure. Think about a bigger, more complex system.

Virtual Machines :

```
                              ________________________________
   ____________________      |TLB                             |
  | CPU        _______ |     |   virtual | physical | R | W   |   ____     __________
  |           | Regs  ||---- |    ....      ......    ..  ..  |---| C |---|          |
  |           |_______||     |________________________________|   | A |   |          |
  |  __________        |                   addr                   | C |   |  Memory  |
  | | Interrupt |      |                                          | H |   |          |
  | | Handler   |      |                   data                   | E |---|__________|
  | |___________|      |------------------------------------------|___|   |          |
  |          ________  |                                        _______   |  Disk    |
  |         | Priv   | |                                       |Disk |----|__________|
  |  ______ |________| |                                       |_____|    |          |
  | |Clock|            |                                   __________     | Printer  |
  | |_____|            |                                  | Printer |-----|__________|
  |____________________|                                  |_________|     |          |
                                                          __________      |   N/W    |
                                                         | Network |-----|__________|
                                                         |_________|
```

Introduce a supervisor that limits programs

A program on this system can write anywhere in the memory. So, on this system memory protection => i/o protection.

Instructions:
- Load / Store among Reg / Memory
- Reg / Reg computation
- Flush the Cache
- Load / Store the IHA

To include a supervisor, lets add a instruction to
- Load / Store to the TLB

But, a program can write out the TLB and change it to allow access to write to some memory.
Hence, we need a privilege register.

Privilege Register :

Load TLB	Yes
Read TLB	No for Security, Yes for Virtual Machines
Load IHA	Yes
Read IHA	No for Security, Yes for Virtual Machines

We need a page-fault handler mechanism. Lets utilize Interrput Handling to handle page faults.
Incase, we have a page-fault, the interrput handler jumps into the interrupt handler code which is supervisor code, the privilege bit is on. Now, there is again an issue with translation from virtual to physical which could cause another page fault. One option is to not use TLB when in privilege mode. The other option is that the TLB keeps a fixed entry for Interrput handler code.

Job of the OS:
Loads page of Interrupt Handler (IH) into TLB (read only)
Loads addr of IH into IHA.
Loads process TLB entries.
Turns off the privilege bit.
Jumps to the program.

Now, we have memory protection !!

If the kernel area (privilege area) is readable by any process, it is obviously a concern.
e.g. File-systems could use memory to cache data / structure.
So, keep nothing that when read is useful or don't allow reading.
OS should reset the registers, clear memory pages of the process.

However, with this design, we don't have availability, since nothing stops a process from running forever.
So add a clock, schedular.

Lecture Notes for 02/02/06:
Access Control

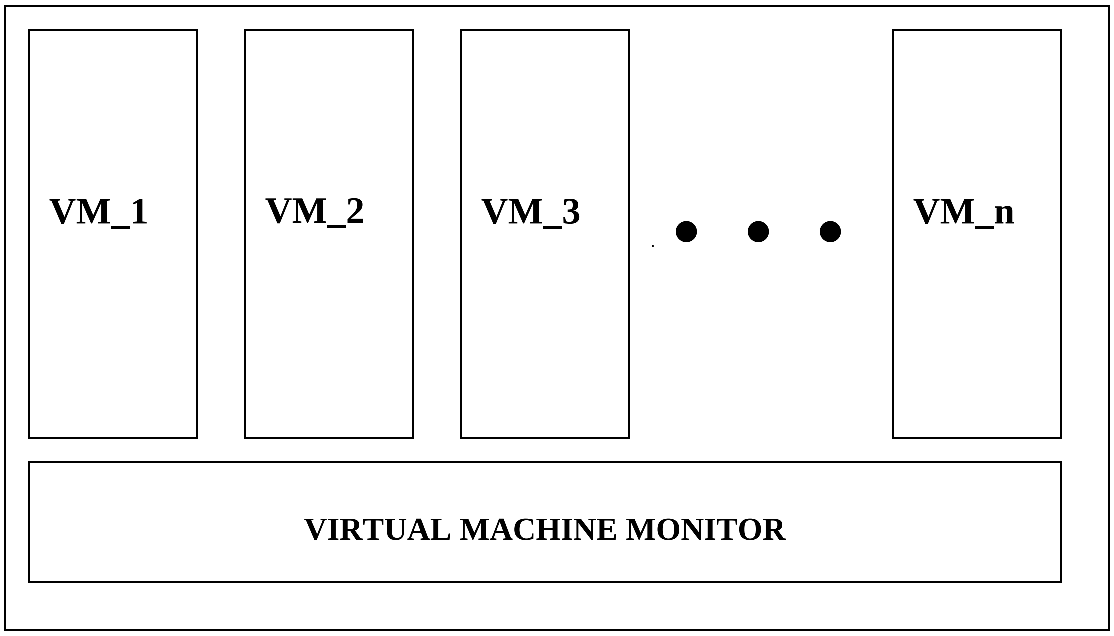
(Blue part is taken from: http://en.wikipedia.org/wiki/Virtual_Machine)

Hardware:

In last class we talked about a computer system that supports a supervisor and that supervisor can control the following:

- Memory accessible to a non-supervisor program. (This is controlled with the help of TLBs, IHA, and supervisor bit.)
- I/O performed by non-supervisors. (We get this one for free because of memory access control.)
- Time on CPU for non-supervisors. (This is controlled because we can have interrupts every 100 or 1000 seconds which would give control back to the supervisor.)

Question) Now, what high level architecture can we build for this computer system?
Answer) We can have the following:



(VM stands for Virtual Machine.The original meaning of **virtual machine** is the creation of a number of different identical execution environments on a single computer, each of which exactly emulates the host computer. This provides each user with the illusion of having an entire computer, but one that is their "private" machine, isolated from other users, all on a single physical machine. The host software which provides this capability is often referred to as a **virtual machine monitor**.)

General information about this model:

- Virtual Memory Monitor (VMM) gives each virtual machine the illusion of a complete computer to itself.
- Each Virtual Machine has its own memory space.
- There is no overlap amongst memory.
- CPU runs each VM for 100th of a second and then switches to the next VM for execution.
- For jobs that require printing from a printer, the CPU has to run each job until it is finished before switching to the next job that is available for printing.
- There exists one big disk, but the disk space is partitioned amongst the VM's.

Sharing In This System:

What can we say about sharing in this system?

- Code: There is NO sharing of code amongst VMs.
- Data: There is NO sharing of data amongst VMs.
- Hardware: All VM's are on a single physical machine.

So, in terms of sharing this is still not so great.

Security in the System:

Which of the security goals are accomplished?

Confidentiality:	This is maintained.
Integrity:	This is also maintained.
Availability:	This is maintained except for when long jobs are using the printer.

So, in terms of security the system is good.

However, there is one security concern. CPU's tend to get hot during long computations and it is possible to measure the heat. By measuring the heat, one virtual machine, say VM_1, might be able to tell that another virtual machine say, VM_2, is doing a lot of computations. Moreover, by using several VM's, the user can get an idea of the scheduling policy, and can figure out which VM has a high scheduling priority.

Trusted Computing Base (TCB) In The System:

What resources do we trust in this system?

- VMM
- Hardware.

Good Design Principles in the System:

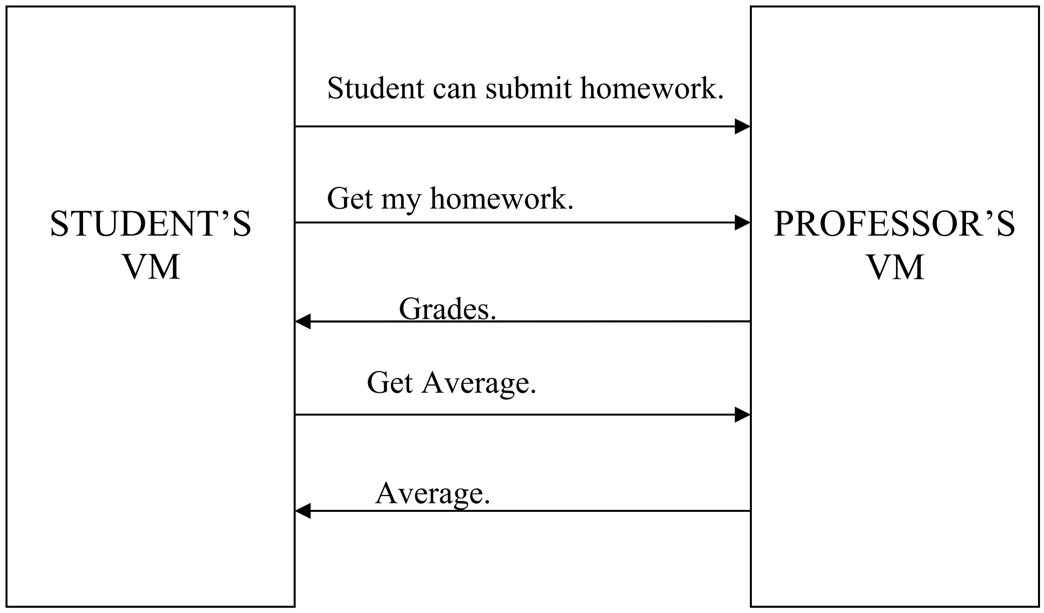
Does the system have

▪ Small TCB:	Yes, the VMM.
▪ Economy of Mechanism:	Yes, The code for this system is simple and easy to understand.
▪ Complete Mediation:	Yes, VMM needs to protect itself and all other Virtual Machines. System depends upon complete mediation.

Changing System for More Sharing:

So far we had VMs that were not cooperating with each other. Let us add a property such that the VMs have more sharing amongst themselves.

Let us take as an example the case in which students would have communication with their professors with regards to their grades:



In this system you can now send messages from one VM to another.
This system has the following aspects;

- Built on Virtual Network.
- VMM ensures the authenticity of the sender ID in each message.

- Client/Server model.

<u>What amount of trust does the two VM have?</u>

- Professor is not confused and is honest.
- Student does not send messages in a huge, irritating amount.
- Trust is just application level.

<u>Remote Procedure Calls:</u>

- Caller and callee can be mutually distrusting. So, we need to build something such that both caller and callee could get what they want even if they don't trust each other.
  - To do this each VM must validate incoming messages.

- This system works:
  - There can be a shared database.
  - There can be shared code for computing the average, etc.
  - There is privacy.
  - There is Confidentiality.
  - There is Integrity because students can't modify their own or other's grades.
  - Availability is also fine.

This system does come at a cost: Each VM has to protect itself from sending or receiving malformed messages.

<u>Who is allowed to do what in this system?</u>

- Professor's VM is enforcing some access control policy.
  - Maybe Ad Hoc.

- Let us formalize access control.
  - We have Objects that professors or students want to access.
  - Subjects $\leftrightarrow$ Domains.
  - We can represent policy as a matrix such that:
    A[d, O] = {r | d can perform r on O.}, where "r" is an access right, "d" is a "domain" and "O" is an object.

One example of an Access Control Matrix would be:

	Professor	**Student1**	**Student2**	**Student3**	**Hw Queue**	**Student1 Grades**	**Student2 grades**	**avg**
Professor	Grant	Throttle	Throttle		Dequeue.	Read. Write.	Read. Write.	Grant. Revoke.
Student1*		grant			Enqueue.	Read.	–	Read
Student2*			grant		Enqueue.	–	Read.	Read
Student3*				grant				Read

* Student1 and Student2 are enrolled but Student3 is not enrolled.

<u>Primitive Operations on Access Control Matrix:</u>

1) Adding a right: A[d,O] = A[d, O] U {r}
2) Removing a right: A[d,O] = A[d, O] \ {r}
3) Add_Object(d,O) : { A[d,O] = {grant, revoke} }
4) Add_Domain(d1, d2)
5) Delete_Object(d,O)
6) Delete_Domain (d1, d2)

It also seems sensible to have conditional operations such as:

grant_right (d1, d2, O, r), which means that "d1 give right 'r' to object 'O' of 'd2', if d1 has the 'grant' access for object 'O'. "

Or symbolically we can say:

$$\text{If} \quad grant \in A[d1, O]$$
$$\rightarrow A[d2, O] = A[d2, O] \cup \{r\}.$$

Similarly we can also have a <u>conditional revoke right</u>:

revoke_right (d1, d2, O, r), which means that:

$$\text{If} \quad revoke \in A[d1, O]$$
$$\rightarrow A[d2, O] = A[d2, O] \setminus \{r\}.$$

<u>Extending Access Control Matrices:</u>

We can have:
A[d,O] = {(r,p) | d has the a right r to O if p}

As an example,
Student3 has a right to the object "average." We can make this right a "conditioned access right" like follows:

A[student3, avg] = {(read, end_of_semester's_date >= current_date)}

<u>Can we determine if an Access Control Matrix can ever go bad?</u>

Question) Given an initial matrix A along with the set of conditional commands and a safety condition ‘r’ $\notin$ A[d,O], does there exist an algorithm to determine if A could ever violate this condition?

Answer) The Harrison-Ruzzo Ulman theorem says No. The proof is given by embedding a Turing Machine in A and invoking Rice’s Theorem.

CSE 509 2/7

Announcements
- No late Reviews
- Volunteer for notes

Last time: <u>Decidability of Access Control</u>
*decidability embodied in access matrix

	File 1	File 2	CDROM
Domain1		write	eject
Domain2	grant read		
Domain3		read	burn

columns ~ objectsss
rows ~ domains

<u>Generic form of commands:</u>

```
command($d_1,\dots,d_n,o_1,\dots,o_m,r_1,\dots,r_k$)
 if( $r_1' \in A[d_1', o_1']$ and
     $r_2' \in A[d_2', o_2']$ and …
 then
        primitive op1;
        primitive op2;
              .
              .
        primitive opl;
```

Primitive operations: insert *r* into A[d, o]
remove *r* from A[d, o]

Primitive ops are :
Insert write
Delete write
Create/destroy object
Create/destroy domain

An Access Control System, $\Sigma = (A_0, C)$ consists of an initial matrix, A_0, and a set of commands, C.

Q1: Given an access control, Σ, and a safety property $r \notin A[d_i,o]$, does there exist an algorithm that can always determine whether A can be transformed, using commands from C, into a matrix that violates the safety property?

Theorem: (Harrison, Ruzzo, Ullman)
The answer to Q1 is "No."
*This means that you can not always determine if the safety property of access control will not be violated.

Proof: Embed a Turing machine in 'A', invoke Rices Theorem.

*Theorem is saying that for any sufficiently complex access control, we can not predict what will happen.

Q: are there ways we can design a system that is not so complex that we can determine is the safety property is always working?

<u>Naive Idea</u>
*Why not make a Rule that you can never add *r* into $A[d_i,o]$. Then it is easy to prove that the safety is never violated.

*In real world systems with hundreds or thousands of domains, having all these add on hacks becomes impractical.

ACM are Discretionary Access Control Sytem, i.e. the users of the system can mutate the ACM in any way the see fit, within the limits of C.

<u>Mandatory Access Control</u>
limit what users can do even
with there own objects.

<u>Belle – La Padula scheme</u> (idea comes from military, predates computers)

Military: Controls information through clearances.

Top Secret
|
Secret
|
FOUO
|
Public

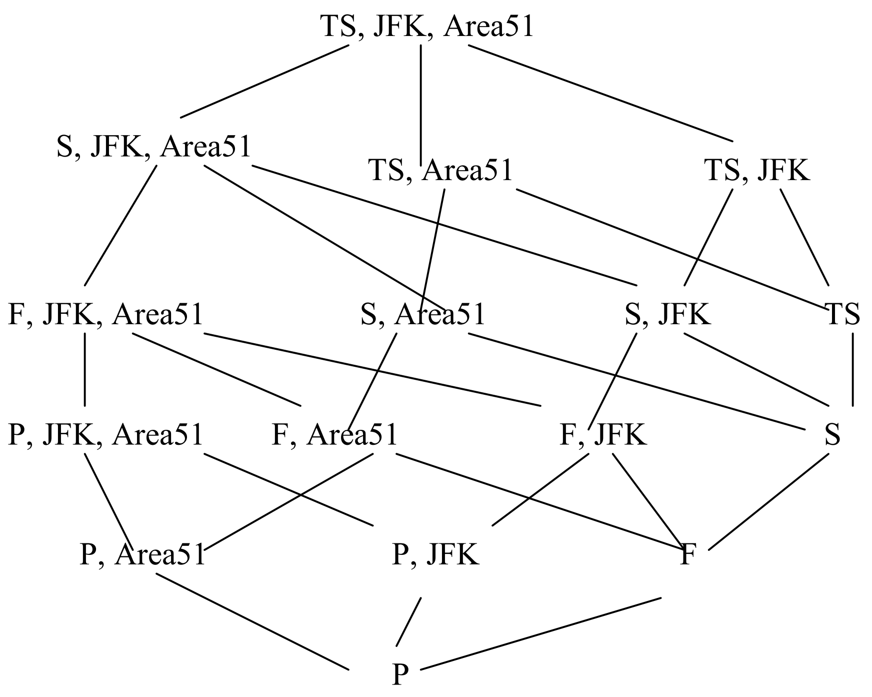
↙ higher priorities can access lower priorities.

<u>Compartments</u>
JFK Area51

Even with top clearance still need special tag to read JFK or Area51

Objects labeled with (h,C)
Domains labeled with (h_d, C_d)
D can read *o* if
$h_o <= h_d$ and $C_o \subseteq C_d$

<u>Lattice</u> TS = Top Secret, S = Secret, F = FOUO, P = Public



<u>Definition 1</u>
label $l_1 = (h_1, C_1)$
dominates $l2 = (h_2, C_2)$
write $l_1 >= l_2$ if
$h_1 > h_2$ and $C_2 \subseteq C_1$

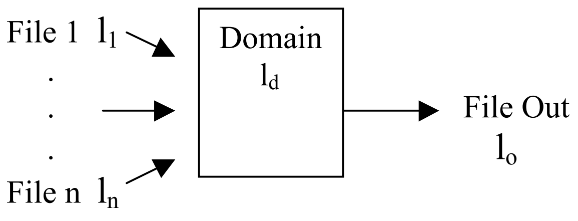
<u>Definition 2</u>
For any tow labels
L_1 and l_2
leastUpperbound$(l_1, l_2) = l_3$
where
$l_1 <= l_3$
$l_2 <= l_3$
and l_3 is smallest

* we have a notion of what to do for reads, but what about writes?

Writes
* (if a domain could read a TS file is it automatically safe to allow it to write to a public file? Seems dangerous: *siphoning secret information into public access files.)*

Domain 1 can only write to a file *o* where $l_o >= l_1$
*(integrity may be an issue, but we are
only concerned with secrecy right now.)

Basic Security Theorem
If domains can only read files with lower labels and only write to files with higher labels than no information can ever leak from its original classification level.



$l_i <= l_d <= l_o$
$i = 1, \ldots, n$

Variants
1.) –if a process tries to read a file of higher clearance => DENY
--write file of lower clearance => DENY
2.) a.) Domains label = LUB(read Files)
b.) Files written by domain are given the LUB(domain label, file's old label)

ex. %cat A B –o C A: Secret B: Top Secret C: does not exist yet

1.) cat starts: l_{cat} = Public
2.) cat opens A: l_{cat} = LUB(l_{cat} , A) = Secret
3.) cat opens B: l_{cat} = LUB(l_{cat} , B) = Top Secret
4.) cat writes C: l_{cat} = l_{cat} = Top Secret

Issue: Declassification
- mark some special *trusted* processes for declassification
-

e.g. grades(TS) → $\boxed{\text{AVG}}$ → Avg. (Public)

e.g. separation of privilege, i.e. a program that requires two users to invoke.

Access Control: Biba, Capabilities, Revocation

Biba Model:
- Similar to Bell-Lapadula Model but for integrity of information
- Bell-Lapadula upside down.

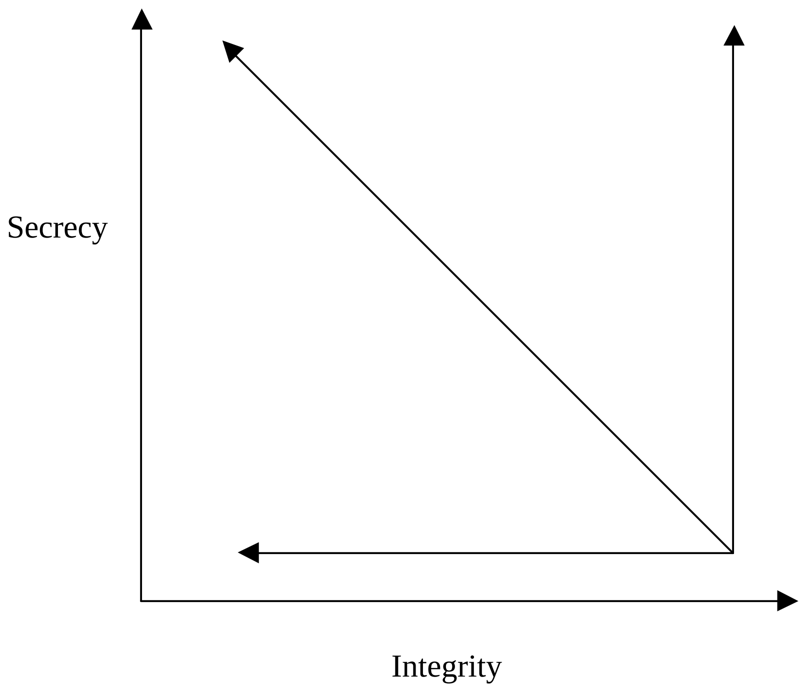
How trust worthy is information obtained from a specific source?
Double checked = Reliable witness = Anonymous tip = Internet
Double checked information is more trust worthy than the information obtained from a reliable witness and the information from the Internet is the least trust worthy.

Label objects with their integrity level as l_o
Label domains with their integrity level as l_d

Domain d can read object o if $l_d = l_o$
Domain d can write object o if $l_d = l_o$

Integrity is secrecy upside down.
Secrecy – No write DOWN
Integrity – No write UP

Flow of information w.r.t secrecy and integrity:



Implementation:
-Bell-Lapadula and Biba
    - Implemented as labels on files and processes
- Access Control Matrix (ACM)
- Access Control List (ACL)

- Store columns of ACM as linked list
- Store them with the objects
    - Ext2, ext3, NTFS, Andrew FS

-Capability List

- Store row of ACM
- Stored with the domains
- Very Rare
    - Pain to administer
    - Psychological acceptability
    - Domains are fuzzy
    - A general question about who all can access this file is hard to answer in this sort of representation.

Bell-Lapadula, Biba and ACMs all let OS manage privileges.

Confused Deputy Problem:
Ex: Printing facility at a Hotel : lpr daemon maintains a billing file in /var/lpr/txns

- Only lpr has read/write access to txns file
- User can write his/her transactions using lpr –b mytxns
- What will happen if a user gives the command lpr –b /var/lpr/txns?

```
        lpr                                              OS
open("/var/lpr/txns", R)    ß--------OK----------à    R/W /var/lpr/txns
<generate user's bill>
open("/var/lpr/txns", W)    ß-------OK-----------à    R/W /var/lpr/txns
<write bill to output file>
```

By the above sequence of steps transactions file will be left with the transactions of the user alone, by a series of smart steps all the user transactions can be deleted from the file.

Solution to the above problem:

1. lpr should check if the output file is user accessible
2. lpr should tell OS which privilege it is trying to use

```
        lpr                                              OS
open(1, "/var/lpr/txns", R)    ß--------OK----------à    R/W /var/lpr/txns
<generate user's bill>
open(2, "/var/lpr/txns", W)    ß-------ERR-----------à    R/W /var/lpr/txns
```

Implementaion:

- Use unforgeable handles, i.e handles that only OS can create and pass to applications. Ex: File handles
- Applications can pass the handle to OS with each request.

fd = open("foo", R) – Only way to get a file descriptor is through OS
read(fd, ….)

write(fd, ….) – Applications have to pass the file descriptor to OS for other requests

In this type of implementation, revocation is easy, all you need to do is remove the entry from the Application's file table.

Cryptography – Signature

- Signature = Sig(Secret Key, Data)
- It is not possible to create the signature of a data without knowing the secret key
- Capabilities are signed tickets from OS

cap = open("foo", R).
- OS checks if the application can read foo.
- OS contructs cap as cap = ("foo", R, signature)

read(cap, ….)
- OS checks if the capability allows R
- OS checks if the signature is valid.

A feature/bug of the above procedure is that applications can copy capabilities.

Revocation is harder.

- Have an expiration date for capability.
    1. No fast revocation
    2. clients must renew their capabilities
- OS maintains a capability black list
    1. OS should maintain the state forever
    2. checking is harder

CSE509 Spring 2006. System Security

Lecture Notes for 02/14/2006
Authentication

Notes: Blue line notes are added by me (Wenbin Zhang).

So far we have learnt about:

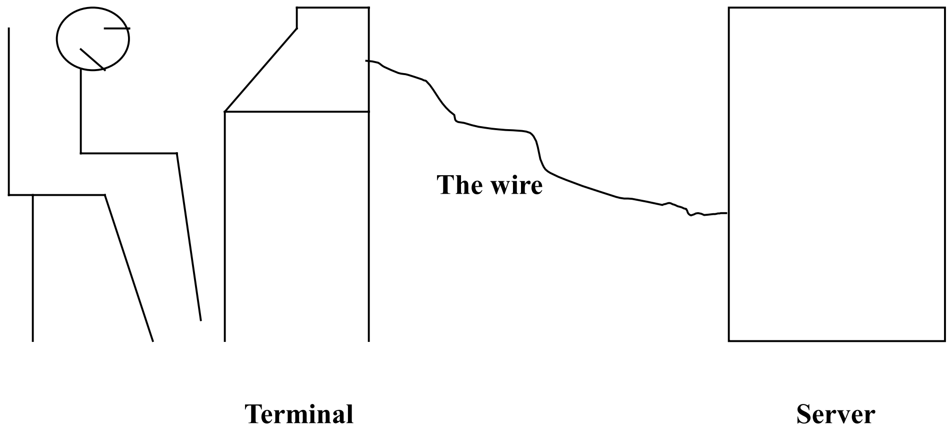
- Controlling access to objects from security domain
- What about people's access to the computer

Today we will talk about Authentication.

The goal of security:

- Map user to domain
- Identify user
- Prove user is who they claim to be (authentication)

Thinking about the scenario in below picture:



What can be happened?

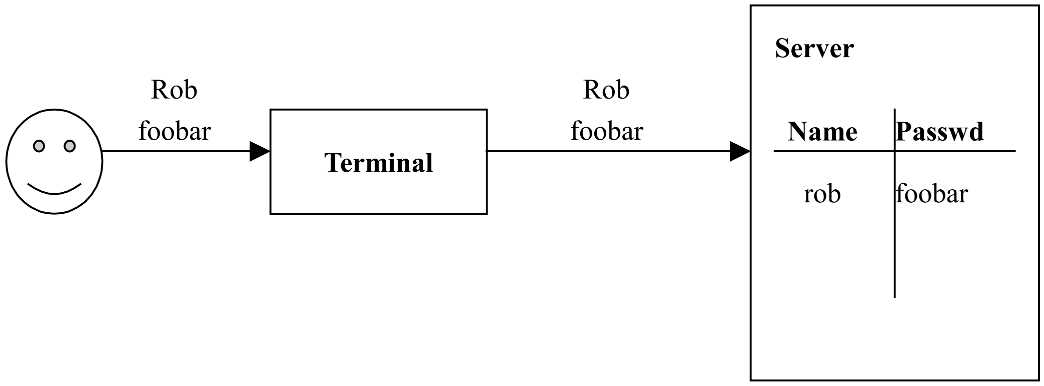
Threats:

- Wiretapping
  - Sniffing
  - Spoofing
- Terminal Compromise
  - Key logging
  - Other covert channels
  - Spoofing
- Server attacks
  - Attacker may break into server

Three kinds of authentication:

- Something you know
- Something you have
- Something you are

The simplest authentication scenario:



The problems in this scenario:

- Password is stored in clear on server
- Password is sent in clear
- Terminal can steal password
- Human factors in passwords:
  - Easy to guess
  - Automatically generate password
  - Shareable
  - Stealable

How to fix these problems?

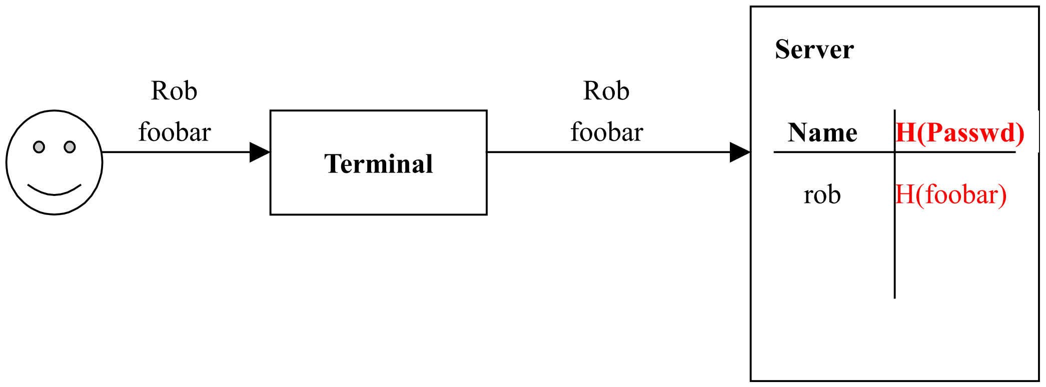
Define: A hash function H is pre-image resistant if, given y = H(x), it is extremely difficult for attackers to find x', s.t. H(x') = y.

E.g. SHA-1, SHA-256

*The **SHA (Secure Hash Algorithm)** family is a set of related cryptographic hash functions. The most commonly used function in the family, SHA-1, is employed in a large variety of popular security applications and protocols, including TLS, SSL, PGP, SSH, S/MIME, and IPSec. SHA-1 is considered to be the successor to MD5, an earlier, widely-used hash function. The SHA algorithms were designed by the National Security Agency (NSA) and published as a US government standard.*

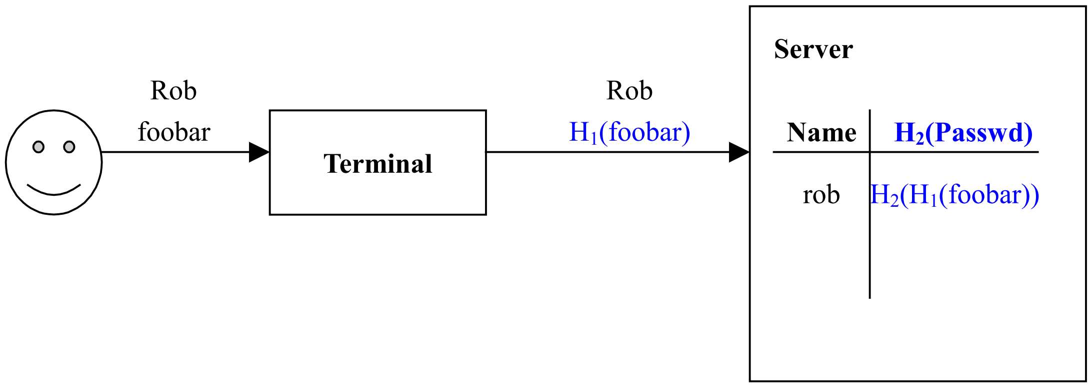
The first member of the family, published in 1993, is officially called SHA; however, it is often called SHA-0 to avoid confusion with its successors. Two years later, SHA-1, the first successor to SHA, was published. Four more variants have since been issued with increased *output ranges and a slightly different design: SHA-224, SHA-256, SHA-384, and SHA-512 — sometimes collectively referred to as SHA-2.*

Example1:



Solved the problem "Password stored in clear on server".

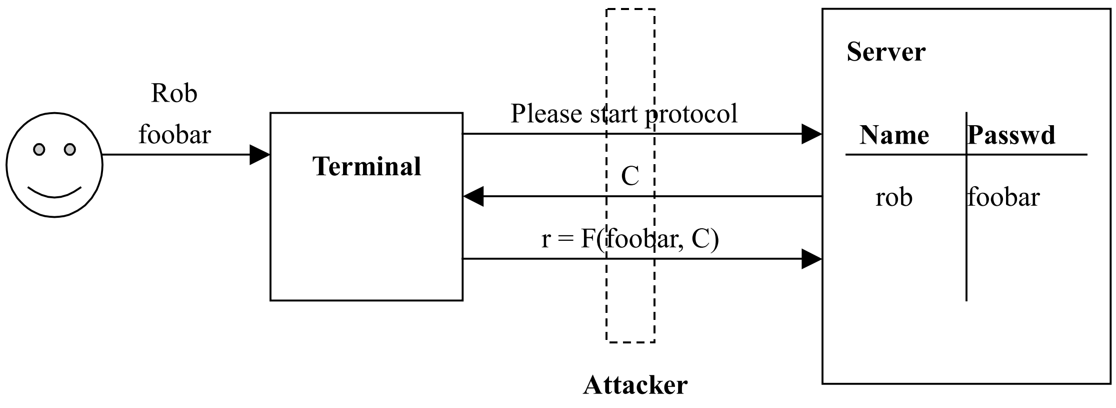
Example2:



Note: In this case, the sender doesn't reveal foobar but attacker can see H_1(foobar), which is enough to login.

Data on the wire:
Encryption: Encrypt & MAC all data sent between the terminal and the server.

Challenge-Response protocol:



Security property of F:

Suppose p is password, given

$F(p,C_1)$
$F(p,C_2)$
……
$F(p,C_n)$

For $C_1, C_2, \ldots C_n$ of the attacker's choosing, it is extremely difficult for the attacker to produce pair (C, r), s.t. $r = F(p,C)$ and $C \neq C_1, C_2, \ldots C_n$.

Server:

1. Set t = current time.
2. Send C.
3. if response arrives within 3 seconds, success.

Attacker:

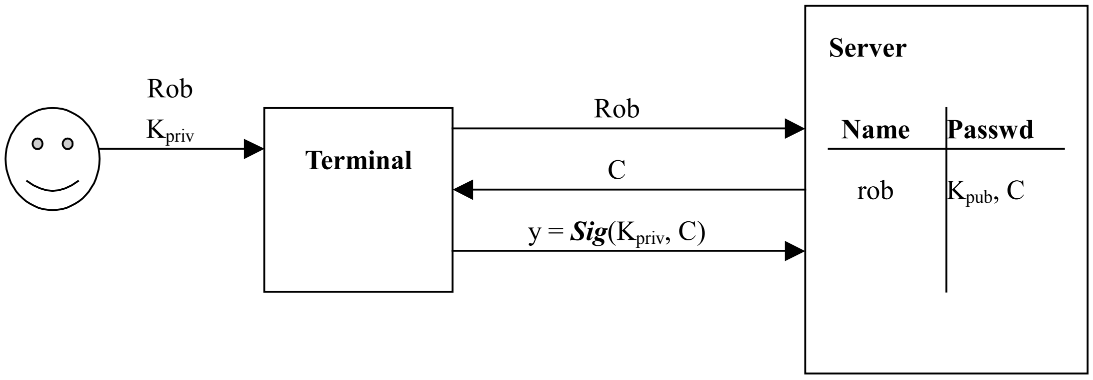
1. Pass C to terminal.
2. Grab response and stop it.
3. Let user try again, then success.
   --- later ---
4. Present original response to server.

Note: Challenge will not expire.

Digital Signatures:

- User possesses a private key K_{priv}, only the user knows K_{priv}.
- World can know a corresponding public key, K_{pub}.
- Extremely difficult to compute K_{priv} from K_{pub}.
- A signature scheme is two functions ***Sig, Ver***, s.t.
  Ver(K_{pub}, M, ***Sig***(K_{priv}, M)) = valid
  But an attacker cannot construct any value X, s.t.
  Ver(K_{pub}, M, X) = valid.

Example1:



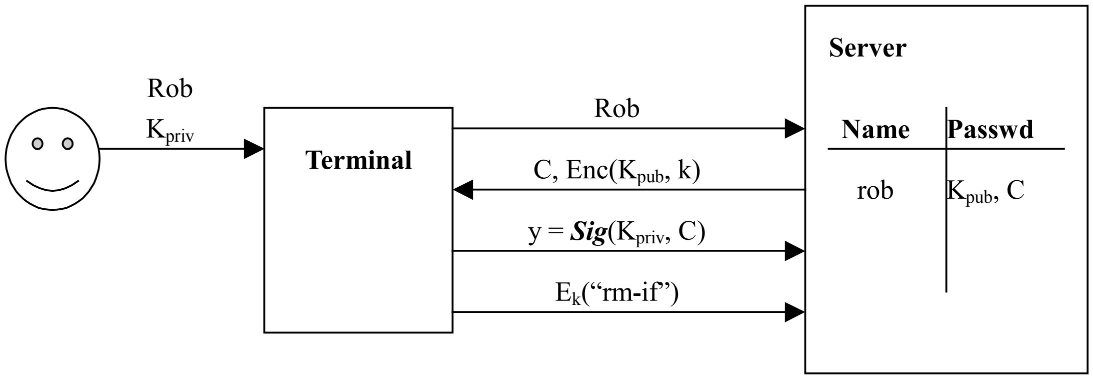
Server check that
Ver(K_{pub}, C, y) = valid

Problem: No binding between initial authentication and subsequent communication.

How to fix:

- Sign all subsequent message, too
- Include C for freshness

Example2:



For real password protocols, please see:

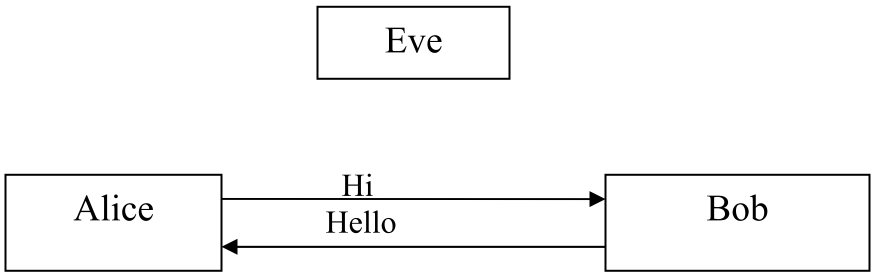
- Encrypted Key Exchange (EKE)
- Diffie-Hellman EKE (DH-EKE)
- Secure Remote Password protocol (SRP)
    - Described in RFC2945: http://rfc.net/rfc2945.html

Lecture Notes for 02/16/06
SYMMETRIC KEY CRYPTOGRAPHY

(Green part is taken from: http://en.wikipedia.org/wiki/Data_Encryption_Standard)

Symmetric Key Cryptography:

Typically there are two parties involved in an encryption system. For example, let us have the following scenario of two people sending messages amongst themselves:



Assume that Alice and Bob want to send messages without a third party knowing. Assume that Eve is listening and she wants to read the messages that go between Alice and Bob and if she succeeds she would be violating the *confidentiality* goal of security.

Why Encryption?

So, what is the main reason for having an encryption system? It is to preserve confidentiality. If the messages that are sent between Alice and Bob are in clear text, then Eve can easily read them. Hence, for security, the messages sent should be encrypted.

Intuition:

Encryption should give us the same secrecy as sealed envelopes. If Bob places a letter in an envelope and he seals the envelope and sends it to Alice, then, Eve would not be able to read the letter in the sealed envelope.

It should be noted that encryption does not hide communicating but it hides what is being communicated. Hence, with our example, if encryption is implemented, Eve might know that a message is being sent to Alice, however, she would not be able to read it. The purpose of encryption is to be able to send messages secretly, even if a third party knows that a message is being sent and he knows some side info about the people who are sending messages.

Types of Symmetric Key Cryptography:

1) **One-Time Pad:**

One time pad has the following components and parameters:

- M = Message of length “l”
- K = Key of length “l”
- For the Key, each bit = 0 with probability = ½. So, in other words each bit for the key is chosen randomly.

How can we encrypt?

- $C = M \oplus K$.
- C stands for Cipher Text.
- K is a string of bits.
- The Cipher Text can be obtained by performing the “XOR” operation between M and K.

Applying this to our previous example, we see that for preserving secrecy, both Alice and Bob must know K for the encrypted message, but, Eve must not know K. In order to come up with the Key, both Alice and Bob have to think of and agree on a set of bits that are arranged randomly. So, if Bob wants to say Hi, he would change “Hi” to ASCII and then XOR the bits with the Key bits and then send the cipher text.

An important thing is to note that in this system, each bit of key is used only once. This brings us to the following theorem:

Theorem:

A One-Time Pad is secure against a computationally unbound third party. So, no matter how long the third party can sit computing, he/she cannot read the message.

One-Time Pad Is Not Practical. Why?

- Key must be as huge as message.
- Cannot reuse bits of the Key.

What Happens If We Reuse The Key Bits?

Suppose we send two messages M1 and M2, such that:

$C1 = M1 \oplus K$ and

$C2 = M2 \oplus K$.

What would happen?
Let us try computing the following:

$$C1 \oplus C2$$
$$= M1 \oplus K \oplus M2 \oplus K$$
$$= M1 \oplus M2.$$

So, now if Eve knows some side information, for example, that M1 and M2 are in English, then she can recover M1 and M2.

Hence in a One-Time encryption system, the Key can **NEVER** be reused.

2) BLOCK CIPHER:

A block cipher has the following components and parameters:

- A function: E (K,M) = C.
  (A function that takes as its parameters a Message 'M', and a Key 'K' and gives back a Cipher Text, 'C'.)

- The function should be such that given 'K', it should be easy to compute E (K, M).

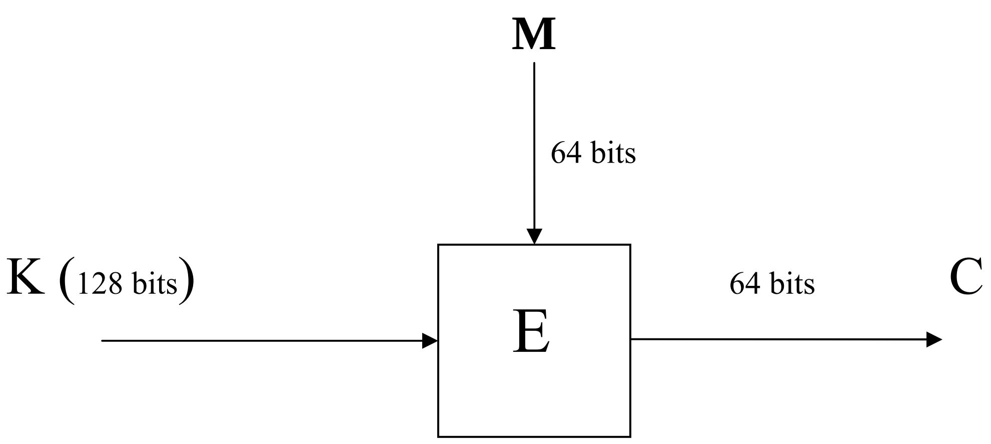
- Moreover, given the Key and Cipher Text, it should be easy to compute the Message. So symbolically, the following should be easy to compute:
  $$M = E^{-1} (K, C).$$

- Block Ciphers work on blocks.

- The function E takes a k-bit Key and n-bit Message and gives back an n-bit cipher text. So, symbolically,

$$E: \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\}^n$$

The following is a simple figure that summarizes the encryption method:



The above figure shows a cipher block encryption that takes a 64 bit Message and a 128 bit key, and gives a 64 bit cipher text.

The following are examples of block ciphers:

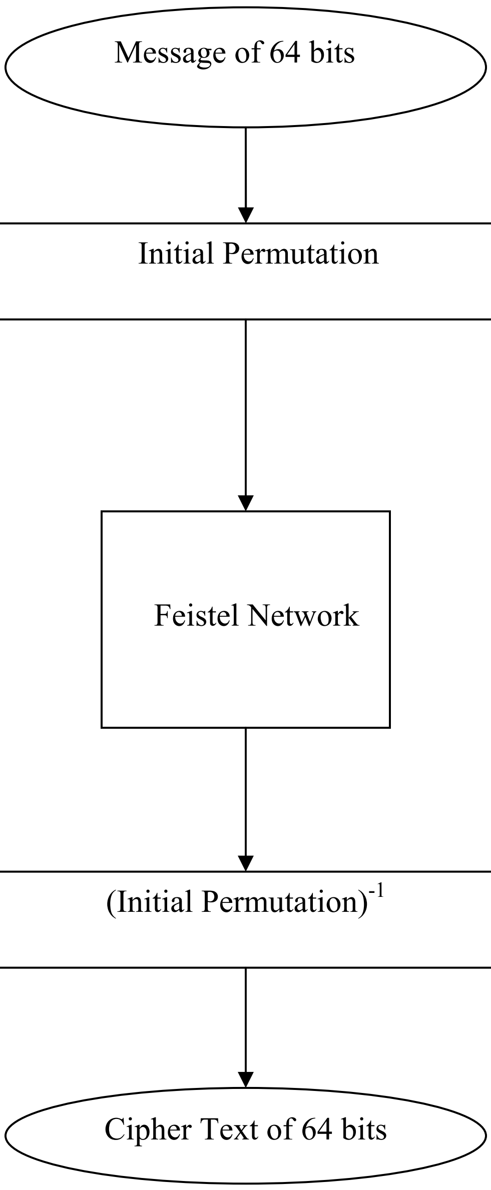
3) <u>DES (Data Encryption Standard):</u>

DES was invented by IBM and NSA in the 70's.

The following is a simple description of DES:

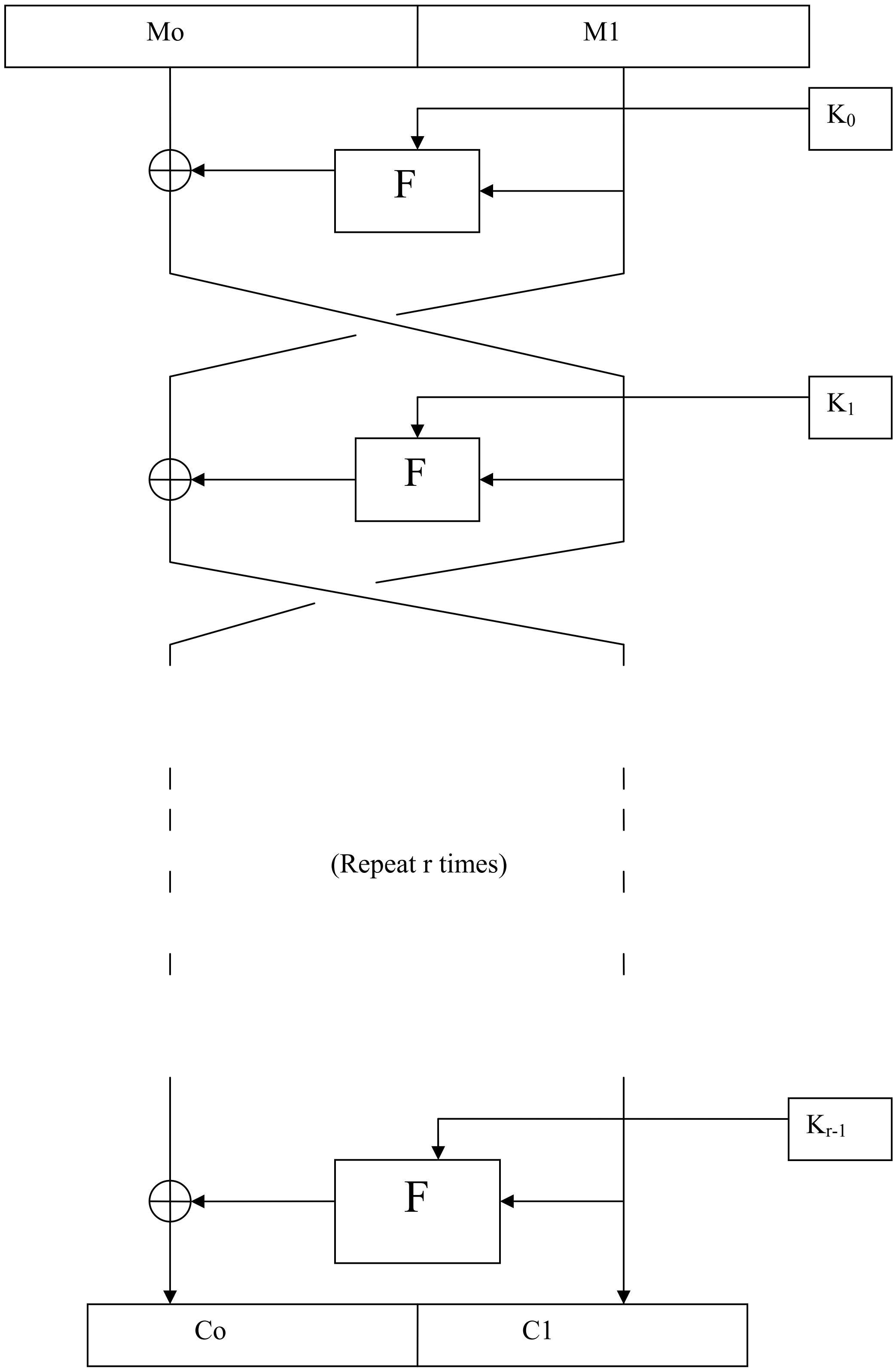
- The Key should be 56 bits.
- Block size should be 64 bits.
- DES is an example of a Feistel Network.
- Given the Key and Message it is easy to compute the cipher text. Moreover, the inverse is also true, i.e., given a cipher text and the key it is easy to compute the message. However, for people who don't know the key, it is hard for them to get the message from the cipher text, or vice versa.
- The message undergoes an initial permutation before entering the Feistel Network.
- The Cipher Text that is computed by the Feistel Network undergoes a final permutation which is the inverse of the initial permutation.

The following is a figure that shows how DES is structured and how it operates:



Now, in the Feistel Network, the main 64-bit message is split in two 32-bit blocks and they are processed, (in a cress-crossing manner,) with the help of Feistel, or "F", Functions. The *F-function* scrambles half a block together with some of the key. The output from the F-function is then combined with the other half of the block by using the XOR function, and the halves are swapped before the next round.

The following is a picture of what happens inside a Feistel Network:


Mo
M1
K0
F
K1
F
(Repeat r times)
Kr-1
F
Co
C1

Question) Did the permutations before and after the Feistel Network make the breaking of the code any harder?

Answer) No, These permutations are pointless in terms of making the encrypted message harder to break. The permutation is not secret, and hence, in terms of security it is a waste of time.

Successful Attacks On DES:

- DES is vulnerable to an attack called "Differential Cryptanalysis."
- The number of bits required for the key is known. It is 56. So, we need to try 2^{56} different keys to break the system.
- With a super computer this can be done within a day's limit.
- EFF built a DES cracker that takes less than 24 hours. (Just need one cipher text to break the system.)

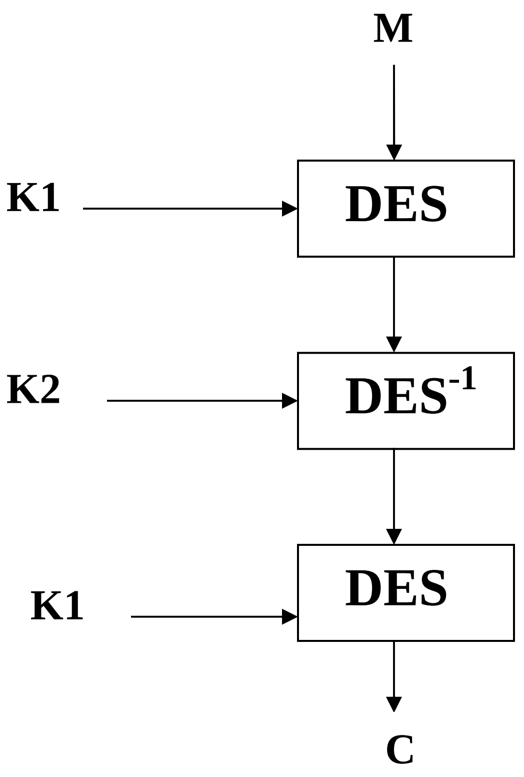
Conclusion:

DONOT use DES as an encryption system!

Most Block Ciphers are not completely safe nowadays. In order to extend the life of a DES people came up with the following slightly better system:

4) 3DES (Triple Data Encryption Standard):

This system uses more key bits. The following is a picture that illustrates how 3DES works. We have a 112 bit key for the following system:



(If we would have had K1 instead of K2 then we would have had a DES system)

<u>Possible Attacks on This System:</u>

This system is vulnerable to plain-text and chosen-text attacks. Hence, this is not that great either.

Moreover, this system also proved to be very slow.

During the 90's a new block cipher known as AES was chosen as an encryption standard by the u.s. government.

5) Advanced Encryption Standard (AES):

- Another name for AES is Rijndael.
- This is a special design that uses Algebra, and it is fast and secure.

AES has the following parameters:

- The key modes are : 128-bit key mode,
  192-bit key mode and
  256-bit key mode.

- The block size is: 128 bits.

So the weakest version of an AES system would have a 128 bit key. Even this is very hard to break and would take probably thousands of years. Hence, AES is a fine cipher.

Now, all these ciphers described above are block ciphers. What should we do if we have to encrypt more than 1 block? In order to encrypt more than 1 block we have the following Modes of Operation:

Modes of Operation:

There are three modes of operation:

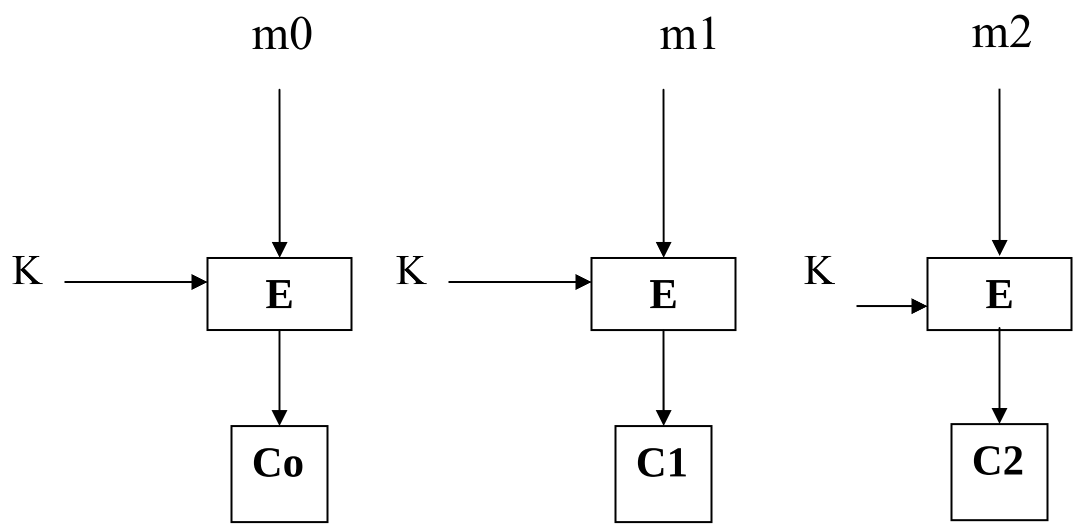
- ECB (Electronic Code Book.)
- CTR (Counter Mode.)
- CBC (Cipher Block Sharing.)

1) ECB:

- Also known as Electronic Code Book.
- It is like a dictionary. To encrypt you look up a word and see what it is translated to. To decrypt you go oppositely, i.e., you look up the cipher text and see what message it corresponds to.
- NOT secure.
- Does not work like an opaque envelope.
- Same message → same cipher text.
- Needs randomization.
- Not good to use.

The following is an illustration of ECB:

`

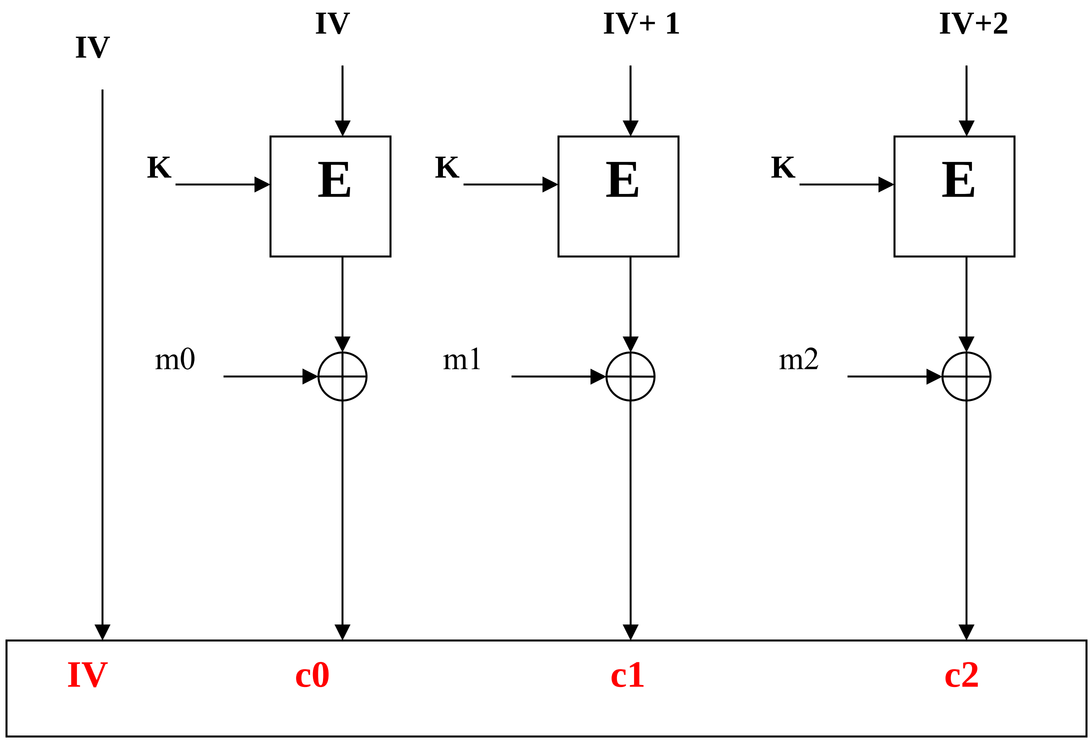


(The above is an illustration of ECB)

2) CTR:

- Also known as counter mode.
- Secure against chosen plain text attacks
- This system has an Initialization Vector "IV," which is a number that is chosen randomly for each cipher text.
- If you use same IV twice then that destroys security, hence, the space of IV should be very large.
- CTR is similar to One-Time Padding.

Below is a picture that illustrates the CTR Mode of Operation:

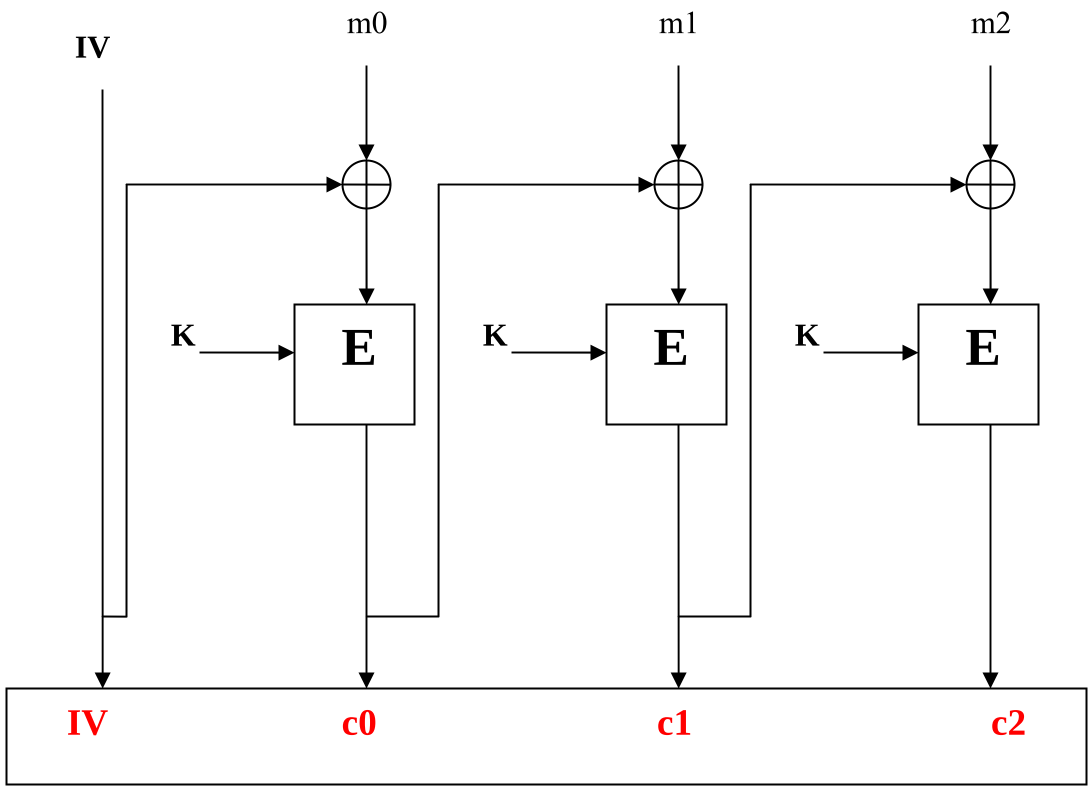


(The elements in red show the output that would be sent)

<u>CBC:</u>

- Abbreviation for Cipher Block Chaining.
- In this mode, each cipher block is chained with the next one.
- Secure against chosen plain text attacks.
- Most Common.

Below is a picture that illustrates the CBC Mode of Operation:



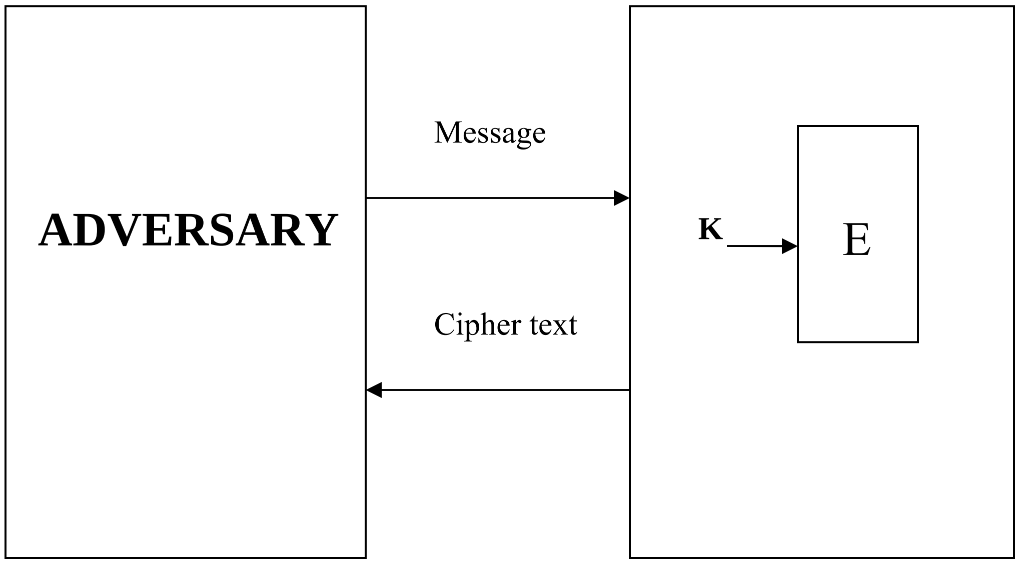
<u>Attack Models Against Cryptosystems:</u>

- <u>Passive Adversary:</u>
  - Weakest form of attack.
  - Observe Cipher texts and
  - Try to infer the plain texts or possibly the key.

- <u>Known Plain Text Attack:</u>
  - Attacker knows the plain text that is given to the cryptosystem, and knows the cipher text.
  - Tries to decrypt future messages.
  - Attacker cannot give inputs to the cryptosystem.

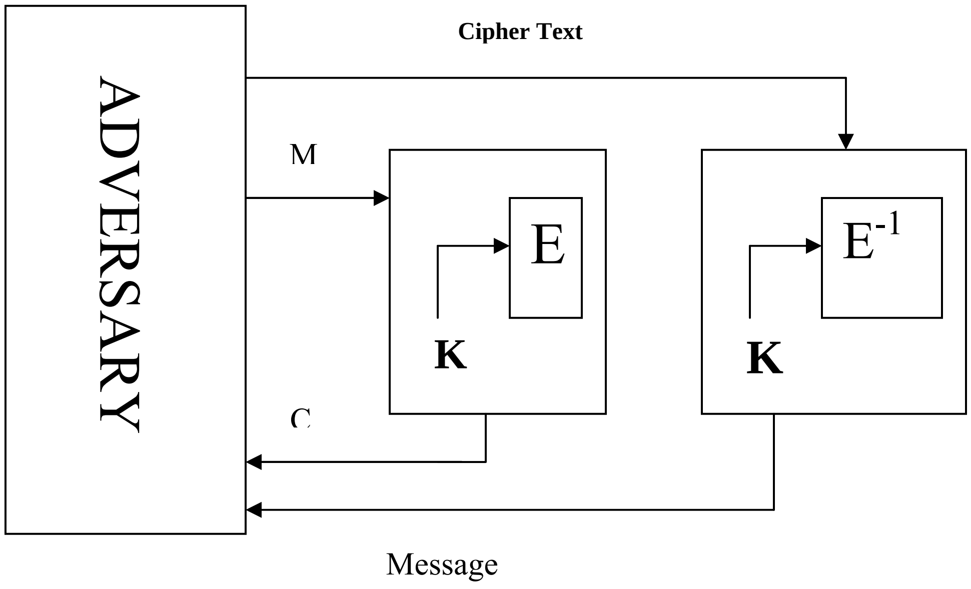
- <u>Chosen Plain Text Attack:</u>
  - Adversary can request cipher text corresponding to any plain text.
  - Then adversary tries to decrypt any future messages.
  - So, it can be thought of as, the adversary is given access to a black box, and adversary is allowed to send queries to the black box and get cipher texts as output from the black box.

The following is an illustration of this case:



(Chosen Plain Text Attack)

- <u>Chosen Cipher Text Attack:</u>
    - This is the most powerful mode of attack.
    - Adversary can request encryption/decryption of any plain text/cipher text.
    - Then adversary tries to decrypt any future messages.

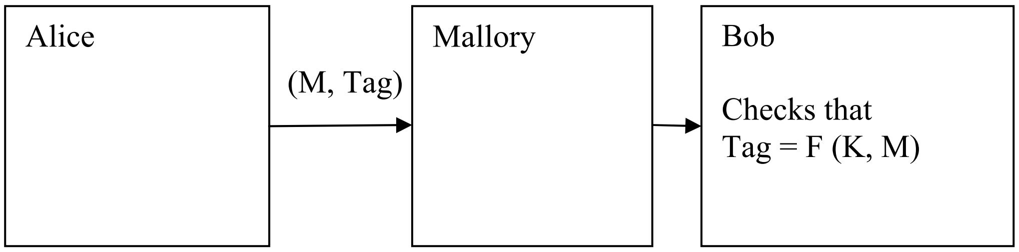


Faisal Islam

Notes for Tuesday, 21 Feb 2006

Symmetric Key Integrity Mechanisms

Message Authentication Code (MAC)



Goal: Mallory can't modify or construct valid messages.

- Replay Attacks
- Garbage Attacks

One time Pad:

$C = C_0, C_1, \dots$

$K = K_0, K_1, \dots$

$M = K_0 + M_0, K_1 + K_2, \dots$

Bit flipping can be done to change it.

MAC

$F(K, M) = Tag$

$F: \{0, 1\}^P \times \{0, 1\}^* \to \{0, 1\}^M$

- Mallory can't guess Tag for any message.
- Mallory can't combine previous messages/tags to construct new messages.

Hash Functions:

H: $\{0, 1\}^* \to \{0, 1\}^M$

H (M) = h

- given y, infeasible to find any x such that H (x) = y.
- infeasible to find x and x', such that H (x) = H (x').

Examples

MD4, MD5, SHA-0, SHA-1: Has been broken

SHA-256, SHA-512: Hasn't been broken yet

RIPEMD – Okay to use

HMAC

H (K$\oplus$opad || H (K$\oplus$ipad || M))

ipad = Ox36 x 64 bytes

opad = Ox5c x 64 bytes

K is any L bit key. Example: HMAC - SHA-256

Fact: If H is secure then HMAC-H is secure.

You can compute H (M_1 || M_2) from H (M_1) and M_2

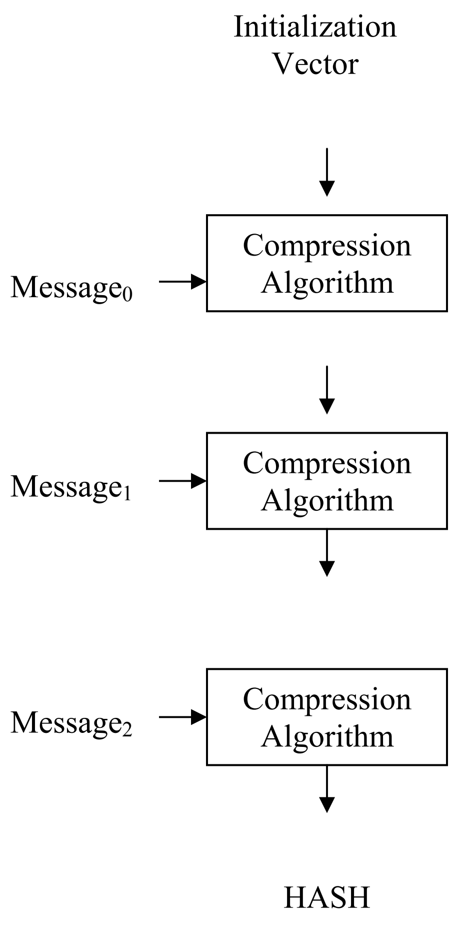
MAC (K, M) = H (K || M)

Mallory sees: Tag =   (K || M)

Then Mallory can compute:

H ((K || M) || M') = MAC (K, M || M') From H (K || M) and M' = Tag

Hash



To send M secretly and unalterably I can:

1. Encrypt – then- MAC

   $C = E_k(M)$

   $T = MAC_k(C)$

   Send (C, T)

2. MAC – then – Encrypt

   $T = MAC_k(M)$

   Send $E_{k'}(M \| T)$

3. MAC and Encrypt

   $T = MAC_{k'}(M)$

   $C = E_k(M)$

   Send (C, T)

(3) does not work because it gives attacker double opportunity to get M.

(2) does not work under some abnormal settings. (Ask Professor)

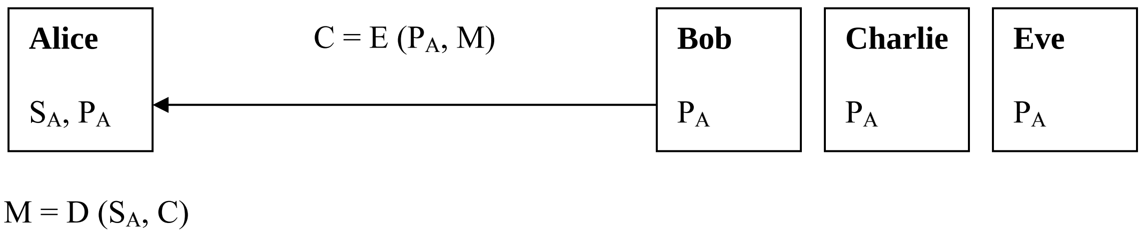
(1) as of yet still works. It has not been broken.

Public Key Cryptography

So far, Alice & Bob both know K

- can't distinguish Alice from Bob
- either party can betray other
- for n parties need $^nC_2$ key < n^2

In Public Key Cryptography, every person has a key that just they know.



Alice

buys a safe

- keeps keys
- mails open safe to Bob
- uses secret key to open the safe when received

Bob

- stuffs M in the safe
- closes door

Number Theory

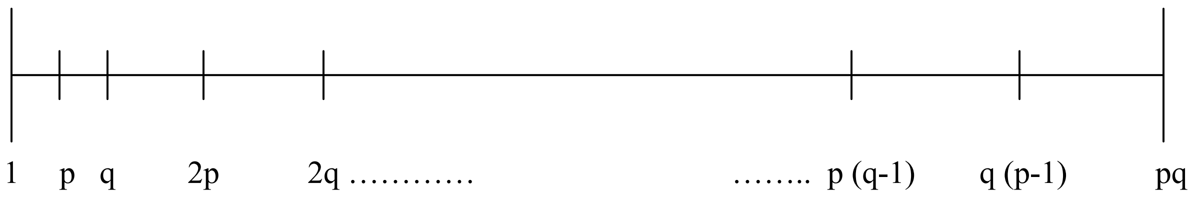
Defn: $Z_n^* = (Z /_n Z)^*$

$= \{r \mid 0 < r < n, \gcd(r, n) = 1\}$

Defn: $\varphi(n) = | Z_n^*|$ Example: $\varphi(7) = 6$ $\varphi(p) = p-1$

$\varphi(pq) =$??

$Z_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$ $\varphi(15) = 8 = (3-1)(5-1)$



$\varphi(pq) = pq - (q-1) - (p-1) - 1$

$= pq - q - p + 1$

$= (p-1)(q-1)$

Defn:

Let, $*: Z_n^* \times Z_n^* \to Z_n^*$

by $a*b = a \times b \bmod n$

$n = 15, a = 13, b = 7$

$a*b = 1$

Defn:

A group is a non empty set G with a binary operator o such that,

(i) $\exists e \in G, \forall a \in G, e \circ a = a$ [Identity]

(ii) $\forall a \in G, \exists b \in G, a \circ b = e$ [Inverse]

(iii) $\forall a, b, c, (a \circ b) \circ c = a \circ (b \circ c)$ [Associatively]

Z_n^* is a group with *

Identity: 1

Is associative

Inverses: Given a find b such that $(ab \equiv 1) \bmod n$

Iff $ab = kn + 1$

$ab - kn = 1$

Number Theory (continued), Public Key Cryptography

Continuing from the last lecture – finding an inverse:

We want to find the inverse of a number a in (Z / nZ)*

i.e. given a, we want to find b such that $a * b \equiv 1 \bmod n$.

Now, $ab \equiv 1 \bmod n$ <==> $ab = kn + 1$ for some k
i.e. $ab - kn = 1$

We can find b (and k) using the Euclidean algorithm.

Euclidean algorithm:

The following equations hold true:

$1 \text{ x } a + 0 \text{ x } n = a$ (1)
$0 \text{ x } a + 1 \text{ x } n = n$ (2)

From the definition of (Z / nZ)* we know that $a < n$.
We now find a value m_1 such that $m_1 a \leq n$
but $(m_1 + 1)\, a > n$
So $m_1 a$ is the largest multiple of a that is less than n.
i.e $m_1 = \lfloor n / a \rfloor$

Multiplying equation (2) by m_1 and subtracting the result from equation (1), we have

$-m_1 a + 1.n = n - m_1 a$

Say $d \mid a$ and $d \mid n$ [$x \mid y$ indicates that x is a divisor of y]
If d is a divisor of a and n, then d is also a divisor of d $(n - m_1 a)$
i.e. $d \mid (n - m_1 a)$

Fact 1: Every divisor of a and n is a divisor of $(n - m_1 a)$
[See sidebar for more.]

So in the Euclidean algorithm, the set of divisors of the right hand side of the equations is preserved at every step.

Fact 2: The value on the RHS of the equations gets smaller at each step.

Fact 3: Eventually, only the divisors will be left on the RHS. Then, the RHS is the gcd of (a, n).

All equations are of the form:

$x.a + y.n = r$

Sidebar:
We can also show that every divisor of a and $(n - m_1 a)$ must be a divisor of n:

Consider:
$n - m_1 a = xd$ for some x
i.e. $n - m_1 yd = xd$ for some y
(since both a and $n - m_1 a$ are some multiple of d)

Then,
$n = xd + m_1 yd$
$= d(x + m_1 y)$

That is, n is also a multiple of d.

Eventually, r will become the gcd of a and n.
i.e. it will become 1 for $a\ \varepsilon\ (Z / nZ)^*$

Therefore, we will get $x.a + y.n = 1$

Example:
We want to find the inverse of 37, modulo 61.
i.e. 37^{-1} mod 61

$1 \times 37 + 0 \times 61 = 37$ (1)
$0 \times 37 + 1 \times 61 = 61$ (2)

$m_1 = \lfloor 61 / 37 \rfloor = 1$

By (2) – m_1 (1) we get:
$-1 \times 37 + 1 \times 61 = 24$ (3)

Now consider (1) and (3).
$m_2 = \lfloor 37 / 24 \rfloor = 1$

By (1) – m_2 (3) we get:
$2 \times 37 - 1 \times 61 = 13$ (4)

Now consider (3) and (4)
$m_3 = \lfloor 24 / 13 \rfloor = 1$

By (3) – m_3 (4) we get:
$-3 \times 37 + 2 \times 61 = 11$ (5)

Now consider (4) and (5)
$m_4 = \lfloor 13 / 11 \rfloor = 1$

By (4) – m_4 (5) we get:
$5 \times 37 - 3 \times 61 = 2$ (6)

Now with (5) and (6)
$m_5 = \lfloor 11 / 2 \rfloor = 5$

By (5) – m_5 (6) we get:
$-28 \times 37 + 17 \times 61 = 1$ (7)

We now have 1 on the RHS.
(7) is of the form $b \times a - k \times n = 1$. Therefore, $b = -28$
A negative number (– 28) is not acceptable as the inverse. We add 61 (i.e. keep adding n until we get a positive number) to get 33.

Thus $37 * 33 \equiv 1 \bmod 61$

Efficient algorithm for computing exponentiation modulo *n*:

We need a way to compute $a^b \bmod n$
where b is huge (for instance, 2000 bits)

To do this, we write b in binary.

$$\begin{aligned} \text{eg. } b &= 1001 \\ &= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 1 \times 2^3 + 1 \times 2^0 \end{aligned}$$

Then we can write:

$$\begin{aligned} a^b &= a^{2\char`^3 + 2\char`^0} \qquad [\text{ x\^{}y indicates } x^y] \\ &= a^{2\char`^3} \times a^{2\char`^0} \end{aligned}$$

So we can compute the exponentiation as follows:

$a^{2\wedge 0}$	=	a	1
$a^{2\wedge 1}$	=	a^2	0
$a^{2\wedge 2}$	=	a^4	0
$a^{2\wedge 3}$	=	a^8	1

i.e. calculate product of bits of b with the corresponding exponent of a, and then compute the product of these sub-products. In the above case, in involves of maximum of 3 squaring operations and 3 multiplication operations.

In general, if b is n bits long, this method requires a maximum of $n - 1$ squaring operations and $n - 1$ multiplication operations.

<u>Note</u>: The individual exponents of a can also be stored modulo n because in general,
$xy \bmod n = [(x \bmod n) * (y \bmod n)] \bmod n$

<u>Algorithm</u>: to compute $a^b \bmod n$

1. Compute $a^{2\wedge i} \bmod n$
   for i = 0, 1, …. | b |
2. Take product of all $a^{2\wedge i} \bmod n$ where $b_i = 1$
   i.e. compute $\prod_{b_i = 1} a^{2\wedge i} \bmod n$

Using the mathematical background discussed, we can now define the RSA public-key cryptosystem.

RSA Public-key Cryptosystem:

<u>Key Generation</u>:

- Alice selects two large primes p and q. She then computes their product.
  $N = p \times q$
- Alice then selects a number e such that gcd $(e, \varphi(N)) = 1$
- Alice then computes d such that $e*d \equiv 1 \bmod \varphi(N)$

[Note that as defined earlier, $\varphi(N) = (p - 1)(q - 1)$]

Then Alice's public key = (N, e)
and Alice's private key = (N, d)

Encryption:

The encryption is done using the public key.
Let the message to be encrypted be $M \varepsilon (Z / nZ)^*$
The ciphertext C is computed as

$$C = M^{e} \bmod N$$

Decryption:

The decryption is done using the private key.

$$M = C^{d} \bmod N$$

$$\begin{aligned} \text{Because } C^{d} \bmod N &= M^{ed} \bmod N \\ &= M^{k\varphi(N)+1} \bmod N \qquad [\text{Because } e*d \equiv 1 \bmod \varphi(N)] \\ &= M^{(\varphi(N))\wedge k} \bmod N \\ &= 1^{k} . M \bmod N = M \qquad [\textbf{Theorem}: \text{For any finite group } G \text{ and } a \varepsilon G, a^{|G|} = 1] \end{aligned}$$

The most obvious attack on RSA is that a malicious user can try to find d from (N, e), i.e. try to get the private key from the public key. However, it can be shown that if the attacker can find d from (N, e), then the attacker can factor N.
(*Proof is a homework problem.*)

Therefore, N must be hard to factor.
For this, both p and q must be of about the same size.
Thumb rule: $p, q \approx$ square root of N

Also, N should be large. But how large?

- 300 bits can be factored in seconds on a PC
- 512 bits can be factored with a small cluster.
- 768 bits is conjectured to be factorable with $10 million

So we should make N significantly larger than this.
Today, usually N $\approx$ 2048 bits.

Strong RSA Assumption:
Given N, C, it is computationally infeasible to find M, e such that $M^{e} \equiv C \bmod N$.

The weaker assumption says that it is computationally infeasible to find M given C and N for a specific key pair (i.e. a specific pair of e and d). The stronger assumption lets the attacker choose e as well, i.e. it says that it is computationally infeasible to find *any* pair of (M, e) given C and N.

Common implementation practice:

In general, RSA is significantly more computationally intensive than symmetric key algorithms, because RSA involves a large number of exponentiation operations on large numbers.

Therefore, to make things easier, in practice, usually $e = 3$.
So encrypting M only requires doing M^{3}, i.e. one square and one multiplication. In this case, encryption is fast but decryption is slow.
(As $e = 3$, the value of d will be significantly larger. And despite a constant e, the value of d will be different for different values of N.)

It is inefficient to encrypt a large file completely using RSA. Therefore, a common practice to encrypt a large file M is as follows:

- Pick a key k
- Encrypt the file using any block cipher with the key k
  $C_0 = E_{\mathbf{k}}(M)$
- Encrypt the key k with the public key of the intended recipient.
  $C_1 = RSA_{\mathbf{Pub}}(k)$
- Transmit C_0C_1 as the ciphertext.

Then, to decrypt, the recipient would have to decrypt C_1 with her private key to obtain the key k and then decrypt C_0 using that key.

* * *

CSE509 – Computer Security – 3/02/06 Lecture

(Going back to authentication)

Recall three basic types of authentication:

1) Something you know (already covered: passwords)
2) Something you have
3) Something you are

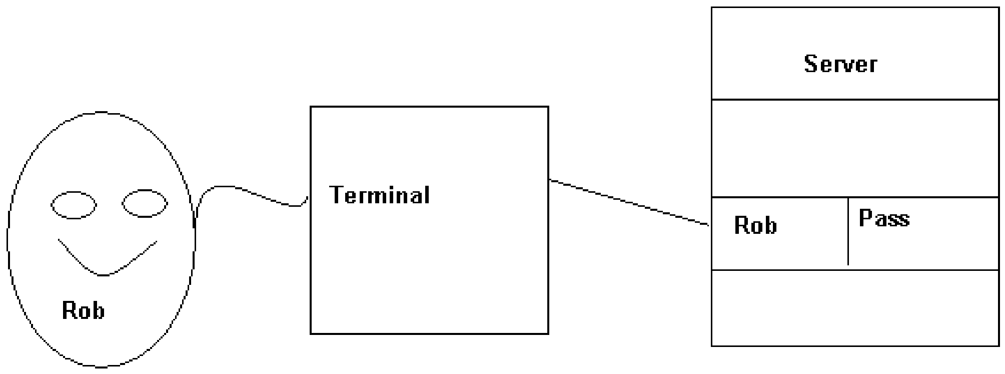
2 – Something you have

Here, this could be a key but in computer systems, we are usually talking about the following two things:

- Smart Cards
- Secure ID

Implementation of Secure ID

Secure Key:

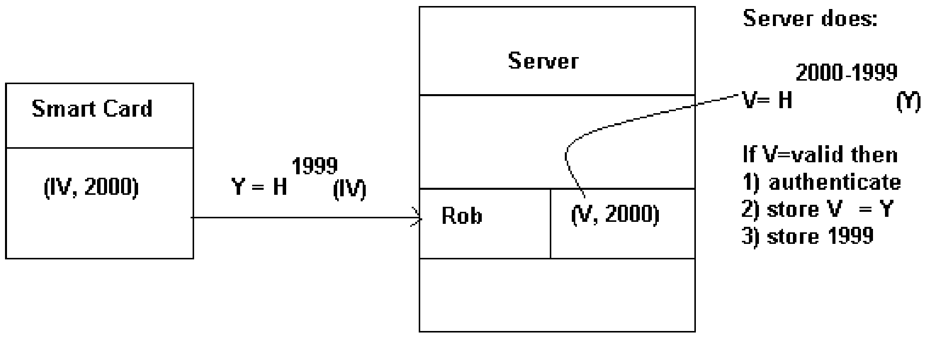


- It is clear that this scheme is not safe, but it can be made safer by using a challenge-response protocol
- Frequent password changes could also make this scheme safer, but they are a hassle for humans

Solution: Put pass on card – password is periodically changed on card, hence S/Key.

Method

1) Pick initial value (IV)
2) Compute $V = H^k(IV) = H(H(H(\ldots\ldots\ldots.H(IV)\ldots\ldots\ldots)))$
3) Initially server stores (V, K), e.g. (V, 2000)
4) Initially store IV, K on smart card, e.g. (V, 2000)
5) To login use card - on the first use: send $H^{1999}(IV)$
6) Card is automatically updated with new login parameters



Why is this secure?

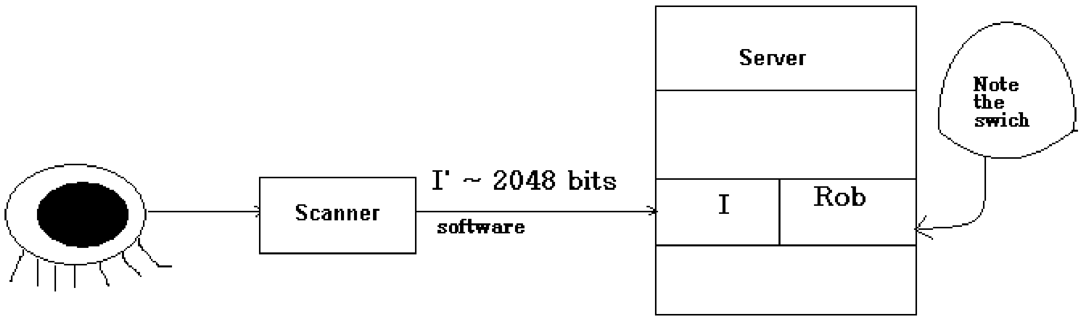
1) We use a different password every login session
2) Attacker cannot get next password because rehashing won't work (note that our login sessions use the hashes from outer to inner, rehashing would be going in the other direction)
3) In this scheme, we must strictly enforce in-order-logins, so 2000, 1998, 1999 would be rejected because $H^{1999}(V)$ must come from a third party (most likely malicious).

Why is this not secure?
It is vulnerable to a replay attack. If an attacker caches in one login attempts and then logs in using Y before the valid user's next attempt, the system has no way of knowing that this is a malicious user.

3- <u>Something you are (Biometrics)</u>

(Fingerprints, iris scans, voice print, face scans, palm scan, etc.)

Traditional Biometric Login



Valid if I' is close to I

Parameters
- If I' and I are form the same eye, then they should agree on about 1600 bits (about 20% difference)

- If I' and I are from two different eyes, then they agree on about 60% of the bits

- The problem that two different eyes agree on 80% is very small.

Problems

- Malicious scanner could store iris scan for later scan (note that a smart scanner could conduct a liveness test to make sure that the eye is a real one and not a picture)

- Usability depends on scanner – might need a higher acceptance threshold

- I stored in clear text on server

- Vulnerable to replay attack
- Iris is for life and supply is limited, so no frequent password changes
- Can’t do challenge-response

Fixing Biometrics

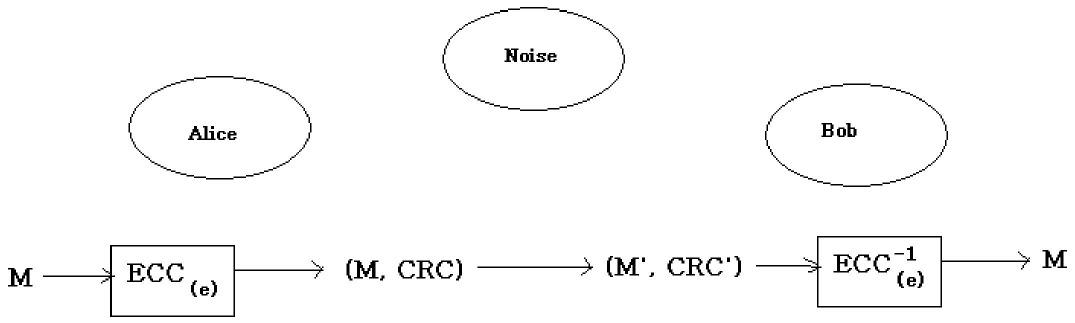
Step 1: Make biometric readings deterministic

$I = b_1$, ……………………………..b_{2048}

$I' = b_1'$……………………………..b_{2048}'

The Hemming distance $d_h(I, I') =< 400$

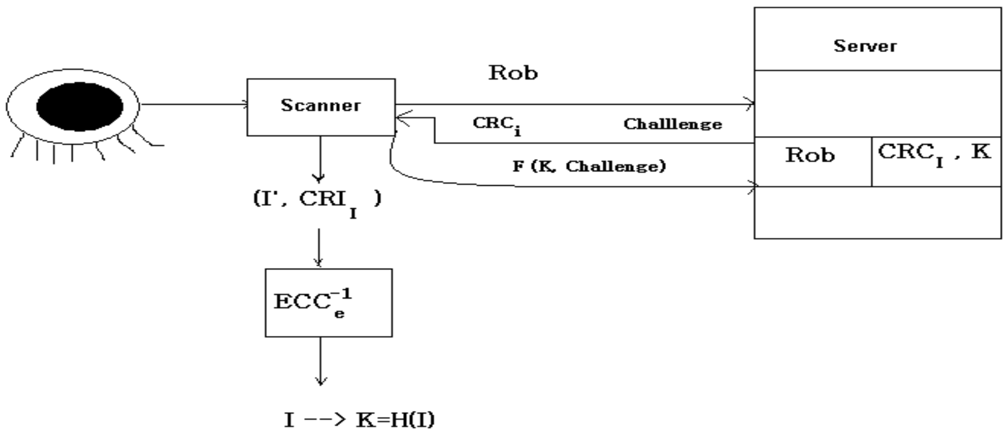
Error Correction



(CRC: Cyclic Redundancy Check)

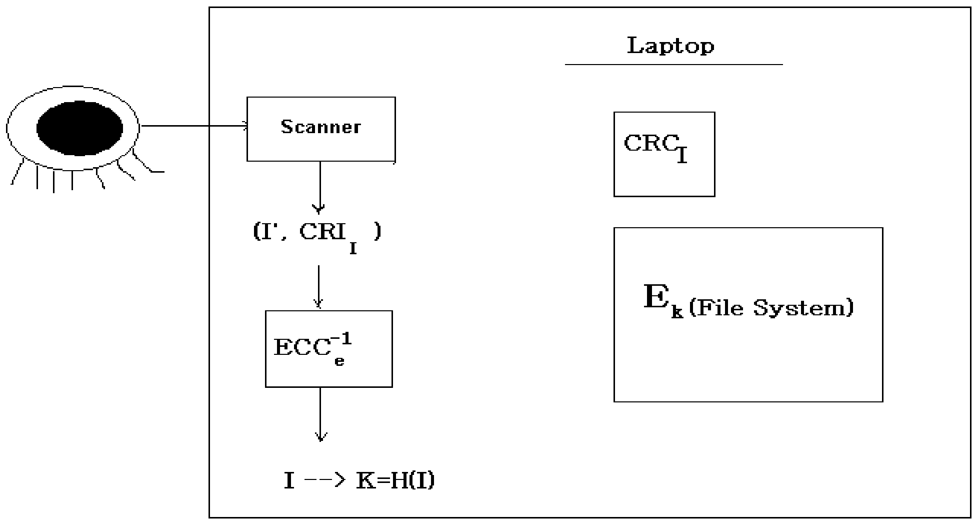
In picture above, Bob recovers M if $d_h(M||CRC, M'||CRC') =< e$. In other words, the ECC can correct up to e-bit errors.

ECC Trick



Here I is not stored on the server. The scanner computes I by running I' and the CRC for I through the inverse of the error correction scheme. The key (K) is obtained by hashing I. Finally, the scanner responds to the challenge with F(K, challenge).

Biometrics & Laptop (more secure laptops)
Sensitive information is sometimes stored on laptops, which can easily get stolen. Biometrics for laptop can protect the file system in a manner similar to the above scheme.



Now in order to get access to the file system, you must present a valid iris.

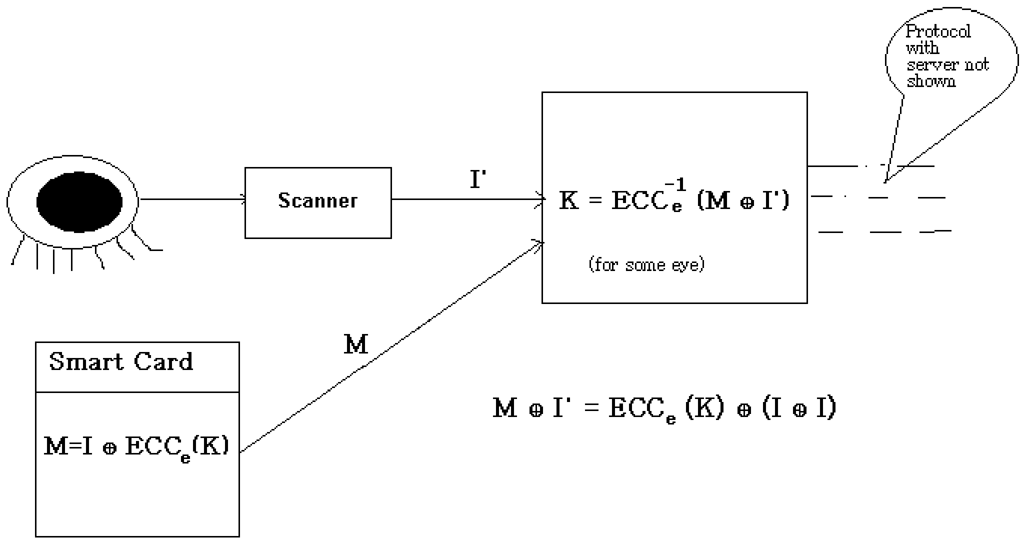
<u>Step 2</u>: Being able to change your key

- Don't: just add a key

  Note that simply using a key in addition to the iris is not enough because the scanner, in the schemes above, only needs the CRC and the iris to find the key. In order words, this does not protect from a stolen iris.

- Do: Ensure that user brings a secondary input

  (Something you have + Something you are)


Protocol with server not shown
Scanner
I'
K = ECC$_e^{-1}$ (M ⊕ I')
(for some eye)
M
Smart Card
M=I ⊕ ECC$_e$(K)
M ⊕ I' = ECC$_e$ (K) ⊕ (I ⊕ I)

CSE509 – Computer Security – 3/02/06 Lecture

(Going back to authentication)

Recall three basic types of authentication:
1) Something you know (already covered: passwords)
2) Something you have
3) Something you are

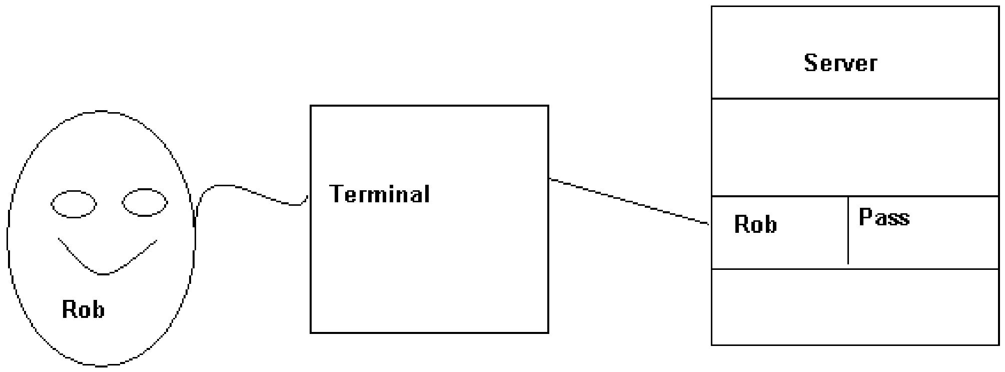
2 – <u>Something you have</u>

Here, this could be a key but in computer systems, we are usually talking about the following two things:

- Smart Cards
- Secure ID

Implementation of Secure ID

<u>Secure Key:</u>

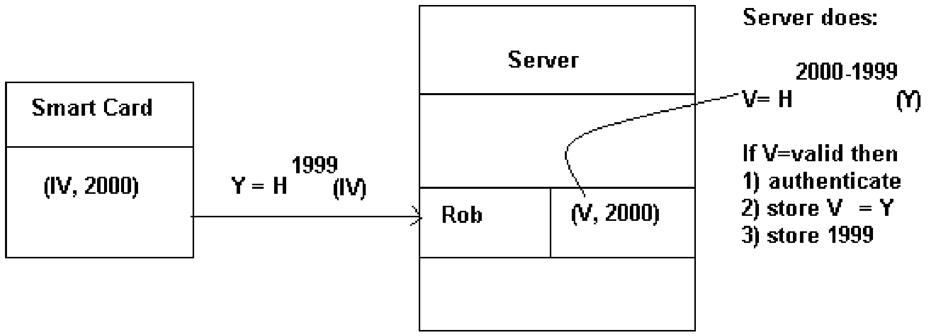


- It is clear that this scheme is not safe, but it can be made safer by using a challenge-response protocol
- Frequent password changes could also make this scheme safer, but they are a hassle for humans

Solution: Put pass on card – password is periodically changed on card, hence S/Key.

Method

1) Pick initial value (IV)
2) Compute V = H^k(IV) = H(H(H(………..H(IV)………..)))
3) Initially server stores (V, K), e.g. (V, 2000)
4) Initially store IV, K on smart card, e.g. (V, 2000)
5) To login use card - on the first use: send H^{1999}(IV)
6) Card is automatically updated with new login parameters



Why is this secure?

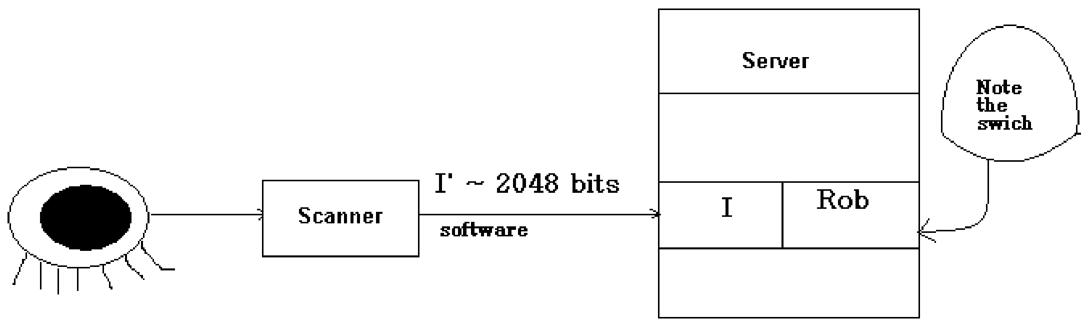
1) We use a different password every login session
2) Attacker cannot get next password because rehashing won't work (note that our login sessions use the hashes from outer to inner, rehashing would be going in the other direction)
3) In this scheme, we must strictly enforce in-order-logins, so 2000, 1998, 1999 would be rejected because H^{1999}(V) must come from a third party (most likely malicious).

Why is this not secure?
It is vulnerable to a replay attack. If an attacker caches in one login attempts and then logs in using Y before the valid user's next attempt, the system has no way of knowing that this is a malicious user.

3- **Something you are (Biometrics)**

(Fingerprints, iris scans, voice print, face scans, palm scan, etc.)

Traditional Biometric Login



Valid if I' is close to I

Parameters
- If I' and I are form the same eye, then they should agree on about 1600 bits (about 20% difference)

- If I' and I are from two different eyes, then they agree on about 60% of the bits

- The problem that two different eyes agree on 80% is very small.

Problems
- Malicious scanner could store iris scan for later scan (note that a smart scanner could conduct a liveness test to make sure that the eye is a real one and not a picture)

- Usability depends on scanner – might need a higher acceptance threshold

- I stored in clear text on server

- Vulnerable to replay attack
- Iris is for life and supply is limited, so no frequent password changes
- Can’t do challenge-response

Fixing Biometrics

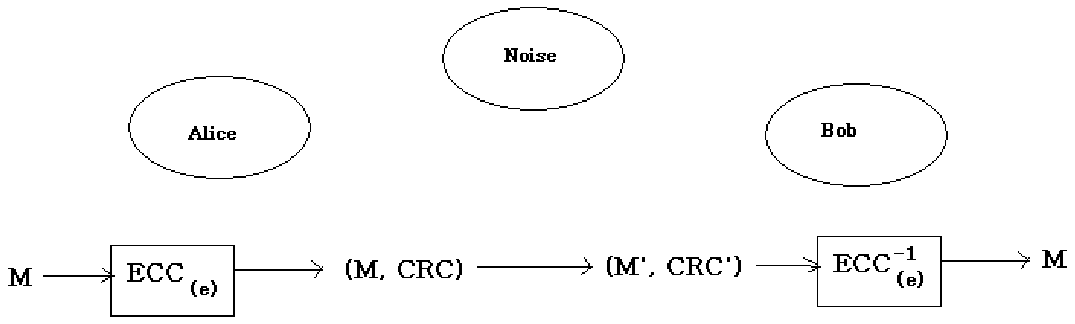
Step 1: Make biometric readings deterministic

$I = b_1, \ldots\ldots\ldots\ldots\ldots\ldots b_{2048}$

$I' = b_1' \ldots\ldots\ldots\ldots\ldots\ldots b_{2048}'$

The Hemming distance $d_h(I, I') =< 400$

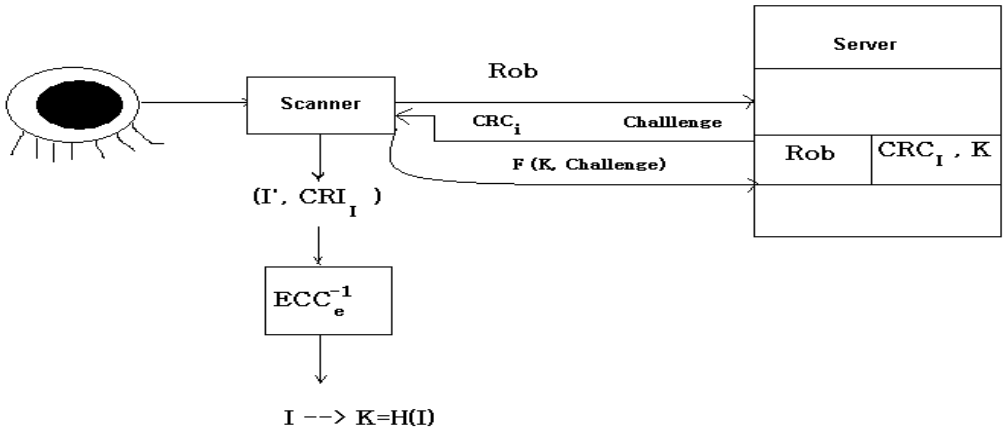
Error Correction



(CRC: Cyclic Redundancy Check)

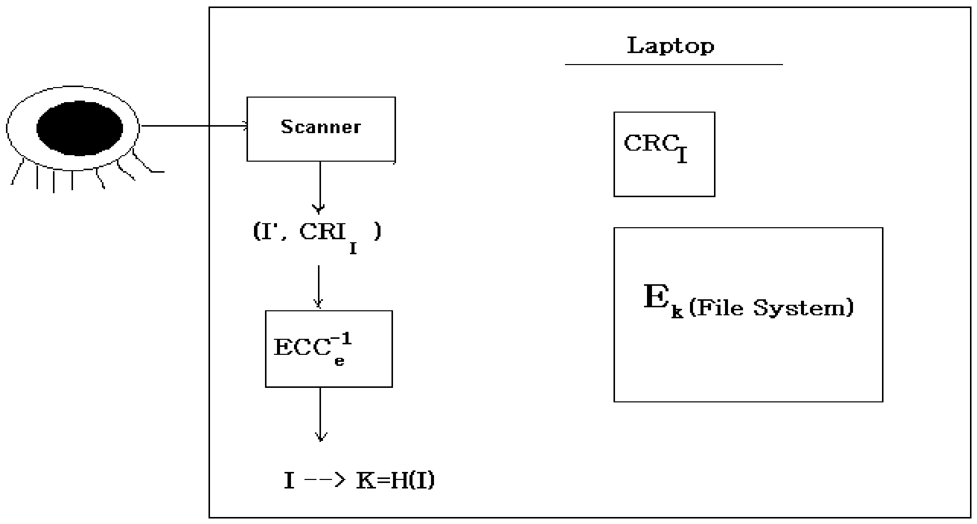
In picture above, Bob recovers M if $d_h(M||CRC, M'||CRC') =< e$. In other words, the ECC can correct up to e-bit errors.

ECC Trick



Here I is not stored on the server. The scanner computes I by running I' and the CRC for I through the inverse of the error correction scheme. The key (K) is obtained by hashing I. Finally, the scanner responds to the challenge with F(K, challenge).

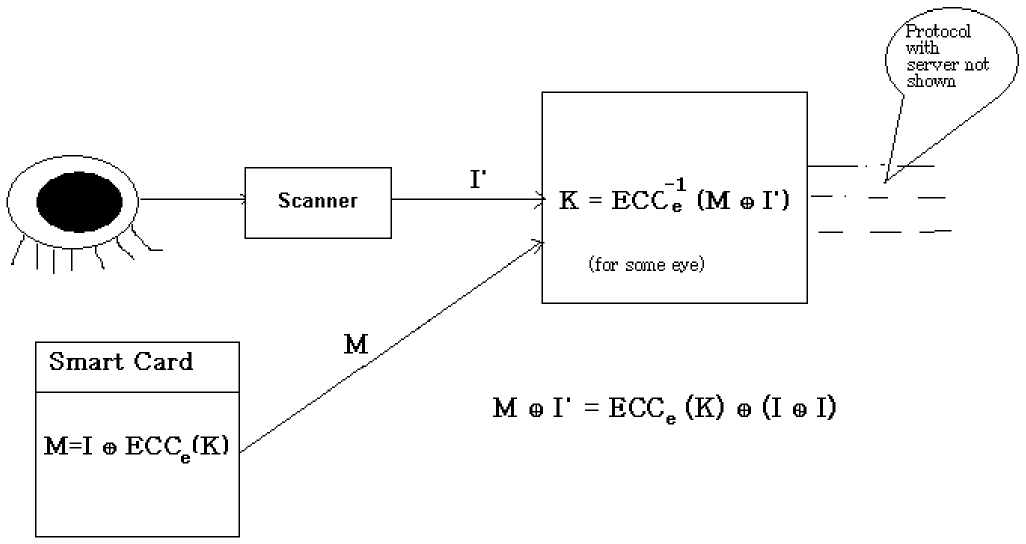
Biometrics & Laptop (more secure laptops)
Sensitive information is sometimes stored on laptops, which can easily get stolen. Biometrics for laptop can protect the file system in a manner similar to the above scheme.



Now in order to get access to the file system, you must present a valid iris.

Step 2: Being able to change your key
- Don't: just add a key
  Note that simply using a key in addition to the iris is not enough because the scanner, in the schemes above, only needs the CRC and the iris to find the key. In order words, this does not protect from a stolen iris.

- Do: Ensure that user brings a secondary input
  (Something you have + Something you are)



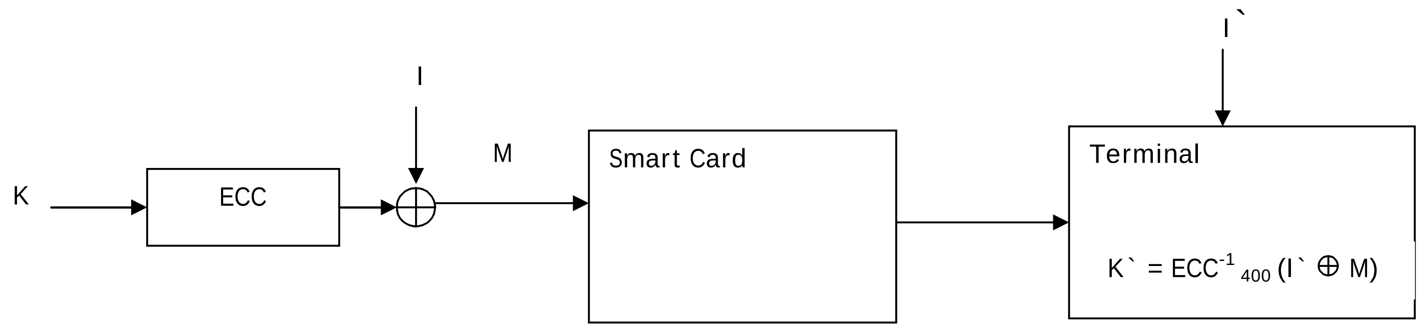
Protocol with server not shown
Scanner
I'
$K = ECC_e^{-1}(M \oplus I')$
(for some eye)
Smart Card
$M = I \oplus ECC_e(K)$
M
$M \oplus I' = ECC_e(K) \oplus (I \oplus I)$

<u>Review from last class</u> – Biometrics to generate keys

Biometric deficiencies

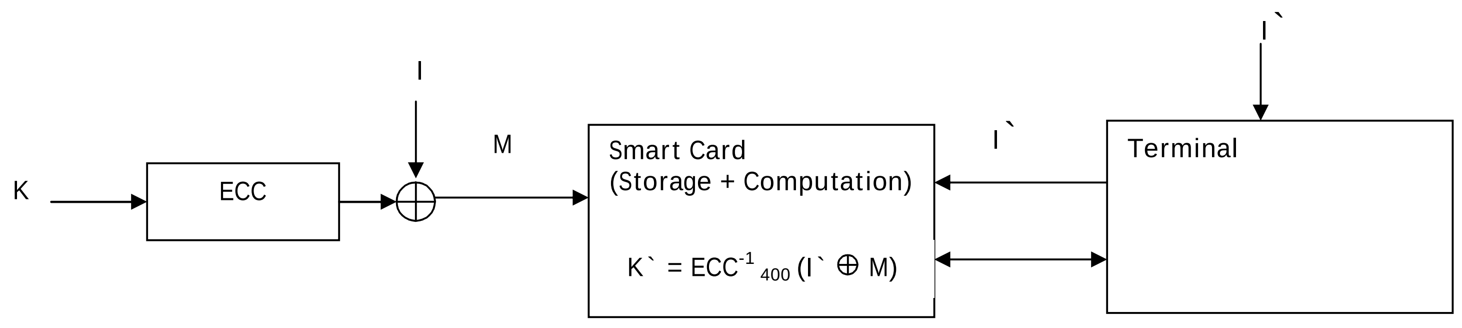
- One biometric for life
- Biometrics can be stolen e.g. by using photography
- Biometric reading are noisy (this can be fixed using ECC)

Naïve Scheme



This scheme cannot be used in public. Consider a scenario with a merchant and a customer who is making a payment using his smart card. Since the terminal knows K` the merchant can produce another smart card. The problem here is that the smart card is being used simply as storage.

Solution

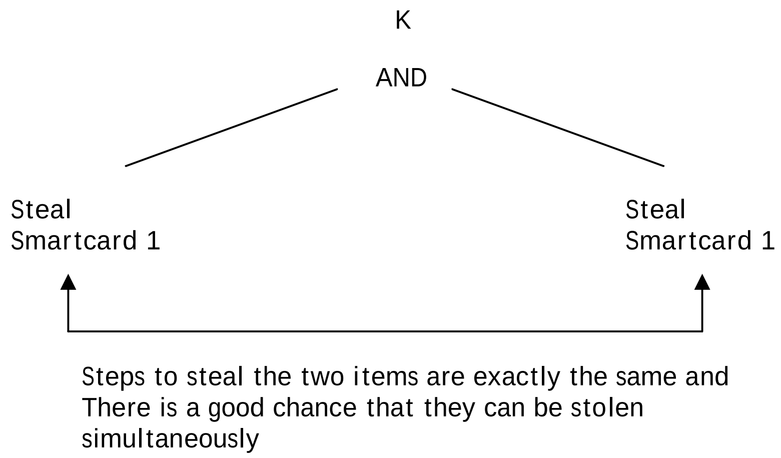


Deletes K` from memory after use

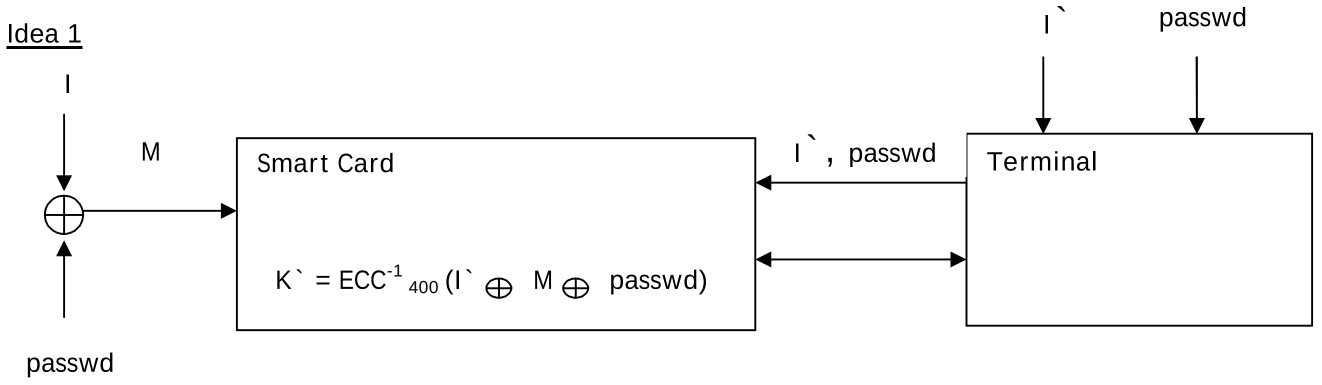
Idea is that nothing secret should ever leave the smart card.

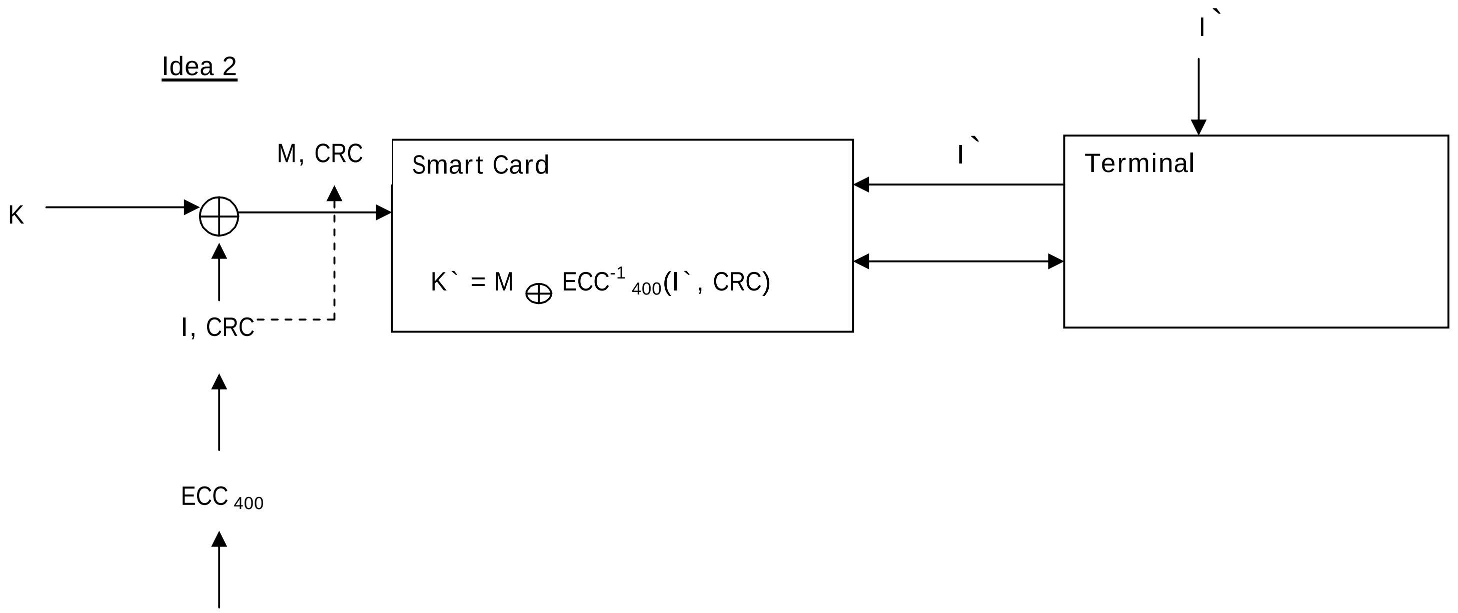
Why is smart card + biometrics better than just using two smart cards?
It's often as easy to steal two smart cards as it is to steal one, since the user will usually keep them together. Stealing an iris, though, requires a completely different attack from stealing a smartcard.

Attack Tree


K
AND
Steal Smartcard 1
Steal Smartcard 1
Steps to steal the two items are exactly the same and There is a good chance that they can be stolen simultaneously

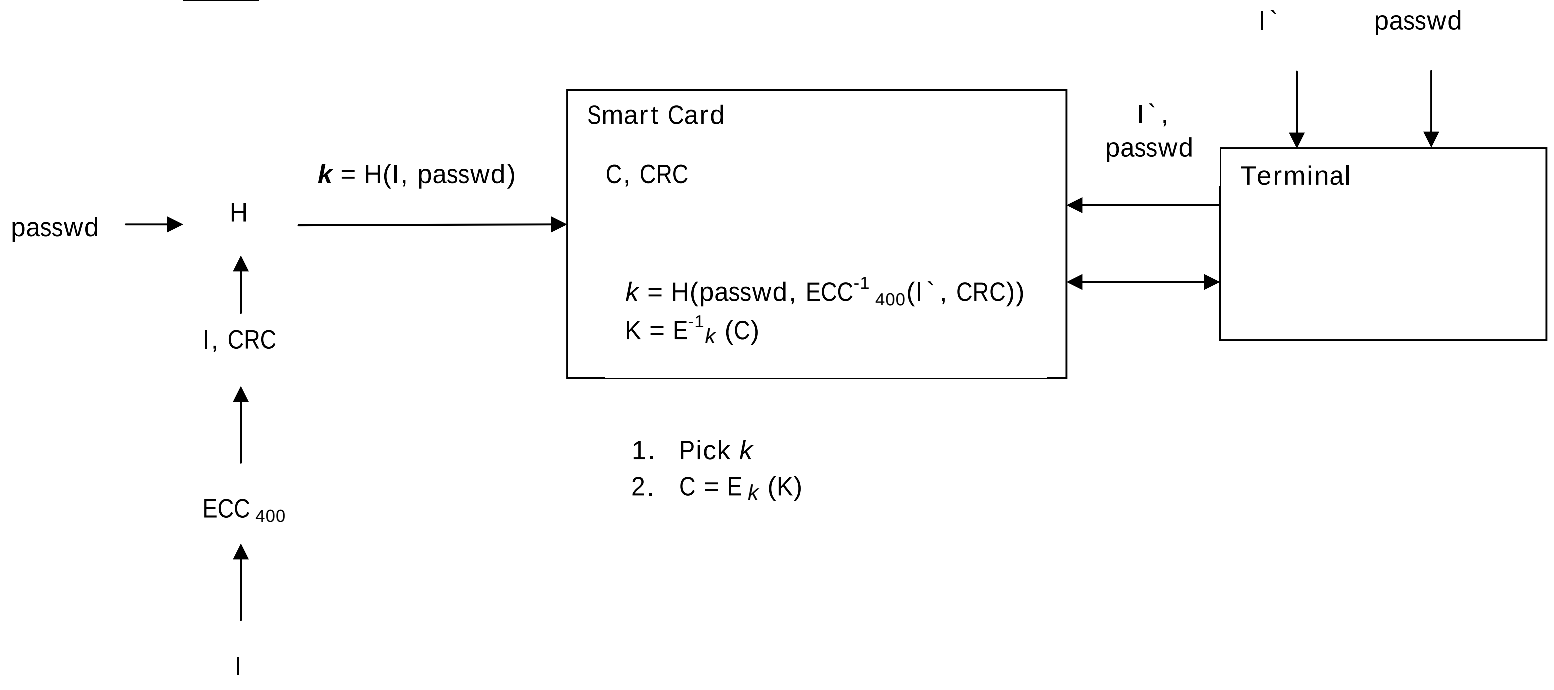
3 Factor Authentication – add something you know





Can we use the password now?

<u>Idea 3</u>



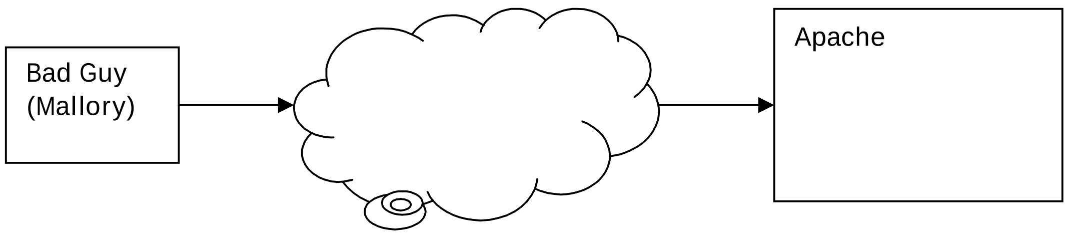
1. Pick k
2. C = E$_{k}$ (K)

<u>User Carries</u>
- Password
- Iris
- Smart Card [C, CRC]

Software Security

- Buffer Overflows
- Format string bugs
- Race conditions
- SQL Injection bugs
- Tractor beaming
- XSS bugs
- Chroot/chdir bug
- File descriptor attacks
- RPC implementation bug
- Bad Software design
- Usability flaws

Except for the last two all are mistakes made by the programmer.



"AAAA
AAAA
AAAA
....
....
....
AAAA"

- Apache ends up running attacker provided code
- Apache runs as root on unix

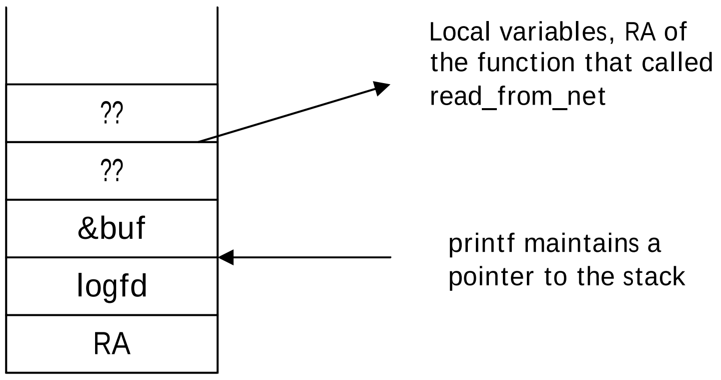
<u>Un-trusted Input Bugs</u>

- format_string.c

```
void read_from_net()
{
        char buf[1000];
        net_read(nuf, 1000);
        fprintf(logfd, buf);
}
```

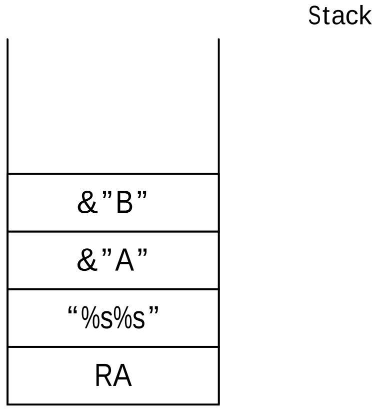
say buf = "%s%s%s ...."

Stack


??
??
&buf
logfd
RA
Local variables, RA of the function that called read_from_net
printf maintains a pointer to the stack

How printf works?

Printf(“%s%s”, “A”, “B”);



Problem!

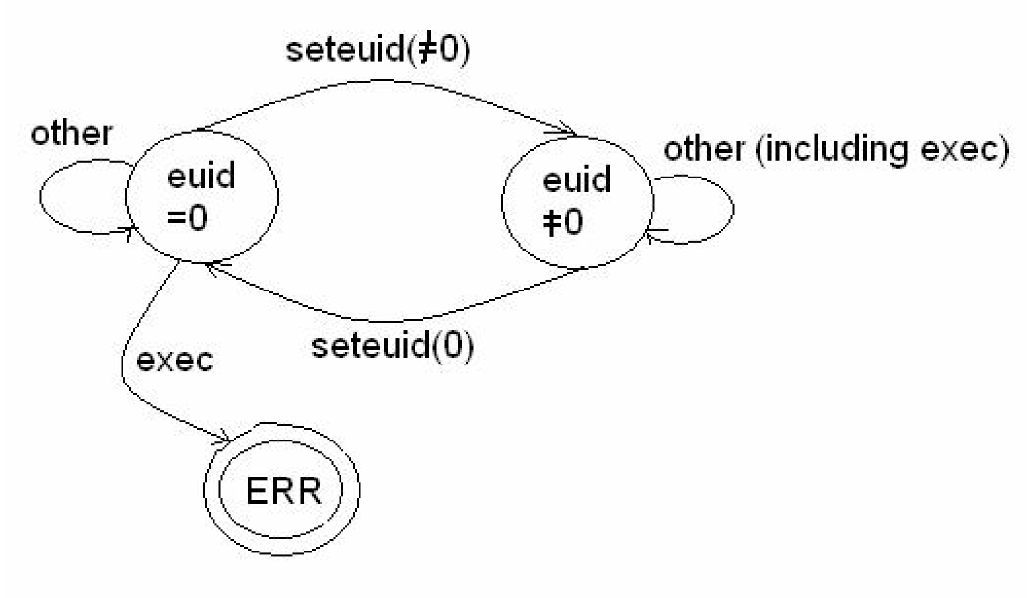
Printf(“Hello%n”, &bytes);
Writes number of bytes written so far to *&bytes

```
Void login_user()
{
        int authenticated = 0;
        char uname[100];
        getusername();


}

void getusername(char* uname)
{
       net_read(uname);
       fprintf(logfd, uname);
}
```

CSE 509 : System Security, Spring 2006.
Lecture Notes for 03/14/2006.

setuid/exec bugs:

Unix (and its flavours) provide a system call seteuid() to change a user's current effective user id.
- seteuid(0) : effective user id to become 0. (So, become root temporarily)
- seteuid(getruid()) : go back / drop root privilege temporarily.



Why would you drop root privilege ?
- If a trusted program wants to run an untrusted helper program.
e.g. emacs launched from cron.

A problem, could be if a user forgot to drop the root privilege.
Consider the following piece of code :
```
void runhelper()
{
        if((cmd= malloc(...))!=NULL)
                seteuid(getruid());
        exec(...);
        seteuid(0);
}
```

Now if Malloc fails, then exec is done as root.

(For getting dirty with setuid, see the paper : Setuid Demystified:
http://www.cs.berkeley.edu/~daw/papers/setuid-usenix02.pdf)

chdir/chroot:

Consider the following piece of code :

```
void switch_to_new_root(...)
{
        chroot("/var/anonftp");
        printf("Switched to new root !!");
}
```

Once the chroot is done, processes can only access files under /var/anonftp
The problem is that chroot doesn't change the CWD (current working directory) of the FTP server.

Inside OS, there is

```
  root          CWD
   /        /usr/share
```

Now, open("pics/stuff"); starts from CWD.

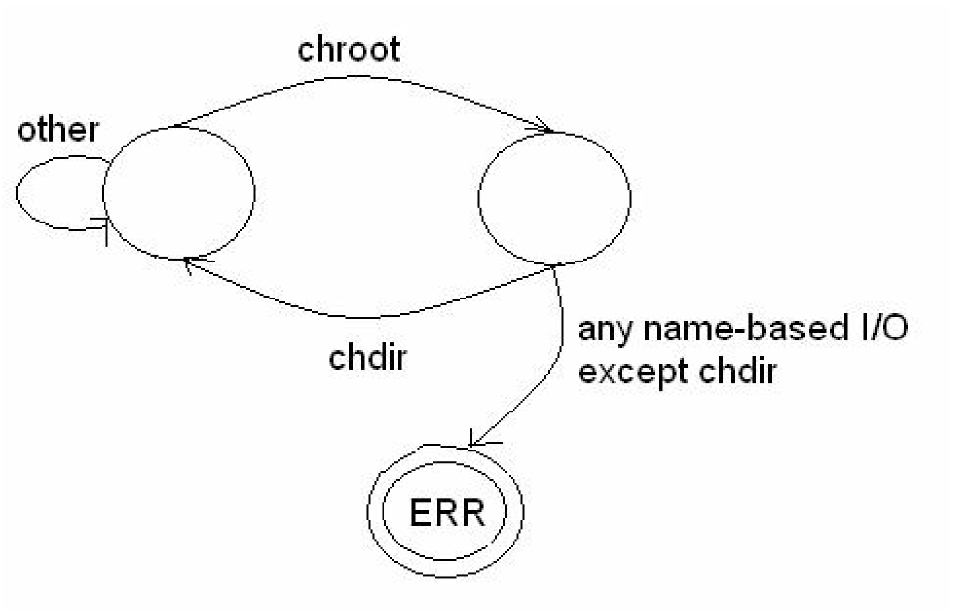
If we have issued chroot("/var/anonftp"),
it looks like this :

```
    root          CWD
  /var/anonftp    /usr/share
```

Now, another open("pics/stuff"); starts from CWD and still works !!

So, need to add a chdir("/"); after doing the chroot().
i.e. the code should look like :

```
void switch_to_new_root(...)
{
        chroot("/var/anonftp");
        chdir("/");
        printf("Switched to new root !!");
}
```



Tractor-Beaming Attack (Funky !)

Consider the following piece of code from a ftpserver:

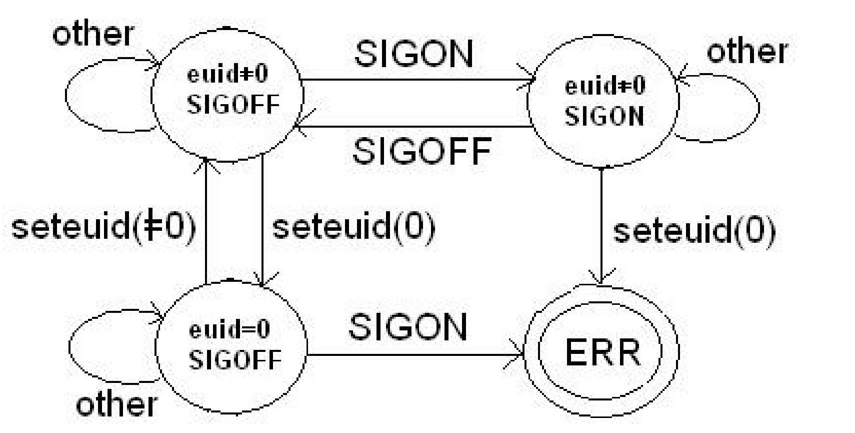
```
jmpbuf jb;
try_transfer(...)
{
        setjmp(&jb);
        do_transfer(...);
label1:
        …………..
        …………..
}

do_transfer(...)
{
label2:
        seteuid(0);
        // do some I/O with TCP
        seteuid(getruid());
}

sig_urg_handler()
{
        longjmp(&jb);
}
```

In the above code, we use setjmp - longjmp. TCP can send an urgent out of band message, and the handler for the urgent signal (sig_urg_handler()) does a long jump to label1 in try_transfer(). What this means is that, the ftpserver is now root, since we did the seteuid(0) but did not drop the root privileges before doing the long jump.

There are several ways to fix this problem, one of which is to ignore the urgent message signal from TCP as shown below:

```
do_transfer(...)
{
label2:
        signal(SIG_URG,IGNORE);
        seteuid(0);
        // do some I/O with TCP
        seteuid(getruid());
}
```

But, our goal is not to enable programmers to do the right thing, but to disable them from doing wrong.
Lets propose a rule, when euid=0, all signals must be ignored.



For 32 signals, 2^32 states in a state machine, +1 for euid, so analyzing all signals simultaneously requires 2^33 states, but we can analyze each signal independently efficiently.

File Descriptor Attack:

For a setuid program, user controls initial state of file descriptors. You always have 3 file descriptors :
STDIN: 0
STDOUT: 1
STDERR: 2

printf("HELLO");   // hard-coded write using fd 1 (stdout).

Now, if I ran the following code with only STDIN open, then fd = 1.

```
fd=open("some-critical-file");
printf("opened critical file\n");
```

So, setuid programs should do close(0); close(1); close(2); and then do open("/dev/tty"); open("/dev/tty"); open("/dev/tty");

As we see, all the attacks above had some state info globally.

Unchecked input:

Consider the following piece of code:

```
int len;
read(networkfd, &len, sizeof(len));
buf=malloc(len);
```

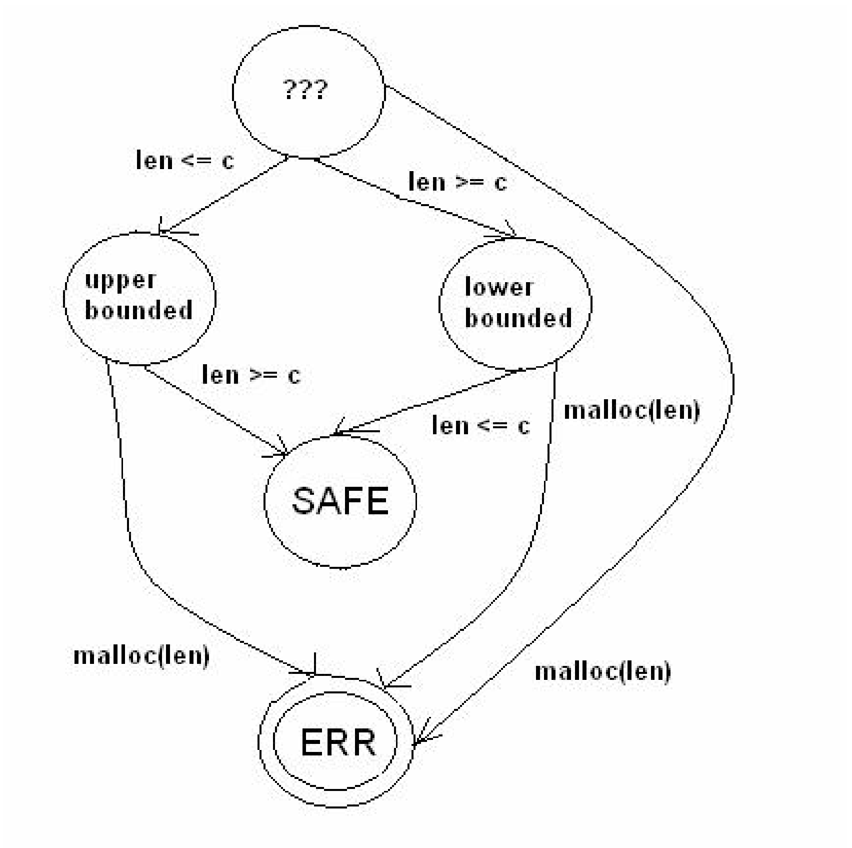
There should be a check how much read returns. (Since, read could read an amount less than that requested.)
something like:

```
if(read(...)==sizeof(len))
```

The value of len here is untrusted (comes from an external source.)
There should be a bound check on len.
something like:

```
if(read(...)==sizeof(len))
        if(len<100 && len>=0)
                buf=malloc(len);
```

So, len has state as shown by the state machine below.



This is not a very conservative approach. The state machine doesn't verify if it is bounded between the values that we need it to be bounded between.

Access / Open :
----
setuid-root program wants to open "file" if and only if invoker could open "file".

if we had something like : open("file") and the file came from an untrusted source, then the victim could possibly open anything (any malicious file).
So, there is an access/open mechanism.

```
if(access("file"))
        open("file");
```

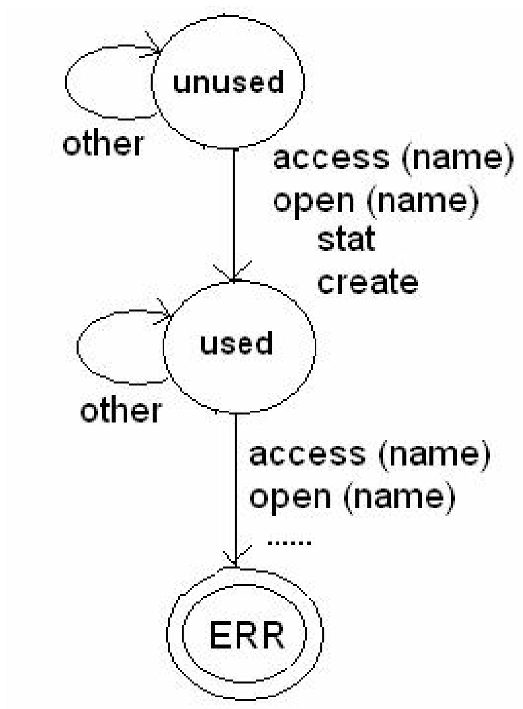
An attack on this, is if file is as follows:
"file" is owned by attacker, invokes setuid victim program. Then victim runs.
Executes access("file"), gets an "ok' from the OS, it is then scheduled out by the OS.
The attacker's code now gets control and executes rm("file") and creates a softlink to the /etc/shadow file. then it is scheduled out by the OS.
The victim code then executes the open("file") which actually ends up opening the /etc/shadow file.

Just showing the same thing, nicely :

```
    Victim runs                        Attacker runs
    -------------                      ---------------
                                       file is owned by the attacker
                                       invokes setuid victim program
access("file")
gets "OK" from the OS
                                       rm "file"
                                       ln -s /etc/shadow file
open("file")
```

So, there is no way to atomically do the access and open.
The real issue is that there are two uses on the name "file" in two different system calls between which the OS can schedule other processes/tasks.

To prevent this, we could wrap around them into a transaction.
But this enables programmers to do the right, doesn't disable them from doing the wrong.



Here are 2 interesting papers:

"Fixing Races for Fun and Profit: How to abuse atime" -
http://www.cs.berkeley.edu/~daw/papers/races-usenix05.pdf

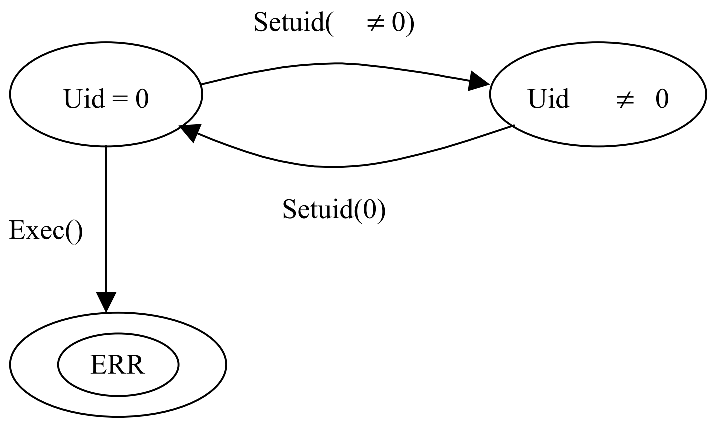
"Fixing Races for Fun and Profit: How to use access(2)" -
http://www.csl.sri.com/users/ddean/papers/usenix04.pdf

CSE509 Spring 2006. System Security

Lecture Notes for 03/16/2006

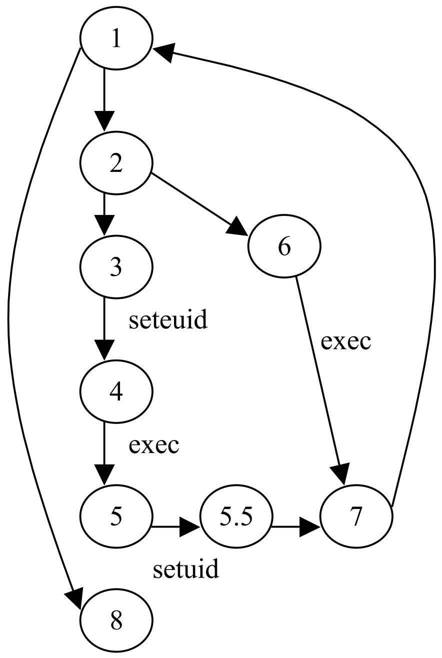
- **Model Checking**

L$_p$:



L$_{CFG}$:

```
     Void func()
     {
1.
          while (……) {
2.
               if   (……) {
3.
                    seteuid(geteuid());
4.
                    exec(…);
5.
                    setuid(0);
5.5
               } else {
6.
                    exec(…);
               }
7.
          }
8.
     }
```



Assume: Code starts with euid = 0.

CFG(Control Flow Graph)

CFG is a Finite State Machine.
Giving regular language L_{CFG}, security property is specified as a FSM, get L_P.
What we want is $L_P \bigcap L_{CFG} = \phi$.
$L_P \bigcap L_{CFG}$ is regular!

Four steps to verify:

1. Specify security property as a FSA.
2. Convert input program into a FSA(PDA, PushDown Automata).
3. Compute product of machines from 1&2.
4. Check if product machine gives empty language.

Tools:

MOPS
SLAM
BLAST
MECA

Choices in Model Checking design:

1. **Function Vs. whole program**
   MOPS, SLAM and BLAST are whole program checking. MECA is using function by function analysis.

2. **Soundness Vs. convenience**

   Define: A program analysis tool is ***sound*** if whenever it says a program doesn't have bugs, then the program doesn't have bugs

   Define: A ***false positive*** is when an analysis tool claims that a program has a bug, but it doesn't

   Define: Completeness – A ***complete*** analysis tool has no false positives.

   Theorem: Choose 2 from ***Soundness, Completeness*** **and** ***Termination.***

   MOPS: sound, terminates
   SLAM and BLAST: sound, complete
   MECA: terminates, convenience

3. **Theorem Proving**
   MOPS has none, SLAM/BLAST have lots of, and MECA has a little.

Unsound: may not be security after using, but still helpful.

- Lasebeam Vs. shotgun approach
- Asymmetry of attacking Vs. defending

- Defender must close all vulnerabilities
- Attacker only need find one weakness

➢ Combined methods:

- Surely to be a bug
- Likely to be a bug
- How about the percentage of possibility to be a bug

MECA Results:

➢ 1000's bugs

➢ Low false positive rate: 10%~100%

➢ Commercial coverage

➢ Misses real bugs

Open:

Design-level bugs.

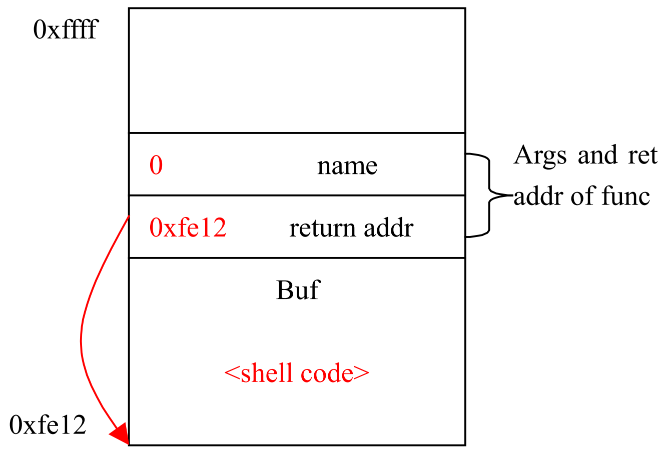
Summary for the four analysis tools:

	Soundness	Completeness	Termination	Analysis approach	Theorem Proving
MOPS	Y	N	Y	Whole program	N
SLAM	Y	Y	N	Whole program	Many
BLAST	Y	Y	N	Whole program	Many
MECA	N	N	Y	Function by function	A little

● Buffer Overflow

Example 1:

```
Void func(char *name)
{
        char buf[1024];

        strcpy(buf, name);
        return;
}
```



Stack

The attacker makes

Name = "<shell code>\xfe12\0"

Name = "NOP NOP NOP <shell code>\xfe12\0"

Q1: How the attacker can know the return address need to be filled?
A: For example, the attacker find the victim machine is using Linux. Then the attacker need install exactly the same version of Linux on his local machine, download the same open source code. Run it, the attacker will be able to find the return address.

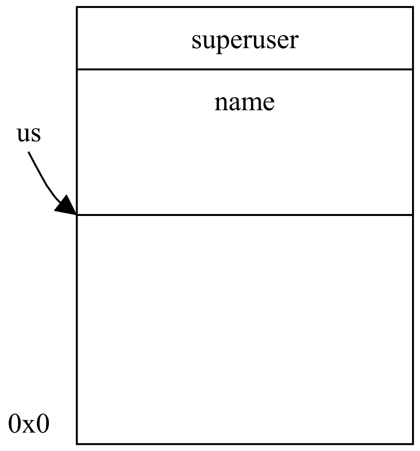
Q2: How to execute the shell code inside the buffer?
A: The attacker can use a decoder to eliminate forbidden bytes.

<u>How to fix?</u>

Use NX bit, forbid execution on stack.

<u>Example 2:</u>

```
Struct usestate {
        Char name[64];
        Int superuser;
}
Void login_user(char *name)
{
        struct usestate *us = malloc(..);

        strcpy(us->name, name);
        return;
}
```

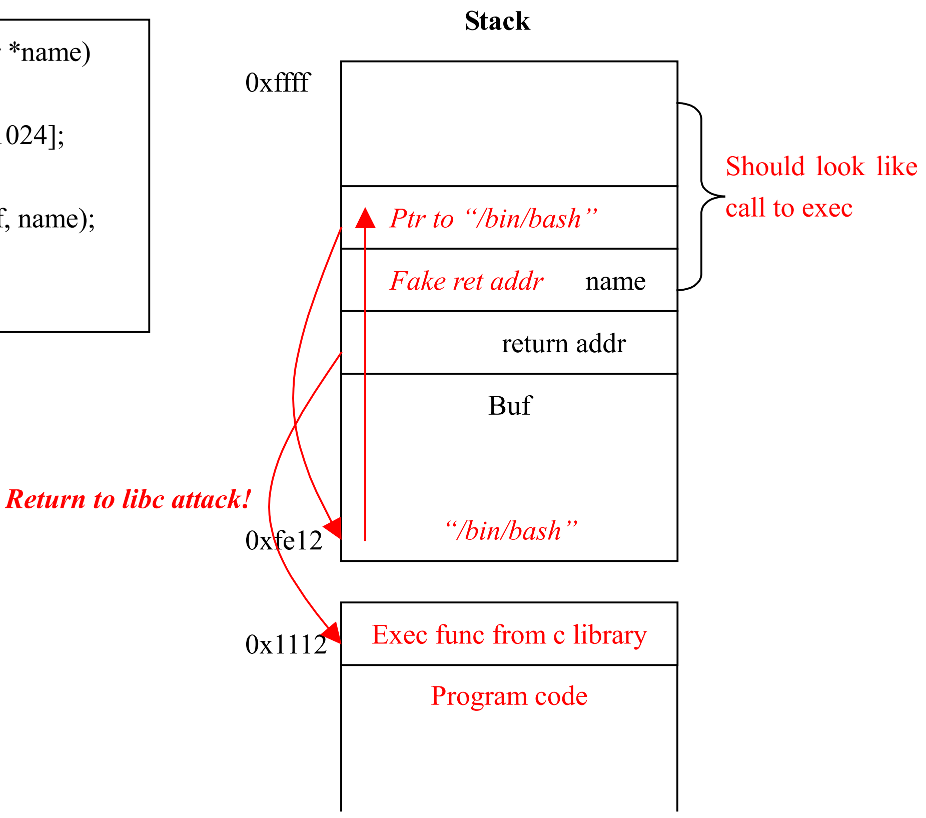


Heap

Note: the 'superuser' field may be overwrited if the argument 'name' is long enough.

<u>Example 3:</u> return to libc attack.

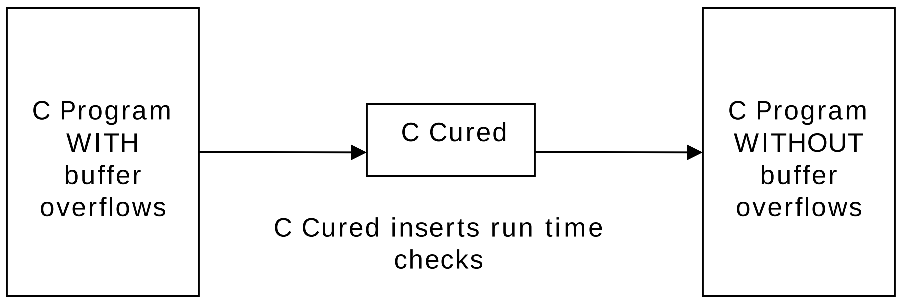
```
Void func(char *name)
{
        char buf[1024];

        strcpy(buf, name);
        return;
}
```



CSE 509 class on 03/21/2006

Reading assignment was reading paper on C – Cured

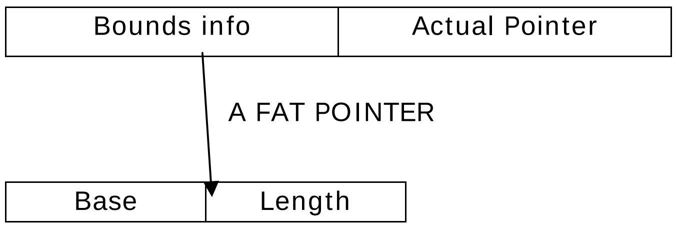
C Cured


C Program WITH buffer overflows
C Cured
C Program WITHOUT buffer overflows
C Cured inserts run time checks

Overview: Some common mistakes that programmers in C make

1. Accessing / referencing outside of an array

Remedy: Bound check the pointer before each dereferencing operation.
How to do this? : FAT POINTER



In this case, the bounds info is the base address of array and length = ‘number of bytes of array’

2. malloc / free bugs
   e.g. freeing the same location may corrupt the heap
   Remedy:
   a. Don’t use free() at all
   b. Use garbage collection instead of free. (Caution: There’s a probable security hole since garbage collector runs lazily and attacker may use this temporary extra space to inject his code. But considering its advantages over its disadvantages, garbage collection seems to be a good move.)

3. Casts and Unions
   - Use runtime checks with casts.
   - Disallow unions

   e.g. of unsafe use of unions

```
union u
  { int a;
    char * ptr;
  };
```

In this e.g., one can initialize the union with an integer and then dreference it assuming it to be a char * pointer.

e.g. of unsafe operation using cast

```
char  * p = (char *) 5;
```

in this case p should not be dereferenceable.

How Fat Pointer helps ?
Casting int to pointer yields an undreferable Fat Pointer.

0 (base)	0 (length)	5 (value)

After type casting '(char *) 5', it will be converted into a Fat Pointer as stated above. This cannot be dereferenced.
RULE: A fat pointer with base = 0, length = 0 cannot be dreferenced.
Thus, C-Cure alters the C structures such that they are still usable as in C but are safer than in C.

Breaking up C in subsets with different levels of associated security:

Language SAFE:
Define SAFE to be subset of C with
- No pointer arithmetic
- No casts
- No union

Therefore, we get rid of security hazards due to
- Bounds checking
- Type checking
- No necessity to keep any kind of bounds information

e.g.

```
int linked_list_search(list * i, int t)
{
        While(i NOT EQUAL NULL  && i->data NOT EQUAL t)
                i = i -> next;
        if(i)
                return true;
        return false;
}
```

Language SEQ = SAFE + Pointer Arithmetic:

e.g.

```
int sum(int * vec, int len)
{
        int acc = 0;
        int  i;
        for(i = 0 ; i < len ; i++)
                acc += vec[i];
        return acc;
}
```

Define SEQ to be subset of C with

- bounds check
- bounds information
- we DON'T need type checks

Language C / Dynamic / Wild – Normal C language that we use

The idea is to find the part of the source code which falls in 1 of the 3 different languages and then process parts falling under languages as above stated 1,2,3 differently using secure.

1. C – Cured partitions user programs into
   -SAFE
   -SEQ
   -WILD parts
2. Compiles each of these 3 parts separately
3. Handles the conversions at the boundaries of transition from one type of language to another.

-Partitions are performed using type qualifier inference.
-Analysis is sound (def of sound : Analysis assures on errors due to casting)
-Code should be free of buffer overflows.

Performance:

Overhead – 1.5 x slowdown

Statistically, a source program has:
SAFE pointers – 90 %
SEQ   pointers – 9 %
WILD pointers – 1 %

BETTER THAN OTHER SIMILAR TOOLS – (e.g. purify slows 50 X times)

Binary compatibility issues:
CCured is not compatible with other libraries which are not compiled using CCured.
e.g. Non CCured compiled libraries may not have Fat pointers

- Need to convert pointers at interfaces between CCured and NonCCured code.

Proposed advancements:
For the interfaces between CCured and NonCCured code :
- Writing wrapper functions to unFatten the pointers
- Writing wrapper functions to Fatten the pointers back

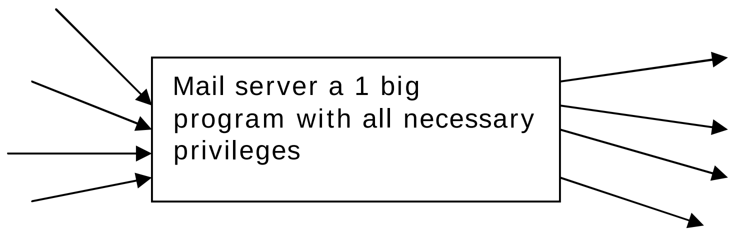
Privilege Separation

Reading assignment was paper – privTrans

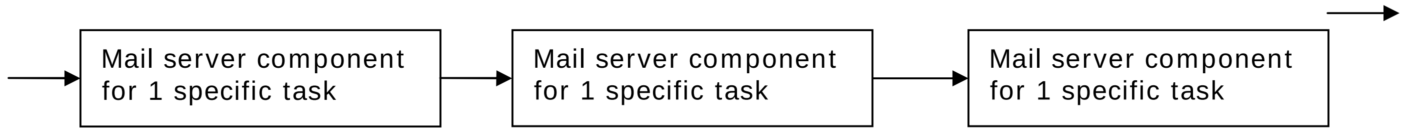
Principle of least Privilege :
Only assign MINIMUM privilege necessary to do a certain task to the user

A program needs privilege for 3 reasons:
- Access secret data (e.g. read a file)
- perform special operation
- change identity

Ways to do privilege separation:
Divide a big program into smaller programs with different privilege for each.



BAD DESIGN


Mail server component for 1 specific task
Mail server component for 1 specific task
Mail server component for 1 specific task

GOOD DESIGN

--- x x x ---
(END OF DOCUMENT)

Privilege Seperation (contd..)

March 23, 2006 mnair@cs.sunysb.edu

Depending on context almost any error can impact security. If a program does not have the proper return value, for example it might not be attackable on it own but in the context where it is called it might be attackable.

Question 7

Discussion of the solution to the mid-term review question 7. *Design a system of privileges and an Access Control Matrix to launch a missile iff at least 3 people agree to launch it and no person objects to the launch.* Let the design of the ACM (Access Control Matrix) be as shown in table 1. Let us just have just one object *Missile*. The domains consists of have 4 users and a Launcher. The *Launcher* domain will be the book-keeping and will record the current state of voting. Initially each of the user domains have a *vote* privilege.

vote1 is used to cast the first vote. It removes the voting privileges of the voter and changes the state of the launcher domain to reflect the fact that a one-vote has been cast in favor of the launch. Similarly *vote2* and *vote3* are used to cast the second and third vote respectively.

```
vote1(d) {
if(vote $\in$ A[d,missile] && A[launcher,missile]=$\emptyset$ ) {
A[d,missile] $\setminus=$ {vote}
A[launcher,missile] $\cup=$ {one-vote}
```

	Missile
User1	vote
User2	vote
User3	vote
User4	vote
Launcher	

Table 1: Initial Condition of Access Control Matrix

```
}}

vote2(d) {
if(vote $\in$ A[d,missile] && A[launcher,missile]={one-vote} ) {
A[d,missile] \ = {vote}
A[launcher,missile] = {two-vote}
}}

vote3(d) {
if(vote $\in$ A[d,missile] && A[launcher,missile]={two-vote} ) {
A[d,missile] \ = {vote}
A[launcher,missile] = {three-vote}
}}
```

veto is used to veto the launch. *abstain* remove the voting rights of a user. It is not possible to get it back, though he may later veto the launch if required. *launch* check whether three votes have been cast in favor and whether everyone has voted then the state of the missile is changed to launch.

```
veto(d) {
if(true) {
A[Launcher,missile] = {veto}
}}

abstain(d) {
if(true) {
A[d,missile] \ = {vote}
}}

launch() {
if(A[Launcher,missile] = {three-vote} && A[User1,missile]=$\emptyset$ &&
A[User2,missile]=$\emptyset$ && A[User3,missile]=$\emptyset$ && A[User4,missile]=$\emptyset$
) {
A[Launcher,missile] = {launch}
}}
```

The above system achieves the following security goals:
· No one should be allowed to vote more than once.
· A person may vote for the launch and then veto it later.
· Once a veto is done it is not possible to retract it.
· Before a launch takes place everyone should have cast his vote or abstained from voting.

Question 2

Discussion of the solution of mid-semester review question 2. *How can a virtual machine monitor set up the page tables for two virtual machines so that they share a page of memory?* The applications could share the physical pages so the virtual machine monitor (VMM) sets up the page translation for the two process, such that different virtual pages of each processes both point to the same physical page in memory. Thus changes made by one process on the page can be seen by the other process. Two approaches for sharing between mis-trusting applications are:

Using Locks : The server could take a write-lock on the page and this should prevent the client from being able to write on the page. The important consideration here is that the locking should be mandatory and not voluntary. Thus if the locking scheme needs to be effective the locks should be managed by the VMM. And VMM should make sure that it is not possible for any process to write on a page on which write-lock is present.

Copying Data : A simple solution to the problem of sharing is that every process before it starts working on the data on a shared page, it first copies it to a private page. This ensures that the data is never in an in-consistent state and after this, all future processing should be done on the data in the private area.

Privilege Separation

Privilege is needed for the following:
1. Accessing secret data.
2. Performing special operation.
3. Changing identity.

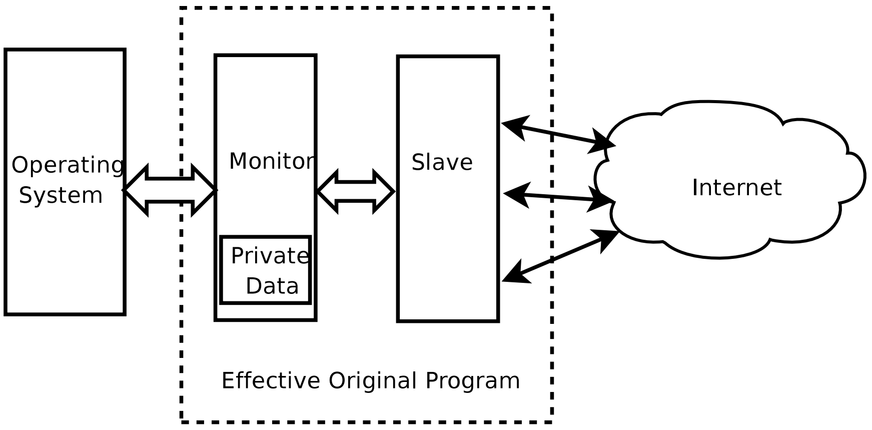


Figure 1: Privilege Separation

The main idea behind privilege separation is encapsulated within the figure 1. The slave interacts with all the external agents. To perform tasks of a privileged nature the slave depends on the monitor. Thus all interactions of the slave to the underlying Operating System(OS) is through the the monitor. Apart from meditating between slave and OS, the monitor also stores all the private data, the program is heir to. The monitor run with complete privileges (root) and the slave runs with no privileges(nobody). The idea is that the original program should perform privileged operation only in certain states.

Figure 2 captures all the states in the authentication of a user in OpenBSD SSH Daemon (sshd). In the figure, the split at *preauth3* is due to the two forms of authentication that is available in sshd; by pass-phrase and by public-key. This model is from the work done by Niels Provos [2]. He constructed this model by studying the code of sshd. Using this model, he proceeds to do a manual privilege separation of sshd code.

Using Provos' work as a basis, we design a modified structure for privilege separation. This is shown in the figure 3. Here along with the actions performed by the monitor in the original design, the monitor also checks a model. This model shows all the possible states which the slave can be in and the actions might be performed in each states. Thus the monitor stores the state and also ensures that no deviation takes place from the model. This is required for privilege separation to be effective.

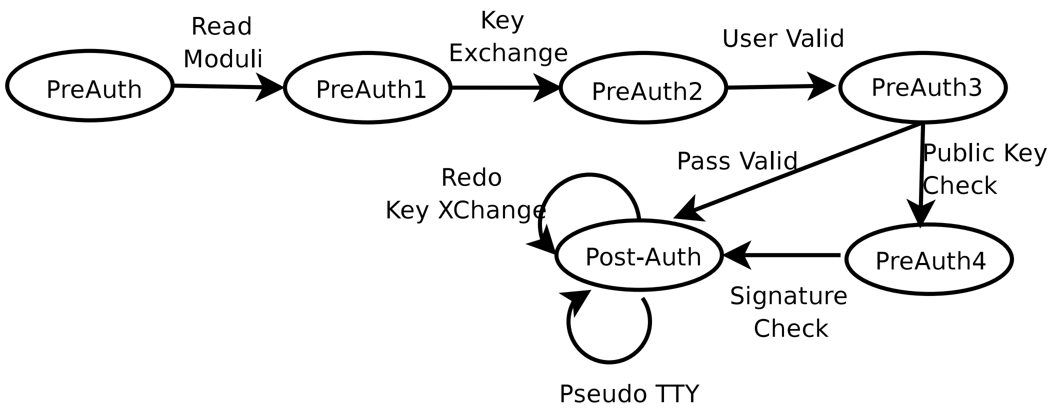


Figure 2: States of SSHD Authentication

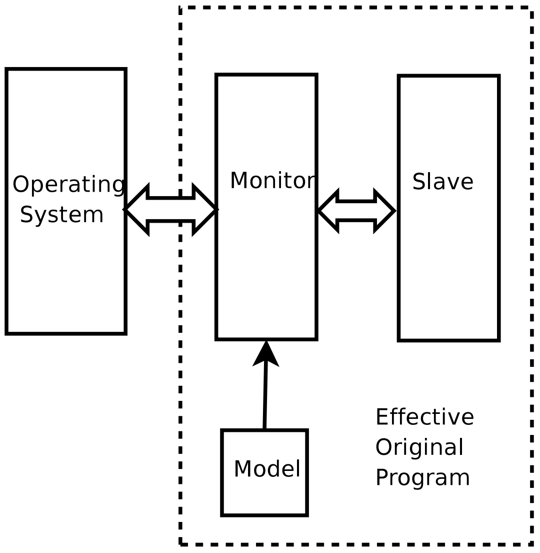


Figure 3: A new model for Privilege Separation

Automatic Privilege Separation

The automatic privilege separation paper Privtrans **??** proposes a solution based on static analysis. They require an annotation of which functions and variable about their privilege. From this by means of a type qualifier inference, all the privileged functions are determined. All the privileged functions are performed by the monitor. There exists a storage sub-system in the monitor and all communication between the monitor and the slave are done using RPC. The paper is very vague and remain in-conclusive how the model is generated.

Issues in Privilege Separation

Should the slave execute the monitor or should the monitor execute the slave? For purposes of security, the monitor should spawn and execute the slave. This way the monitor can have finer control over the actions of the slave. Also, monitor should only talk to the slave and no other application, this cannot be ensured if the slave execute the monitor functions, as a malicious slave could high-jack the monitor.

References

[1] *Privtrans: Automatically Partitioning Programs for Privilege Separation*, David Brumley and Dawn Song, Carnegie Mellon University.

[2] *Preventing Privilege Escalation*, Niels Provos, Markus Friedl and Peter Honeyman, 12th USENIX Security Symposium, Washington, DC, August 2003.

CSE509 Lecture Notes for 03/30/06

Pramod Adiddam

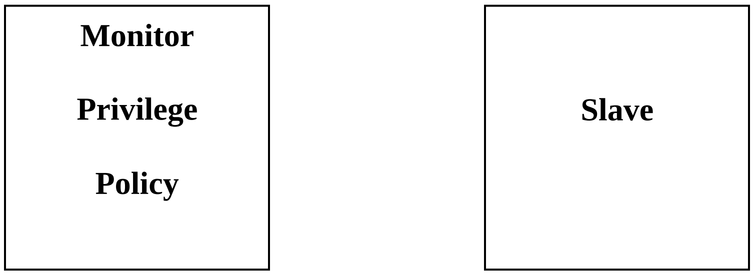
Privilege separation:

Policies

Privilege separation is to be secure. In privilege separation, there are two components -

Privilege program = Privilege + Policy

where, privilege and policy are usually bound together.



Hence, it is required for the policy to go into the monitor because the privilege would be in the monitor.
Also, an ideal privilege separation wouldn’t introduce new bugs. That is, new separated program should have a bug if and only if original program has a bug. Original program may have many bugs. These bugs will be carried intact to either the monitor or slave. Such a bug can land in slave or monitor. If it lands in the slave (and the privilege and policy are in the monitor) then it is OK to be in such a situation because now that bug can’t be exploited to gain privileges. If the bug lands in the monitor, then it is not clear whether it is easy or difficult to exploit it as it might be the case that now there is less code to find the bug or the bug can now be difficult to reach.

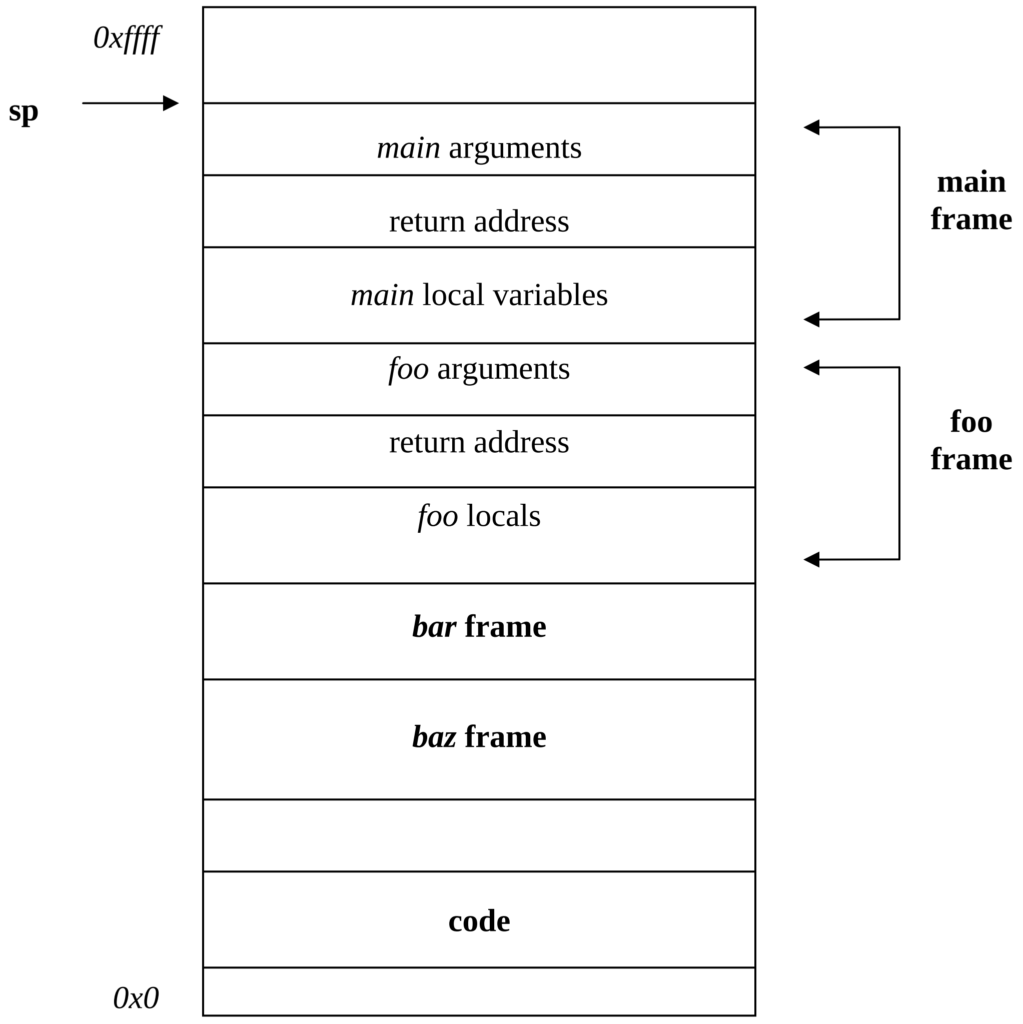
Buffer overflows:

Let us see where the attacker can be stopped by going through a list of what an attacker would do:

1. Find an overflow in source – (These kind of attacks are countered by tools like BOON)
2. Send overflowing message – (Tools like CCured)
3. Overwriting the return address – (by Address Space Randomization)
4. Program returns to Return Address
5. Execute shell code
6. Makes system call

Randomization

For a buffer overflow, attacker needs to get the address of the buffer. If he can get the exact copy of the program that will run, then he can do that by running a debugger
If we make it hard to find address of the buffer then we are fine. One way is to randomize the top of the stack. Return address is absolute but the location of the return address in the stack can be computed by the size of the arguments, local variables etc.



The Stack

main calls *foo* which decrements the *sp* by some number. Each new decremented *sp* is relative to the previous *sp* and hence changing the stack top every time a process starts, is going to change the *sp* that is stored in return address. Here attacker cannot be prevented from accessing the buffer and hence the return address (RA) and can only be prevented from entering the absolute address in RA. This solution only prevented code injection in buffer accessed from RA written by the buffer overflow.

More randomization for other kinds of attacks

- randomize stack top
- randomize *libc* location
- randomize code location (recompile the code to not use absolute addressing or one shot randomization by rewriting the binary)
- randomize data location
- interframe padding (code needs to be invasively modified to insert padding in between the local variables etc. and return address)
- rearrange function (can be done with an added level of indirection in the code for *jmp* to the functions)
- rearrange basic blocks
- rearrange local variables (the last 3 including this make it real difficult for return to code attacks)
- rearrange data
- randomize *malloc*s (for preventing exploitation of the serial order of allocation by *malloc*s

Not all of the above make it difficult for any attack. Each of them mostly is attack particular. As discussed in the review paper, on a 32-bit system, address space randomization makes the attacker guess 22 bits to attack the system. It is shown that it is easy for such a guess to find the code and get in. Solutions include using 64-bit etc.

Pointguard

Most of the solutions discussed can be made worse by a format string attack previous to the buffer overflow attacks.

```
void foo (char *c)
   {
      char buf [1024];

      strcpy (buf, c);
      return;
   }
```

We can see above that buffers are usually *char*s or *int*s. RA is a pointer type and buffer is not. We don't use pointer types for buffers since we usually do not read in pointers into a buffer. So if the buffer is tagged (the RA is tagged as *p* and the buffer with its type in the memory. So these tags will be overwritten with incoming data type every time there is a memory write to a tagged location. So if we write *int* buffer overflow to RA, then while loading RA we see that it is not a pointer type and raise an exception, possibly. So only a pointer RA is used to *jmp*. Question is where to store the tag?
Another idea is pointers are stored in the memory encrypted and *int*/ *char*s etc. are not. At startup pick a key *k* and encrypt all the pointers. So overwriting will

give garbage upon decryption. To make encryption/ decryption faster, XOR is the encryption method used.
If the key is in register it is the safest bet. But if for some performance/ processor limitation reason we can’t store the key in the registers, it can be stored in the memory and can be read. By a format string error, we can print out an encrypted RA and XOR it to the RA (which we know already) to get the key back.
Another method is to use a bigger landing pad (discussed in previous classes) to make the last bits of the buffer address redundant and hence make it easier to guess the address.

Stackguard

Problems with recompiling to separate buffer and RA include problems across old and new libraries. Hence here the idea is; at startup of the process pick random *canary* (a 32-bit number) and write the *canary* below RA. So *canary* cannot be guessed and if overwritten by a buffer overflow (that overwrites the RA too after overwriting the *canary*) we can catch the buffer overflow by comparing it to the *canary* and check before the RA is loaded.

Lecture Notes for 04/04/06: UNTRUSTED CODE

Fatima Zarinni.

Last class we started to talk about the different System Solutions for Stack Overflow. We are going to continue the subject.

Stages of Stack Overflow:

The following are the steps for performing a buffer overflow attack:

1) Find a bug in the code.
2) Send overflowing input.
3) Overwrite return address to point to buffer.
4) Jump to *RA
5) Execute code.
6) Make System Calls.

Solutions for Each of the Above Steps:

1) Static Analysis.
2) CCured.
3) Address Space Randomization(ASR) and PointGuard
4) Stack Guards
5) ISR (Instruction Set Randomization)
6) System Call Randomization.

We talked about Steps 1 through 4, today we are going to talk about steps 5 and 6.

Instruction Set Randomization:

Instruction Set Randomization is used in order to prevent an attacker from running his injected machine code on the victim machine.

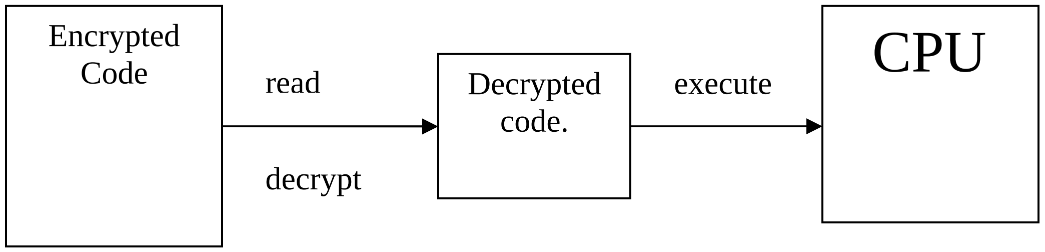
Assume that an attacker finds a way to inject some machine code on the victim machine. Now, the attacker wants to inject machine code that would be executable on the victim CPU. So, in order for the attack to work the attacker needs to know the following:

Attacker must know:

- ➢ OS of the victim machine.
- ➢ Instruction Set of the victim machine.

So, our goal is to not allow the attacker to know the instruction set of our machine. We can do the following:

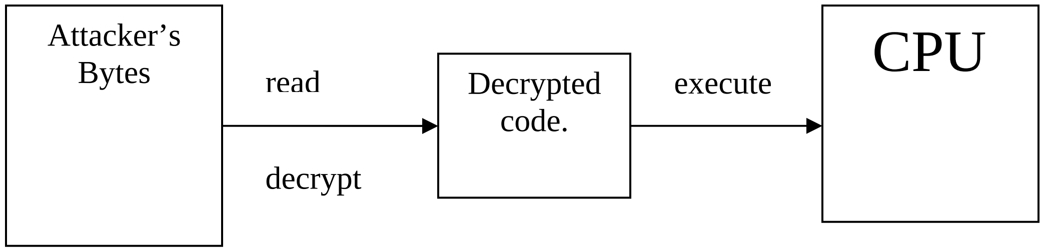
1) When executing code, we encrypt it and load the encrypted version into memory.
2) Right before executing an instruction, we decrypt it and give it to the CPU.



In the above case encryption is only for code and not for data. The idea is similar to PointGuard.

So, now what happens when attacker finds vulnerability in the code and tries a code injection buffer overflow attack?

Assume that the attacker successfully injected his code in the memory and made the return address point to his injected code. So, when the program gets the return address, the attacker succeeds in making the PC point to his bytes. So, now the following scenario would happen:



CPU decrypts the Attacker's bytes before executing. So, if attacker did not know the Key the decryption process would give random bytes and then these random bytes would be executed by the CPU. Hence, if the attacker did not know the Key, it is less likely that he would get what he wants. Now, if we feed anything to the CPU, then the program would just crash.

Thus, the above technique is called Instruction Set Randomization.

Let us look at some comparisons:

- ASR :
  - Can be used on a binary file very automatically.
  - You do not need to change source code.
  - It is very easy to use.
  - Weaker guarantee, but less work

- Static Analysis:
  - Strongest Guarantees.
  - Lots of hard work.

- CCured:
  - Works when there are buffer overflows.
  - Moderately successful.

- StackGuard:
  - Stack Guard is quite successful.

- Point Guard:
  - PointGurad is not successful.
- ISR:
  - Too much overhead
  - It only works for code injection buffer overflow attacks.
  - Not Successful.

Now, let us look at System Call Randomization.

System Call Randomization:

Permanent damage can be caused by system calls done by the attacker. Hence, we need to prevent attacker from performing system calls.

So, how does a system call work?

Let us look at the following assembly code:

```
push eex      (number of bytes.)
push eax      (address of buffer.)
push ebx      (file descriptor.)
push 127      (read system call. 127 represents a system call type, maybe a read syscall.)
int 0x80
```

The last line asks the OS to perform the system call.

How can we prevent an attacker from performing system calls?

- Randomize system call numbers.
    - So, each time when a program is run, all system call types would get a different number.
    - In our example above, 127 is a system call number.

- Randomize system call argument orders.

- Change system call instructions.
    - Update system call table.
    - Change C library.
    - Change Kernel.

- Rebuild the entire system to perform random system calls.

At what stage of the buffer overflow attack would System Call Randomization be used as defense?

After the attacker succeeds in overflowing the buffer and changing the return address in order to perform a return-to-libc attack and when he uses code already present, to do system calls.

For System Call Randomization, defense has high overhead and low security.

[New Topic Starting from Next Page]

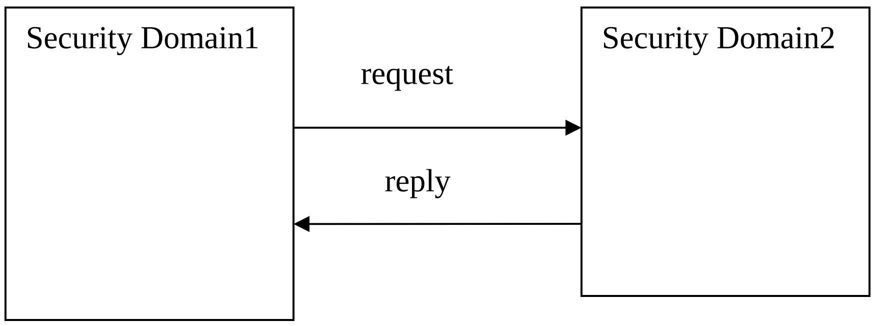
Software Fault Isolation:

Theme: The direction of Computer Security research is towards finer grain sharing.

In the beginning of the semester we saw the example of a closed room with an operator. There was no sharing on that system. Then we saw the VM architecture in which VMs did not share memory, but they could send messages back and forth. Now, one process can make request to another process on Unix.

The problem is that everything is very slow.

Sharing and IPC:



Unix Processes.
(Using Pipes)

(Pipes sort of work like method invocations. Security Domain 1 requests something from Security Domain 2, and Security Domain 2 replies back to Security Domain 1.)

- Messages can go back and forth via pipes.
- Requests go as string of bytes.
- Replies come back as string of bytes.

A lot of things have to happen for making messages go back and forth between processes:

- There must be a context switch to stop 1 process from running and save the state of that process. Then load the state of the second process, and start running the second process.
- So, Context Switches are expensive.

So, now let us look at some efficiency issues that arise due to calls between different domains.

Function call	0.1 micro seconds
Pipe IPC (Send 1 byte over and get 1 byte back)	200 microseconds. (2000 times slower than function calls.)
SFI IPC	1 micro second.

Since Pipe IPC is very slow, therefore we cannot make a lot of these calls. If we want to transfer a lot of data, Pipe first copies data from domain 1 to the OS, and then from the OS it copies to domain 2.

One important operation in SFI is that it reduces the number of copies to just 1 copy.

<u>Our Goal:</u>

Keep two security domains that do not trust each other in a single address space and do calls much faster.

How do we do faster inter domain communication?
We must get rid of Context Switches.

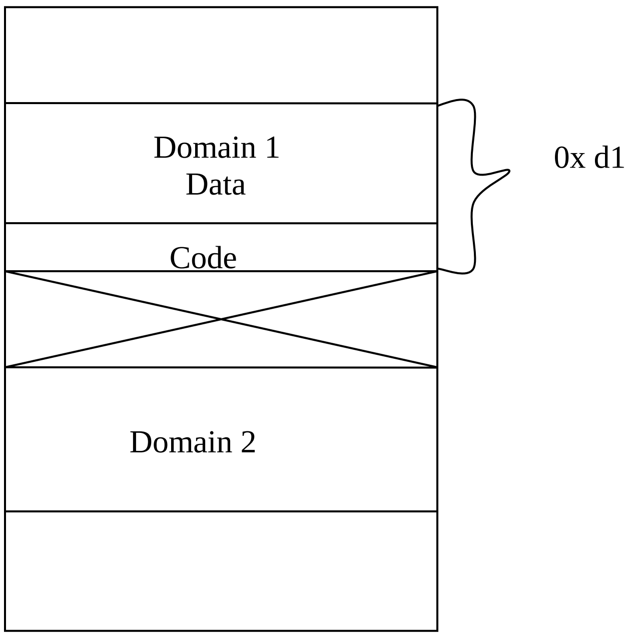
So, the following is what we want:

- Domains must exist in the same address space and process thread.

    - Inside a 1 Unix process we can have two domains that do not trust each other and live in the same address space and run correctly.
    - If we could do that then I don't need a pipe or context switch between the two domains.

- Is memory protection an issue?
    - System call overhead is so much that we do not want to involve the OS.

So, again, we want to make two domains live in the same address space without allowing them to step on each other's toes. How can we do that?



Address Space

So, we want to make sure that whenever process is running code in domain 1, we cannot step on stuff that is in Domain 2, and vice versa.

Idea:

To be able to do the above we must precede every memory access by the process with a bounds check.

So, let us look at the following sample code:

Domain 1:

```
add eax ebx
mul ecx eax
mov ecx [eax]   ← Moves ecx to memory location at eax. We must check that eax points
                  into domain 1.
```

So, we need to fix the above third line of the code.

Rewriting the code with the fix, which ensures that eax points in to a location of domain1:

Domain 1:

```
add eax ebx
mul ecx eax
sethib eax 0xd1 // Sets the high bytes of eax to 0xd1
mv ecx [eax]
```

So, this fixes the problem of domain 1 trying to access domain 2's memory locations. Is there any way we could get around this fix?

We can have a jump statement right before the sethib statement, in order to jump to the "mv ecx [eax]" instruction. To prevent this we must have Control Flow integrity.

Control Flow Integrity:

Sub Problem to Solve to be Able to Achieve our Goal:

- All basic blocks must execute from the beginning.

What is a basic block?

A Block in a program is a sequence of consecutive instructions that do not have any jump statements, except that the last instruction can sometimes be a jump. This jump statement gives us the decision of where to go next.

Example:

Let us say that we have the following C code:

```
x = 7;
y = x + 5;
if (x > 11) {
        printf ("Hello");
}
else { exit(0);
     }
```

The corresponding assembly code is:

Basic Block 1:

```
mov 7 eax
add eax 5 ebx
comp ebx, 11
jge L1
```

Basic Block 2:

```
        :
        :
        :
call printf
jmp L2
```

Basic Block 3:

```
L1:
        call Exit
```

// End of Example!

So, we want the code for Domain 1 and Domain 2 to execute basic blocks from the beginning, because we want to make sure that all accesses to memory locations are preceded by checks.

So, for example, we want to make sure that
“sethib eax 0xd1” is executed always before
“move ecx [eax].”

So, Control flow integrity would do the following:

- It would introduce a new instruction “label Tag” at the top of each block.
- “label” does not do anything.
- “TAG” is a 4 byte number
- We must make sure that each jmp statement is going to the label instruction of a building block.

So, for example control flow integrity would add the following instructions to a block of the assembly program:

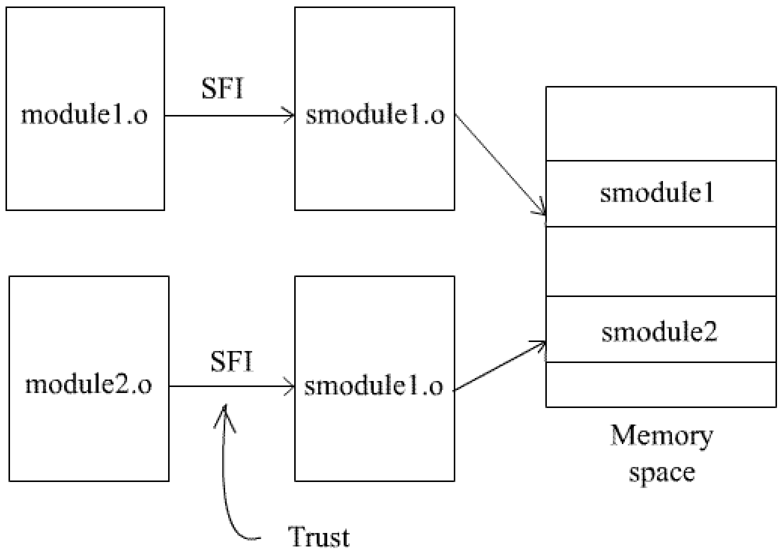
```
Label TAG
:
:
cmp [eax + 4], TAG  // Check if jump would actually jump to the beginning of a
jne ABORT               // basic block. If not then abort.
:
:
jmp eax
```

Now, there is another issue. Code comes from un-trusted domain, so how do we know that checks exist?

- We can re-write the code dynamically to include checks.
- We can have compiler to insert the labels and checks.

SFI and CFI

To recap, the main idea of software fault isolation is to have two modules that do not trust each other to safely execute in the same address space without reading/writing each other's data and without making jumps to each other's code.



The binaries of the modules are disassembled by the SFI translator and checks are inserted before jump or read/write instructions to restrict the modules to their own domain and the fault-safe modules which are output can safely execute in the same address space.

It should be noted that all the trust lies in the SFI translator, which is relied upon to correctly transform the input modules into safe modules.

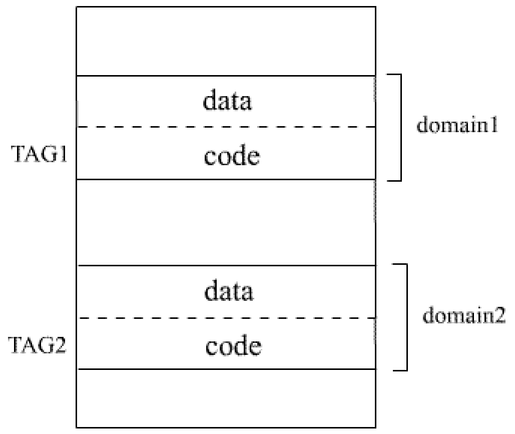
Basic Block Property: Every basic block executes from the beginning.
(A basic block is a sequence of instructions that do not have any jumps. A basic block ends with the appearance of a jump instruction of some kind.)

The primary goal of SFI is to restrict any module to a certain subset of the address space.

We can insert runtime checks to ensure that each module makes read/store calls only to its own domain.

However, code from domain 1 could try to jump into the middle of some code from domain 2. We need to prevent such jumps between domains.

To solve this, we have a unique tag for each domain, and every basic block in any domain begins with a **`label TAGx`** statement, with **`TAGx`** specifying the tag for that domain.



For instance, every basic block in domain 1 would begin with **`label TAG1`** and every basic block in domain 2 would begin with **`label TAG2`**.

Then before every jump, a check could be inserted to make sure that the jump points to a basic block for that particular domain. An example is shown below:

```
cmp eax, TAG1   <- compare target with tag
jne abort       <- abort if mismatch
jmp eax         <- actual jump call
```

There is still a problem. Some code from a domain could try to write to its own code and write some malicious code that it can later execute. Solutions:
- the code can be made read-only
- similar runtime checks can be inserted and it can be ensured that write operations take place only to the data area and not the code area.

We have thus achieved the basic block property and prevented cross-domain read/write/jumps. But we still need to let domains call each other. This is addressed in the section below.

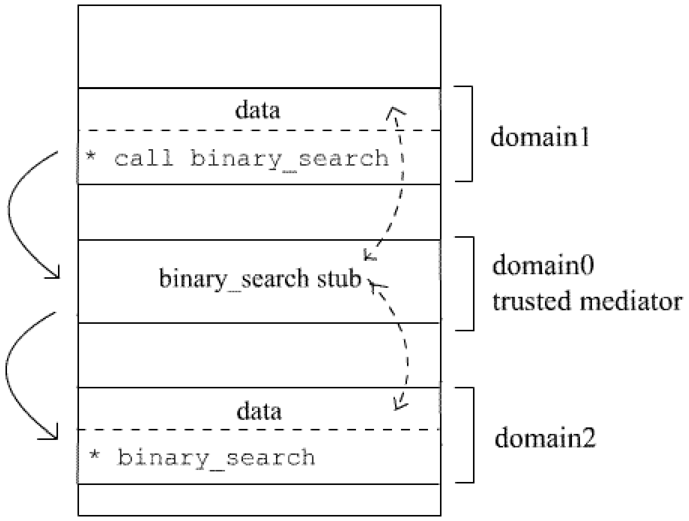
Cross-domain calls via stubs

A stub is a trusted third domain that mediates between two domains that want to call each other.

For instance, let us assume that domain 1 wants to use a binary search function in domain 2. The code in domain 1 actually calls a stub which exists in domain 0, the trusted mediating domain. Domain 0 can read and write data to both domains 1 and 2. It can also jump to a basic block in either domain.

The caller should still be able to jump to the stub in domain 0. Hence, this block in domain 0 is labeled with **`TAG1`** so that code from domain 1 can jump to it. The stub then reads data from domain 1 and passes it to domain 2, and then jumps to the search code in domain 2.

When the processing is done, the code in domain 2 must be able to jump back to domain 0. Hence this block is labeled with **`TAG2`**. All other blocks in the stub are labeled with **`TAG0`** and so neither domain can jump to other code. The trusted domain then reads the results from domain 2's data, writes them to domain 1's data and jumps back to domain 1.



Performance:

In terms of performance, SFI provides low overhead for frequent domain crossings. The relative performance figures are as follows:

Pipe	:	200 microseconds
SFI	:	1 microsecond
Function call	:	0.1 microseconds

SFI also supports flexible security policies that are independent of the hardware and the operating system.

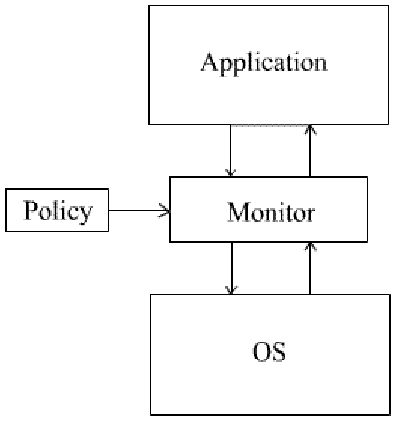
System Call Interposition

A user may want to restrict the system calls that a downloaded or untrusted application makes. This may take the form of completely blocking certain system calls, or disallowing certain parameters. System call interposition is a method of checking system calls and blocking/changing them if necessary.

System call interposition can be used to configure a user's applications to a restricted access policy. For instance, if a user wants to ensure that a downloaded version of

Photoshop that is untrusted does not write files under C:\Windows, system call interposition can be used to check such a condition.

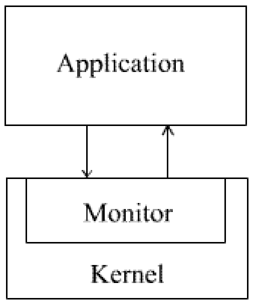
This is generally achieved by placing a monitor program between the application and the kernel. All system calls go through the monitor and the monitor enforces a user-supplied policy.



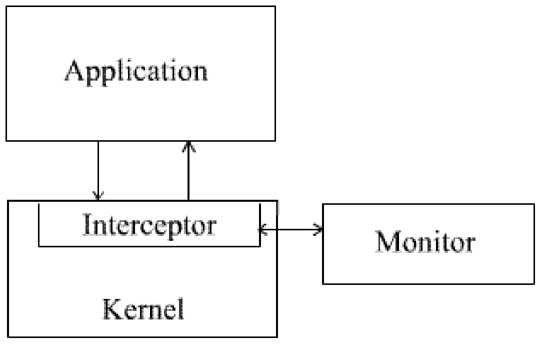
<u>Where to locate the monitor</u>:

A naïve approach would be to place the monitor directly into the kernel at the entry point of system calls. This way, all system calls would go through the monitor which would then pass them to the kernel.

This avoids context-switches during calls, but a monitor may be thousands of lines of code and it can make the kernel bloated.



A better solution is to have a thin interceptor layer within the kernel which intercepts the system calls, and then passes control to the actual monitor which lies outside the kernel.



However, this simplistic method is susceptible to a race condition, as described below:

Consider that the malicious application has multiple threads. Initially, the normal thread of execution is running, and makes a system call to open a file in the user’s home directory.

	Thread 1	Thread 2
Application	`fname = “/home/usr/somefile”` `open(fname)`	
Kernel	Interceptor passes `open(fname)` to monitor	
Monitor	Monitor fetches string pointed to by `fname` from application’s memory space; monitor checks and approves	
	Scheduler executes thread 2 ---->	
Application		Application overwrites `fname` with `“/etc/shadow”`
	<---- Execution returns to thread 1	
Kernel	Kernel fetches `fname` and executes `open` call with the filename pointed to by `fname`	

With a little luck, the scheduler will execute the second thread at exactly the right moment, after the monitor has approved the call. The filename will be overwritten with a file that the application should not normally be allowed access to, and when execution returns to the main thread, the kernel will execute the call with the new filename.

* * *

Detecting Intrusions..

April 20, 2006 mnair@cs.sunysb.edu

Last Class

The structure given in figure 1 is very similar to Ostia/Sandboxing. The *policy* is a sequence of system calls which confirms Apache to the correct behavior. Here Apache is used just as an example. A similar design can be use for almost any network based service.

This design is based on the assumption that when a attacker injects errors, there will be anomalous system calls. "exec("/bin/sh")" the kind that cannot be expected during ordinary execution. The information of correct behavior, called *policy*, is checked by the monitor. We develop methods for extracting and storing(representing) these policies.

Model 1

1. Run the Apache on a safe controlled environment.
2. Learn the set of system calls that Apache uses (Learning Phase). This is stored in a set 'S'.
3. Allow the apache only to make the system calls in the set 'S'

This is a dynamic analysis method.

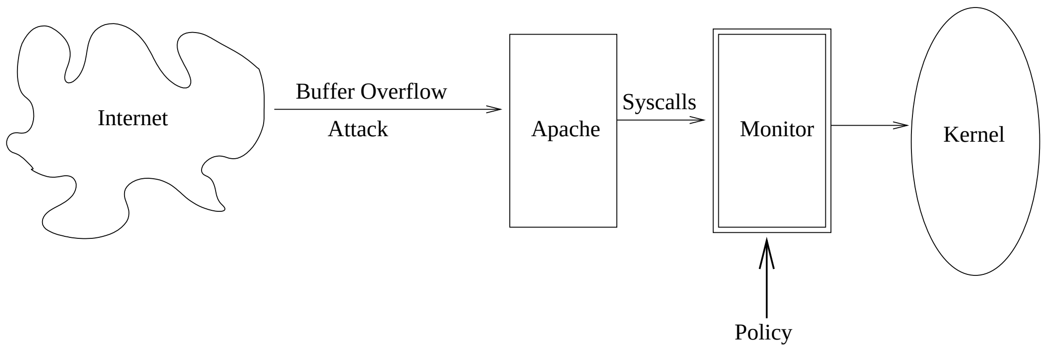


Figure 1: Schematic Model of Scheme

Model 2

1. grep the apache source for syscalls
2. Collect the set S of syscalls apache uses

This is a static analysis method.

Static vs Dynamic

The two common problems with Intrusion Detection Systems are:

- **False Positives** Events flagged as dangerous but are not really threats. Analogy : "Boy who cries Wolf".
- **False Negatives** Threats which are missed by the detection systems. Analogy : "Sleeping Security".

The limitation of static method (model 1) is that it is dumb. For example it will parse through comments also. Even if an intelligent parser is made which skips all the comments. There might be areas of the code that are not necessarily executed, or might be executed conditionally, which might always allowed by this scheme. Thus more often than not we end up getting a very large set. Thus leading to false negatives.

The dynamic analysis (model 2) firstly requires a safe environment for running. Code coverage is very important; we need to exercise all the possible execution paths. If we miss any of the path it could lead to a set S that is not complete – smaller that it should. This means that there will be high probability of many false positives. This could be very irritating to the user and be unusable. "Unusable security becomes Unused Security".

Model 3

```
1: read(...)
2: if(...)
3:      write(...)
4: else
5:      exec(...)
6: endif
7: read(...)
```

If either model 1 or 2 are used then the set 'S' would contain the following operation {read, write, exec }. Thus if the following sequence of operations

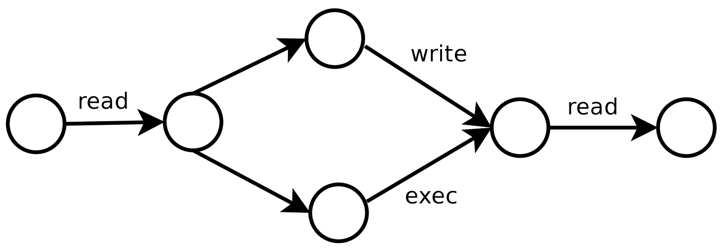


Figure 2: Finite State Automata for the code segment

is seen, read, write, exec, read is seen, it will be allowed. Clearly this is a bad sequence and should not be allowed. Unfortunately our model is not powerful enough to handle these cases. We use a more intelligent design to store our information. Instead of a set we use a finite state automata.

Figure 2 represents the finite state automata for the code segment shown in the beginning of the section. This automata is generated statically. One of the important things is that we consider only system calls. We are only trying to catch privilege abuse not logical errors. All other states are collapsed. Also by construction this is a non-deterministic automata. But by an important result from Automata theory we know that a non-deterministic Finite state Automata (FSA) is equivalent to a deterministic FSA. Thus we a build a deterministic FSA from the generated non-determinitic FSA.

When this model is used, with each system call, the monitor traverses states in the automata. If from a state, a call occurs that does not lead to a valid state, then we know that a error is present. We can then take preventive action; such a kill the offending process.

Model 4

In this model too, a FSM is used to store the results of the allowed actions. But the difference is that the FSA is generated dynamically when the process is being run in a secure environment. This is similar to Model 1 but rather than a set an FSA is used.

Consider the code segment shown below:

```
while(...)
{
    if(flag)
    {
```

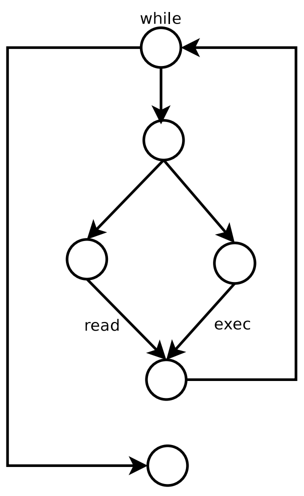


Figure 3: Problems with Models 3 & 4

```
        exec(...)
    }
    else
    {
        write(...)
    }
}
```

If we know that flag does not change within the while loop then we know that the regular expression that can occur with this code segment is $(read)^*|(exec)^*$. Rather if the automata is used along with either models 3 or 4 the following will be the regular expression of the automate $(read|exec)^*$. Figure 3 represents the automata of this (wrong) expression. This means that sequences such are read, exec, read are allowed by the automata but this is not what the code is intended to do.

The main problem is that automata does not store any state. We are unable to decipher where we came from. This issue is solved in the next model.

Model 5

This model is similar model 3. We generate the automata statically but rather than a FSA, we use a Push Down Automata(PDA). In a PDA, apart

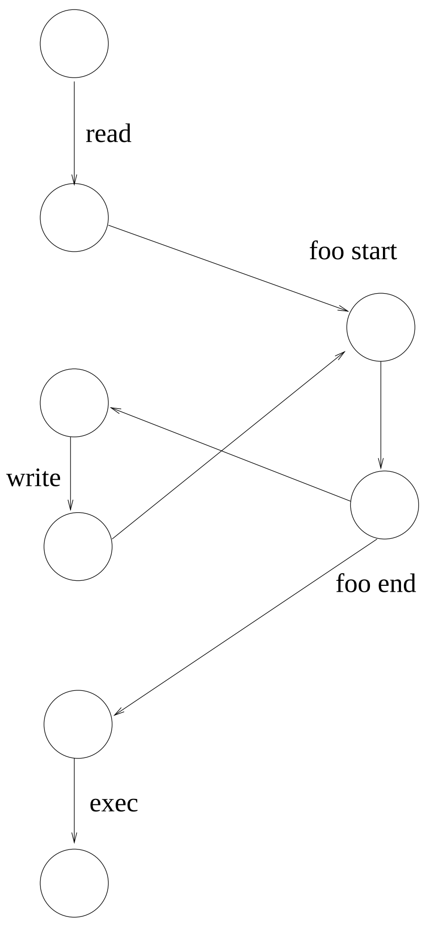


Figure 4: Finte State Automata for “foo” segment

from the states we also have a stack. The stack can be used to record from where we came from.

Let *foo(...)* be a function from which no system calls are made. According to our model foo can be collapsed into a single node. Consider the following call sequence:

```
read(...)
foo(...)
write(...)
foo(...)
exec(...)
```

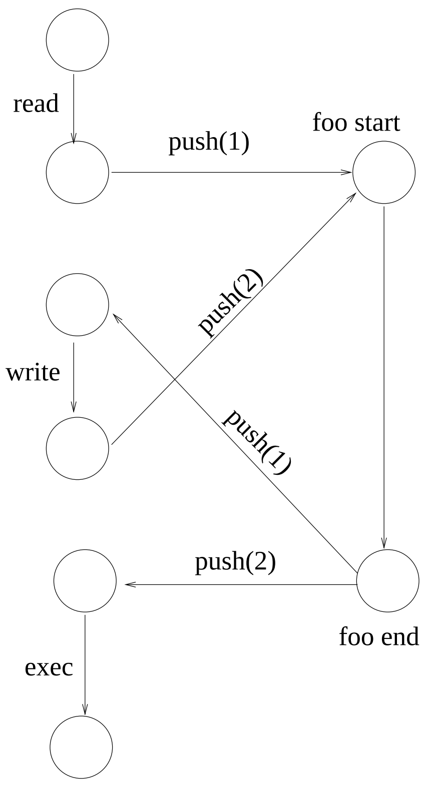


Figure 5: Push Down Automata for "foo" segment

Fig 4 shows the FSA that is generated from the above sequence. It is clear that such an automata would lead to the wrong result. It is also clear that it is an inherent problem with finite automata. We solve the problem by using push-down automata(PDA). Fig 5 show the PDA generated by the code segment. PDA does not suffer from the problem that FSA had.

Unfortunately PDA are not a very efficient. This is because unlike a FSA a deterministic PDA is not equivalent to a non-deterministic PDA. Thus if there is ambiguity the stack could grow infinitely.

Consider the non-deterministic PDA shown in figure 6

```
recursive_read(...)
{
    if(...)
    {
        read(...);
```

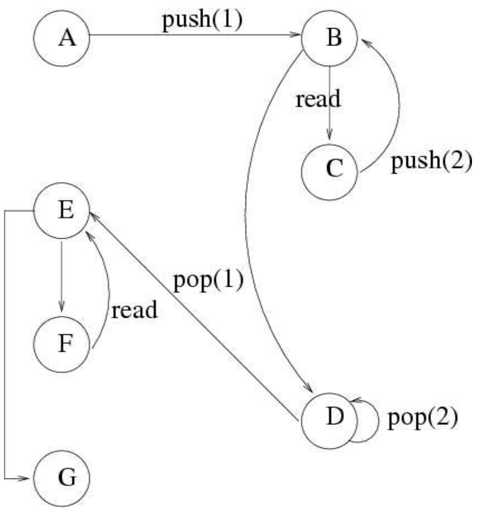


Figure 6: A non-deterministic Push Down Automata

```
        recursive_read(...);
    }
}

other_func(...)
{
    recursive_read();
    while(...)
    {
        read(...);
    }
}
```

For the above code snippet make the PDA shown in figure 6. Here the monitor is unable to decide whether the call is attributed to the read in other$_funcorwhetherthecodeisstillinrecursive_read.Fundamentallythereisambiguityortwoposs$

Performance Issues

Monitor needs to see what is the sequence of calls and returns. Only then can we ensure that the PDA is determinate. One of the ideas suggested to solve this is introducing two new system calls:

Approach	Performance
Set-Based	Very Efficient
Finite Automata	Efficient
Push-down Automata	Unusable (efficient with tricks)

Table 1: Performance Comparison

· call_function(...) - called just before a function call takes place.
· return_function(...) - called just after a function returns.

To both the system calls we pass the function pointer to the new calls. Unfortunately the downside with this method is the high overhead. This make each function call make a system call. While a function call involves only 2 instructions, a system calls involves atleast a 1000 instruction.

A better solution to the problem is to use macros to store the call and return pointers. These pointers are stored in fixed location in the memory. This call return sequence can be used to determine the calling function and resolve ambiguities.

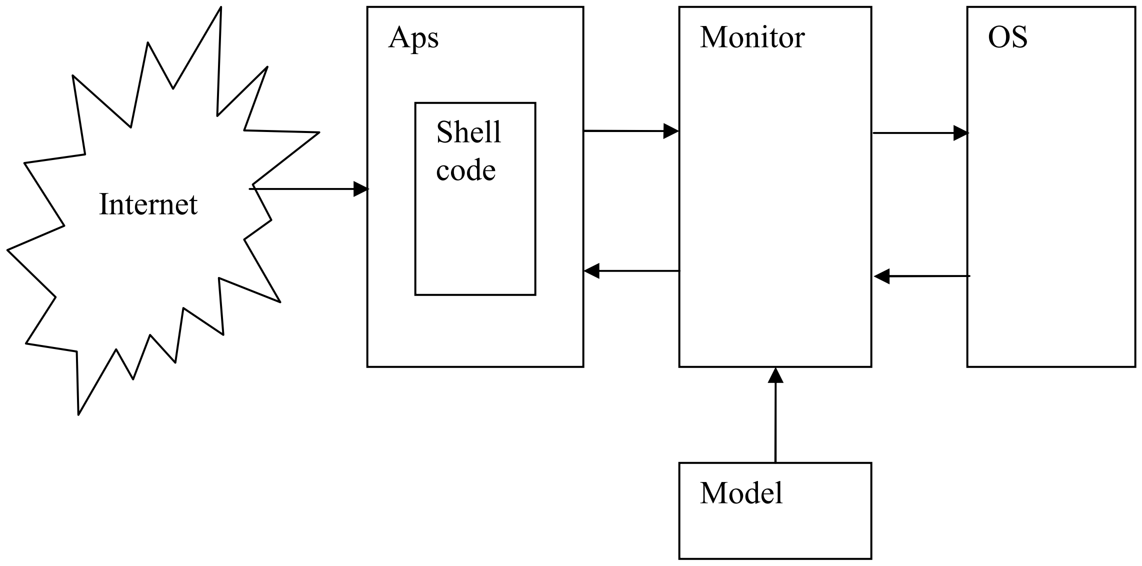
Some Improvements

In all the models discussed, we only look at the system call and not its arguments. Sometimes it is useful to look at the arguments. For example the Apache may execute only some binaries but never *exec("/bin/sh").* This could lead to a more intelligent monitors. Other improvements could be looking at the looking the relationship between the arguments. For example the monitor can guess that if a particular file is opened and a file descriptor is obtained, then read/write operations and finally close syscall should performed on the same descriptor; never on a descriptor which has not been obtained from open.

4/25/2006 Lecture Notes: DOS
Beili Wang

Last lecture we talked about how Intrusion Detection works. Today we will talk about the attacks.

Intrusion Detection



In Intrusion Detection, we call "Model" instead of "Policy", because "Model" is derived from the application itself. It describes the correction behavior of the application, what the application allows to do, and what system calls that the application can make.

The attacker writes the shell code to attack.

Mimicry Attack:

Mimicry Attack:
Shell code agrees with the Model.

1. Simple Model: Set of system calls.
   How the attacker will attack if he knows the set? The attacker writes the shell code that the monitor pass to model and the model says the code is ok. → shell code agrees the model.
   Shell code only uses system calls in the set of allowed calls.
   The attacker can only use the system call in the set to write the shell code. He may download Aps and just use the system calls allowed in the Model.

2. FSA (Finite State Automaton):
   What about the FSA model? Shell code agrees with FSA.
   The attacker has to
   1. download Aps
   2. construct model (same model as the victim's)
   3. search for sequence of system calls accepted by FSA and that violates security

   Typically, this is not too hard. The attacker can calculate average of number of options for every step. It is only slightly more complicate than the Simple Model.

Static models:	Dynamic models:
➢ over approximation	➢ more precise
➢ valid sequence accepted	➢ mimicry attacks are harder
➢ no bogus warnings	➢ may get false positives
➢ easier mimicry attacks	

Mimicry attack: any models that get shell code agrees with model.

What if attack does not make nay system call?
What kind of attacks does not make system calls:
- DOS attack (Denial – of – Service)
- changing important data structure
  - authenticated flag
  - copy security data into outgoing buffers (for example: gain private key)
  - others (mimicry attack more flexible)

Therefore, Intrusion Detection cannot prevent attack but detect attack. It raises the bar for attackers and makes them more creative.

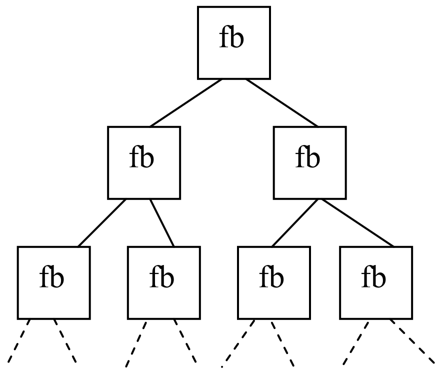
DOS (Denial-of-Service) attack

Resource Exhaustion Attacks:
- Memory
- CPU
- Network
- OS objects
- Disk

For example: fork( ) bomb exhaust:

- Memory
- CPU
- OS objects

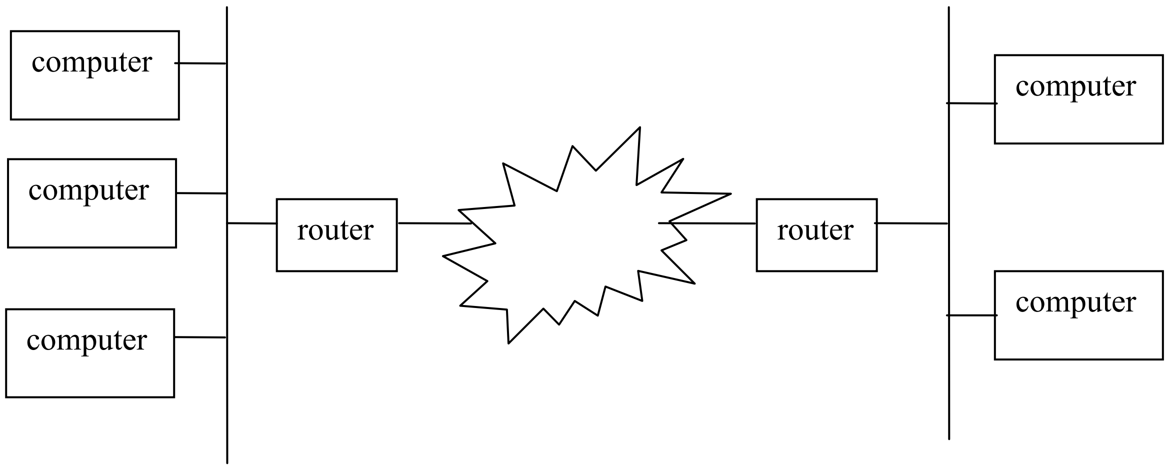
Code:
```
while(1)
      fork( );
```



Another example: Randomly access to a huge array and causes thrashing.

DOS on Networks:

Ethernet



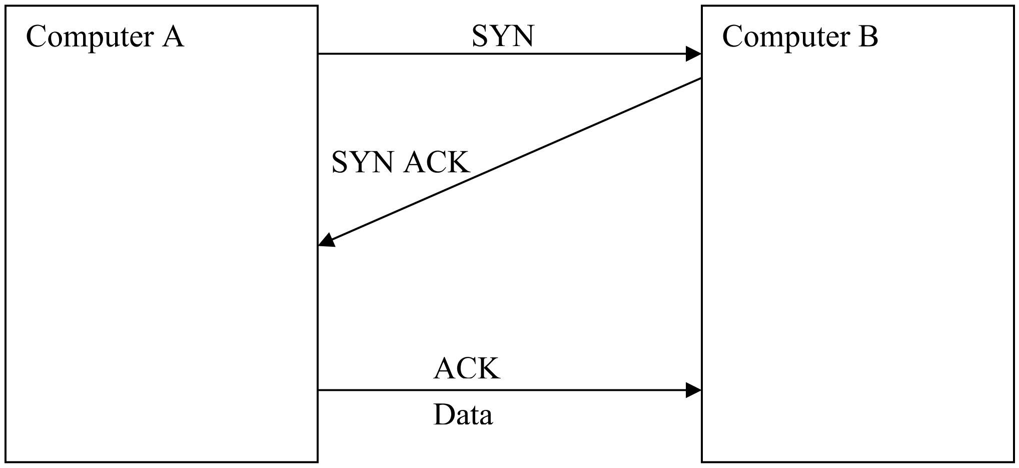
net: 237.128.7.0
(0 – network address)
(1 to 255 for machines)
broadcast: 237.128.7.255

Protocol: TCP (Transmission Control Protocol)

Use of TCP/IP:

- In order (delivery)
- Reliable (If drop, resend again)
- Consecution /flow control
- Ports (to which computer, which process on computer)

TCP: 3 ways handshakes:

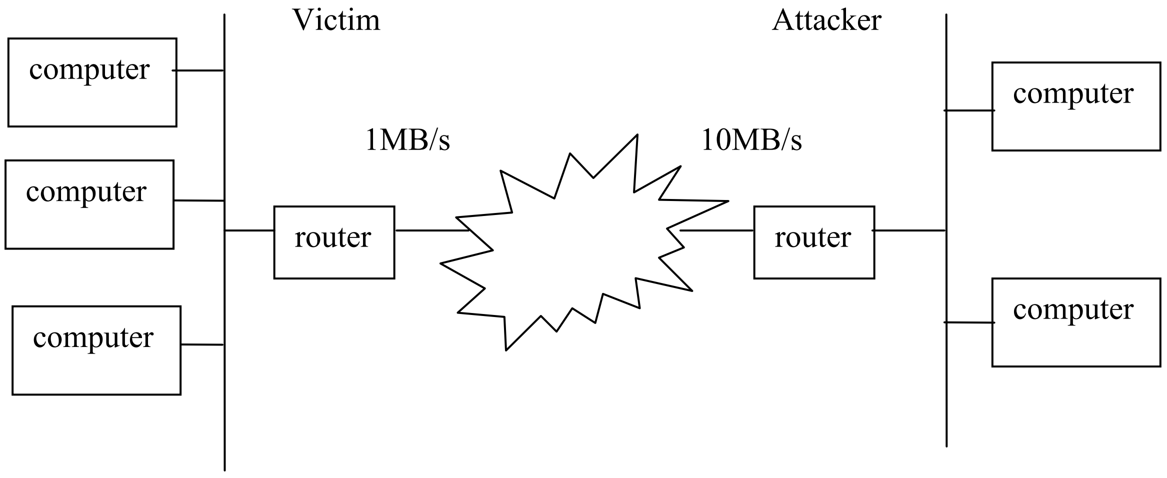


1. Computer A sends SYN packet
2. Computer B replies SYN ACK (acknowledgement) packet
3. Computer A sends ACK includes Data to Computer B

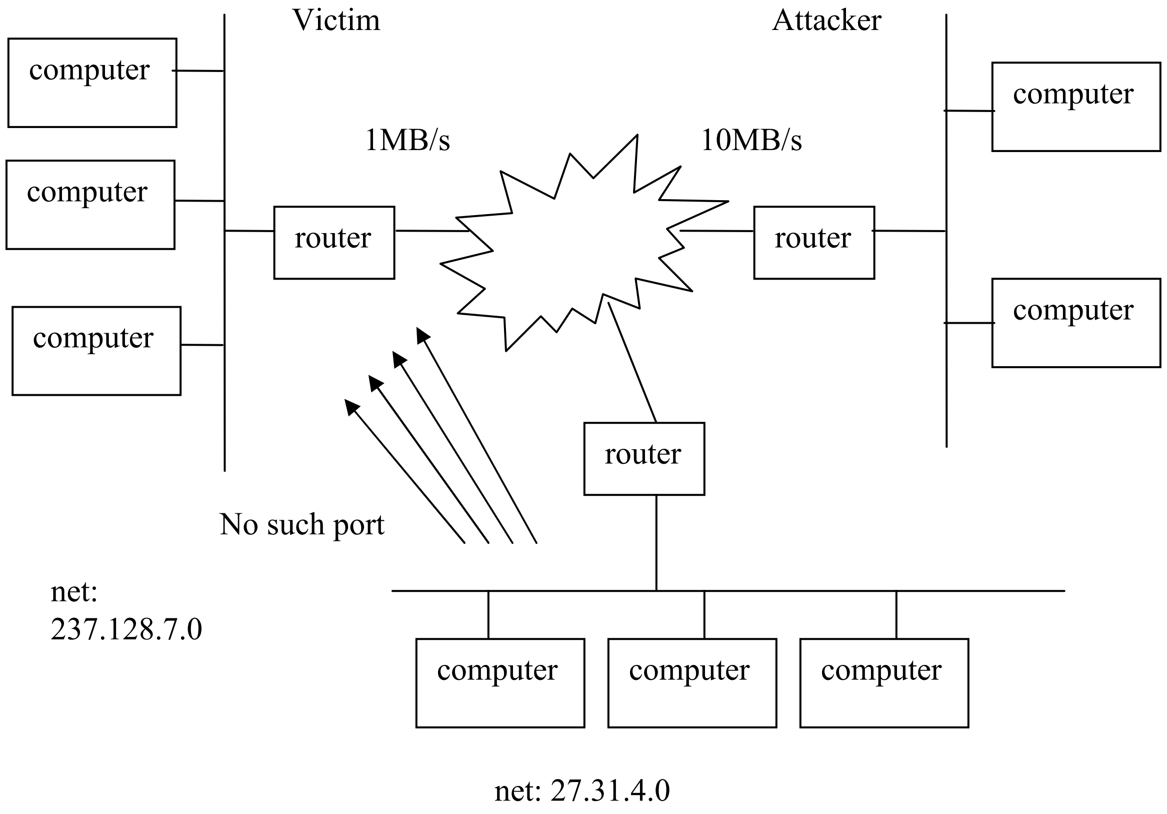
Attacks:

- **Small site – easy (Bandwidth exhaustion)**
- **Medium sized sites – (Smurf Attack)**
- **Huge sites – (Worm)**

Bandwidth exhaustion:
Attacker sends 10 MB/s packets flood down the connection. The victim's 1MB/s router spends all the time to deal with them. No space for the victim's packets in or out.



Medium sized site: Smurf Attack



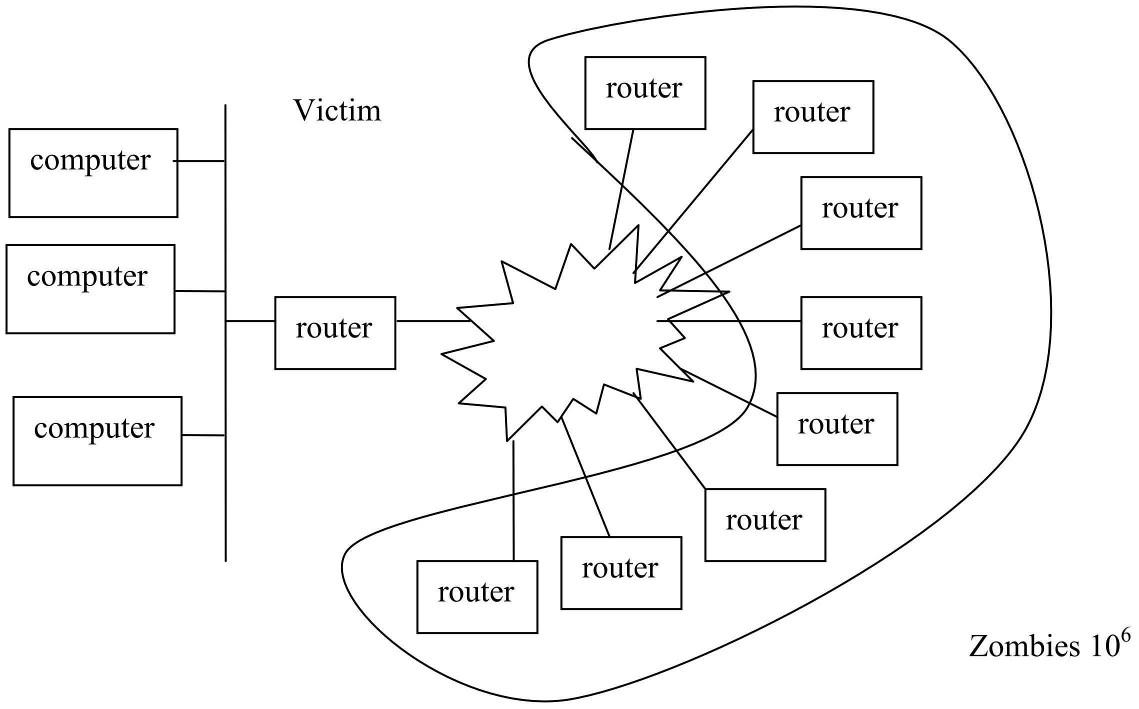
Conditions that make the attack possible on the networks:

1. Attacker can lie about the return address.
2. There is no such web server at the address.

For example, attacker sends package to 27.31.4.255 (broadcast) with port 27,000, so that everyone on the network gets the package. Since there is no process on the 27,000 port, everyone on the network definitely will return the package to the sender. The attacker lies about the "From" address and writes the victim's address: 237.128.7.12, so everyone on the networks sends back the package to the victim. The attacker sends 1 package, but victim receives tones. 10MB/s becomes GB/s.

This attack may not be realistic today, because the router will reject if it uses firewall, net machine, etc.

Huge Sites (for example: Amazon.com) – Worm



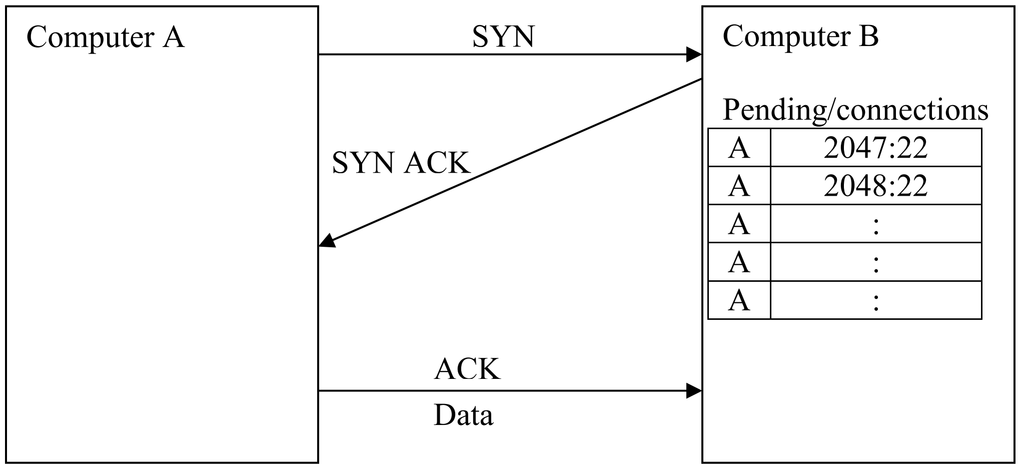
- ➢ Finds buffer overflow.
- ➢ Exploit code (worm).
- ➢ Write 1 command message, all the machine send to the victim.

15 minutes, about 100,000 machines are under control.

Victim side to defeat the attack:

- ➢ **Small site – firewall, drop all the packets from the attacker**
- ➢ **Medium sized sites – drop all the packets from the particular net**
- ➢ **Huge sites – cannot block all these packets from random addresses**
  Solution:
  - ▪ **Get a really big connection**
  - ▪ **Mirror server everywhere**

3 Ways handshake TCP/IP



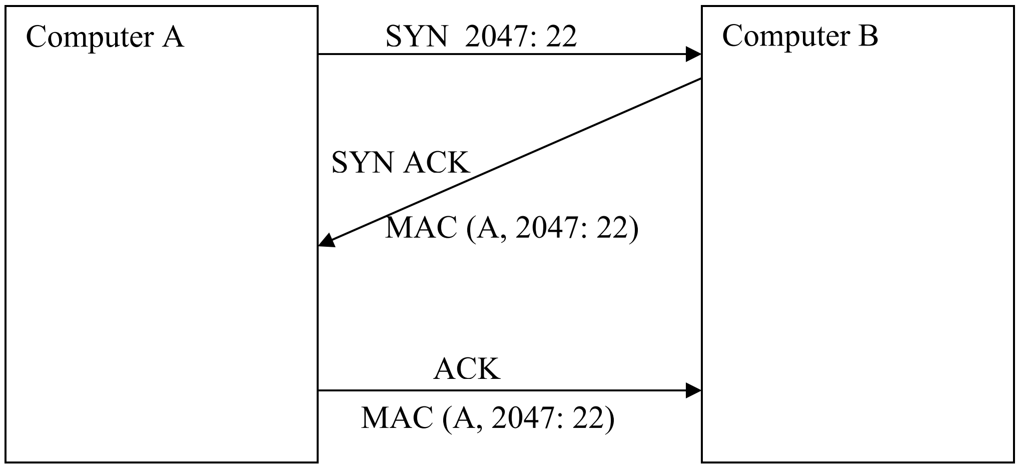
Idea: Computer B has a little array which stores pending connections. For example, Computer A sends SYN message, Computer B stores A with port number 2047 : 22 (22 is SSH) in the array. Then, Computer B sends SYN ACK message to Computer A and waits for ACK message from Computer A. When 3 ways handshake connection completes, Computer B erases the entry with the port in the array. If Computer A does not send ACK message, Computer B keeps the entry and will time out, then waits Computer A to send again.

Attack: If attacker sends all the SYN messages and does not complete any of them, it will take all the entry spaces in the array so that when there is Computer C wants to talk to Computer B, Computer B does not have space available for Computer C and eventually drop message from C. This attack is called SYN Flood Attack.
Early Implementation of TCP/IP only has 5 entries in the array.

Solution: To defeat SYN Flood Attack, use SYN Cookies.

Idea: How to make backwards compatible? The basic idea is used for a lot of sever design to prevent OS resources get used up.
SYN Cookies uses MAC (Message Authentication Code) and security key. Computer B does not store anything in the array or table, but packages up the entry, sends back to client Computer A and let the client store it. Computer B uses MAC so that the client Computer A cannot modify or change the entry. And, to complete the connection, Computer A has to send back the MAC entry with data to Computer B. By this way, Computer B does not need to use the table any more.
Per client:

- Package it up
- Send it to client
- Let client to store it
- MAC so that client cannot modify it
- Client has to send it back again


Computer A
SYN 2047: 22
Computer B
SYN ACK
MAC (A, 2047: 22)
ACK
MAC (A, 2047: 22)

Denial of service attacks

Pramod Adiddam 04/27

Network denial of service (Contd.)

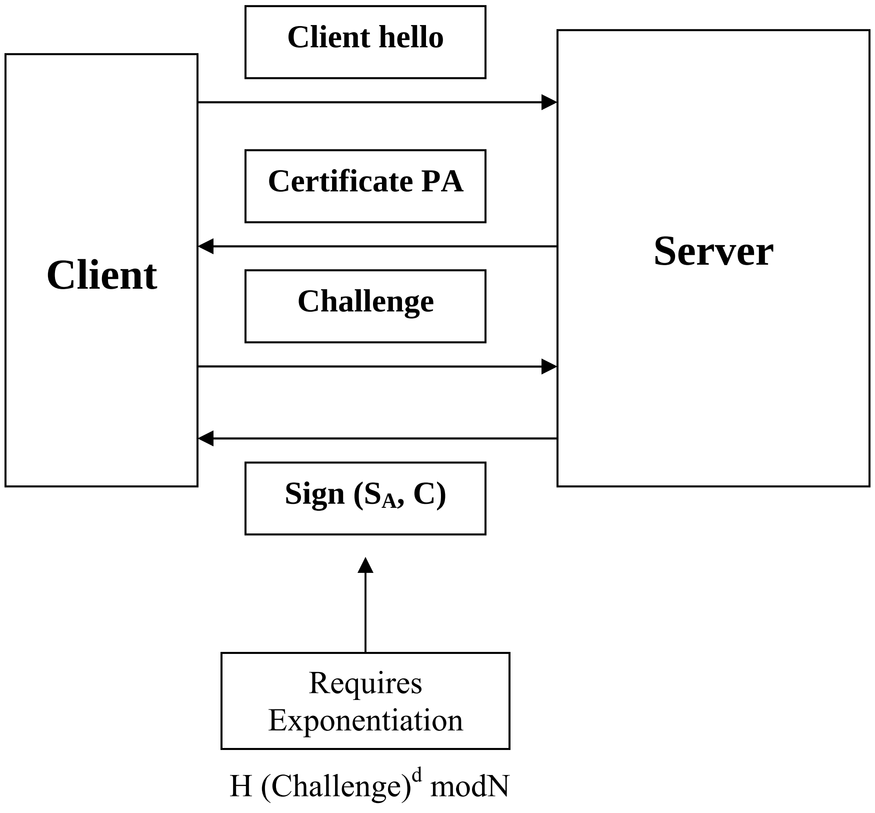
-> Smurf attack – third party broadcast

-> Zombie net - property for the attacker

- huge number of machines (100K zombies)
- attack traffic can be legitimate traffic
- no way to distinguish attackers from legitimate browsers
- CAPTCHAs (degree of sharing not possible otherwise) so other than CAPTCHAs we have very little to go on

Computational Denial of service

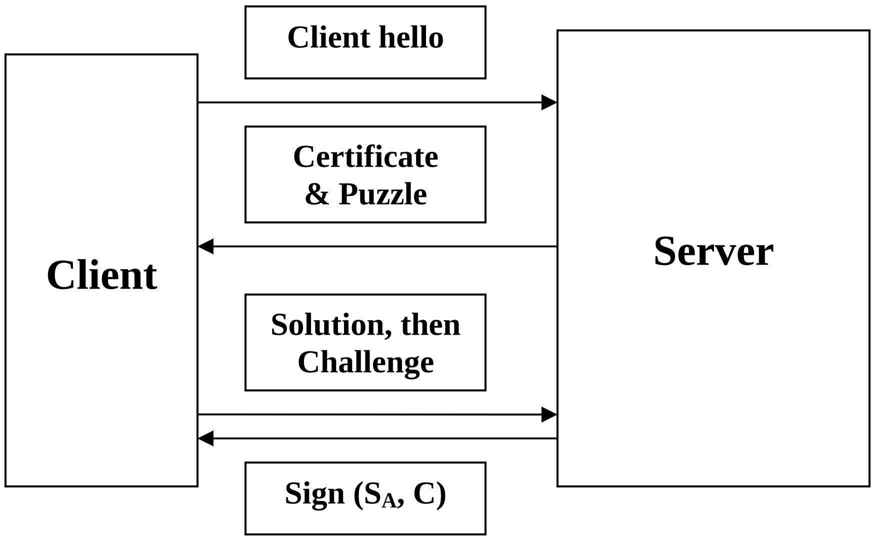
Client puzzles and TLS



d is large and hence there is a huge computation cost on the server side.

Say if the computation cost on the server side is 10ms and the computation cost on the client side is 1ms then the difference in the computation costs between server and client are exploited to overload the server. If client is not computationally close to the server then we can send more requests from a single client or a network of clients more than the server can handle.

To even out the computational costs or fix the computational asymmetry we use client puzzles.

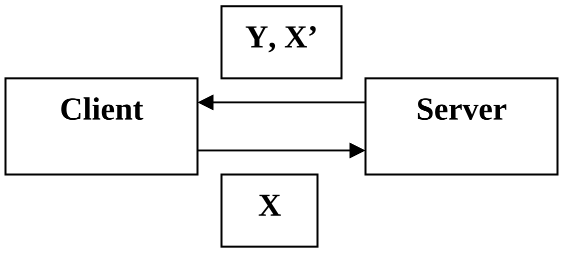


Puzzle should be

- hard to solve (for client)
- easy to generate (for server)
- easy to grade (for server)

Examples of puzzles are:

If h(X) = Y



X' and X differ only in some small LSB bits.

X: 12 32 16 104 16 35 7
H(X): X': 12 37 16 104 16 __ __

We don't

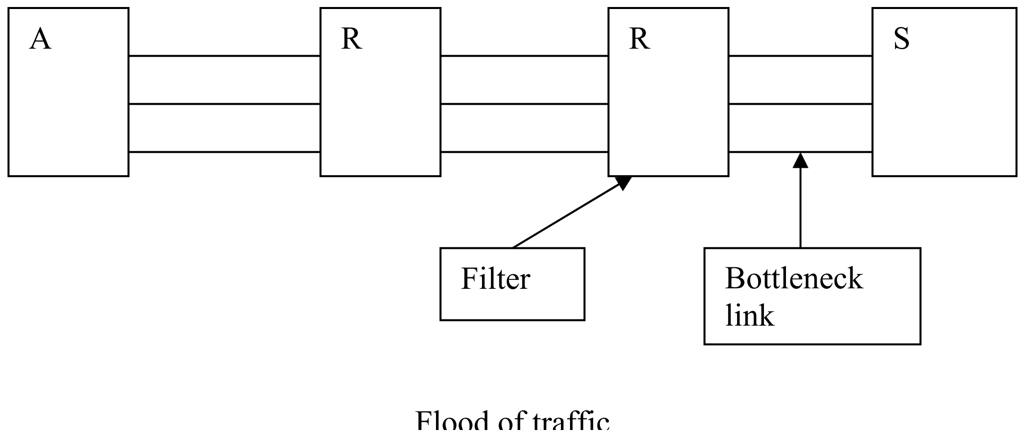
1) give the whole set of bits for solving (too long for the client)
2) give L bits only i.e., a small X (too small and all possible X and Y pairs can be stored an a lookup table)
3) keep the client puzzles mandatory all the time (too expensive for the clients so only turn it on when under attack)

Now if client computation cost is 1000ms and server computation cost is say still 10ms, then to overload the server

- you must force it to perform 100 signs/sec
- attacker must solve 100 puzzles/sec
- attack can be from 100 zombies

In-network solutions

IP traceback



Without faking source address we have

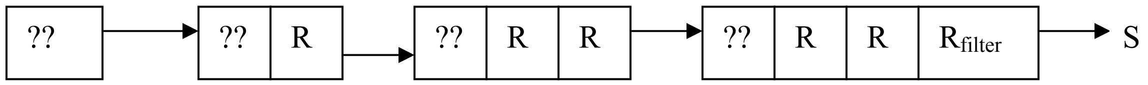
- A-R – not out problem
- R-R – links very high speed
- R-S – overloaded links

where A is attacker, Rs are routers and S is the server.

If A is not faking the address, then if the filter is in R, the traffic will stop at the last R.

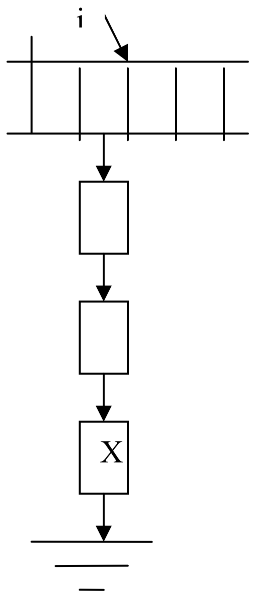
If A is faking the source address then R know what IPs can come, then R stops them (egress filtering) But there is little or no incentive for egress filtering to be implemented at any router since it has to maintain a list of possible addresses

Packet will have the list of routers they are traveling through and hence can be verified for source



Hence an attack can be traced for DOS.

Hash table attack



Structure of a hash table

```
Insert (X)
{
        i=h(X);
        LinkedListAdd (T[i], X);
}
```

If T contains n cells and we insert n objects the insertion/ lookup costs are O (1) with high probability – this is the hash table theorem

If h is fixed and known to the attacker, then he can force a lot of collision and send lookups for the last inserted X continuously to overload the server. Solution would be to use random h at some point like startup of the program from a family of hash functions H.

A modified theorem for hash functions is - if T contains n cells and insert n objects and $h \varepsilon H$ over R and H is 2-universal then insertion/ lookup costs are O(1) with high probability. A family of hash function H is 2-universal if given $(X, h_i(X))$ for some X, you have no information about which hash function h_i is. E.g. $h(X) = aX+b$

Trusted Computing

May 2, 2006 mnair@cs.sunysb.edu

We design a system to deliver content from a movie server over the internet. Figure 1 shows the schematic structure of such a system. The security goals of a such a system are:

1. Client must pay for download.
2. Can view only for 3 days.
3. Cannot share the movies.
4. Must watch all the advertisements.
5. Integrity of the movie (No Censorship).

In current systems with the root privileges, you can perform any action that you want; thus you can obtain the binary. The twist is that you donot trust the owner/root. We look at three solutions to this problem:

· Closed Platforms
· Trusted Computing Platforms
· Terra - Trusted Virtual Machine Monitor

Closed Platforms

A closed platform is usually a system with limited computing capability. It caters to only a small, specific function. The characteristic feature(closed) of these systems is that their structure and design is known only to the system designers. Examples of such computing devices include cellphones and DVD players. Cellphones are difficult to program. The designer donot intend it to be used for any purpose other than the one it is expressly programmed for in the factory. All DVDs are encrypted with a key. There exists a chip

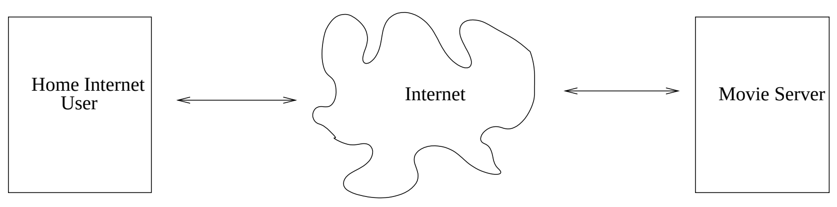


Figure 1: Schematic Design of Internet Movie Distribution Scheme

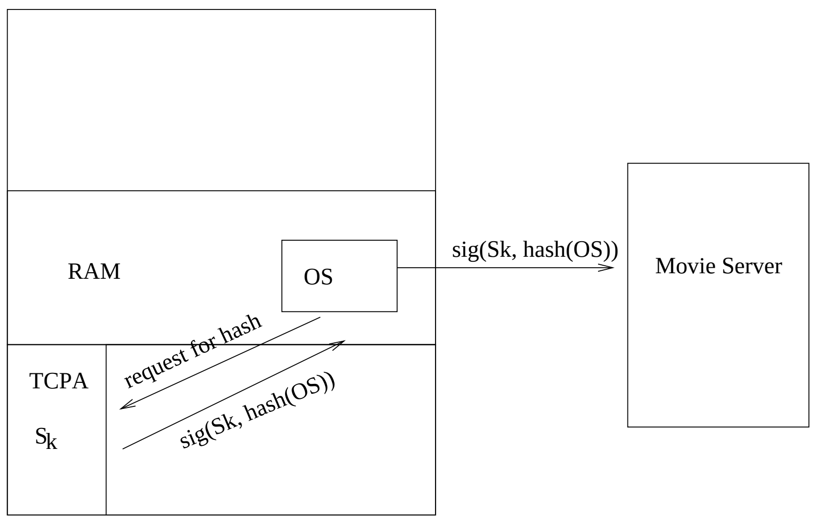


Figure 2: Design of Trusted Computing Platforms

for cryptography in the player which has the secret key. When a company designs a DVD player it is contractually bound to design it in a manner that does not compromise this secret key.

With closed platform we get very good security. Unfortunately we get it at the expense of flexibility and extensibility.

Trusted Computing Platform(TCP)

This comes under various names, TCPA(Trusted Computing Platform Alliance), Palladium, NGSCB(Next Generation Secure Computing Base). We want a small piece of tamper resistant hardware, which even an attacker(?) with a high-level of access cannot modify. What we require is *remote attestation*, i.e. we want to be able to securely identify the software running on a remote computer. If we trust this software then we can derive trust relations with other software running on the system and thus we will be able to achieve our goal.

We will thus require a hash of the operating system running on the system. However a system could lie about its hash – a wayward system could compute the hash from an image of another OS which exists on disk and send it. Thus we will require the hash to be signed by a piece of hardware we trust; the trusted computing platform(TCP). TCPs should be tamper resistant and should sign only the currently running OS. A possible design for TCPs are elucidated in figure 2.

TCP proves to the remote party that the currently running client is "WinXP". Then the server trust WinXP to enforce the security policy. Thus the client enforces the security policy. This is the essence of remote attestation.

Criticisms

· **OS Fingerprinting** - Establishes accurately the OS running on the client side.
· **Vendor Locking** - Inability to change h/w or s/w vendors.
· **OS Inflexibility** - Ability to use only trusted operating systems.

Terra

The ides is to run a trusted Virtual Machine Monitor under the operating system. A Virtual machine(VM)is a number of different identical execution environments coexisting on a single computer, each of which exactly emulates the host computer. A Virtual Machine Monitor is that which manages these virtual machines. The structure of Terra is shown in figure 3.

Virtualization is a well studied technology. It is possible to get good performance and strong isolation (security) from VMMs. Here too we have a TCP which signs the TVMM images. TVMM in turn signs VM images. TVMM stores the signature of each VM securely on disk. If the signature of the currently running VM is that which is stored on disk, then the TVMM is a able to verify that the VM has not be tampered with and thus can be trusted.

Trust is managed in the following manner:

1. TCPA to verify TVMM.
2. TVMM to verify the Virtual Machines and the disk images.

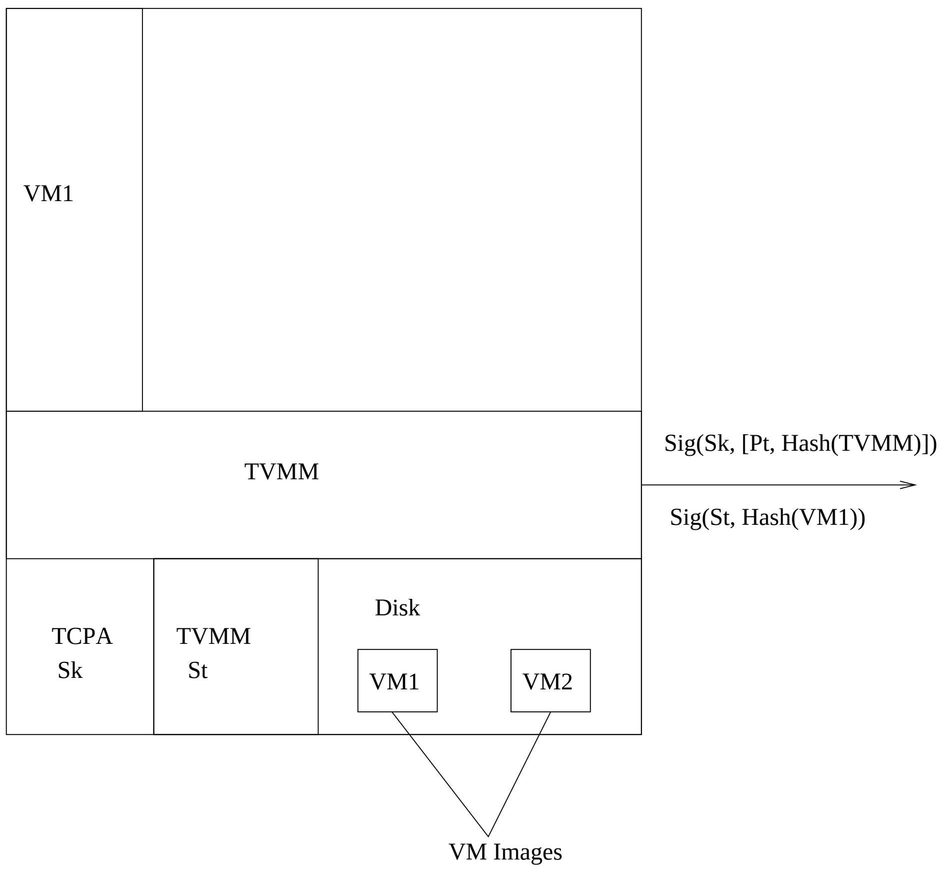


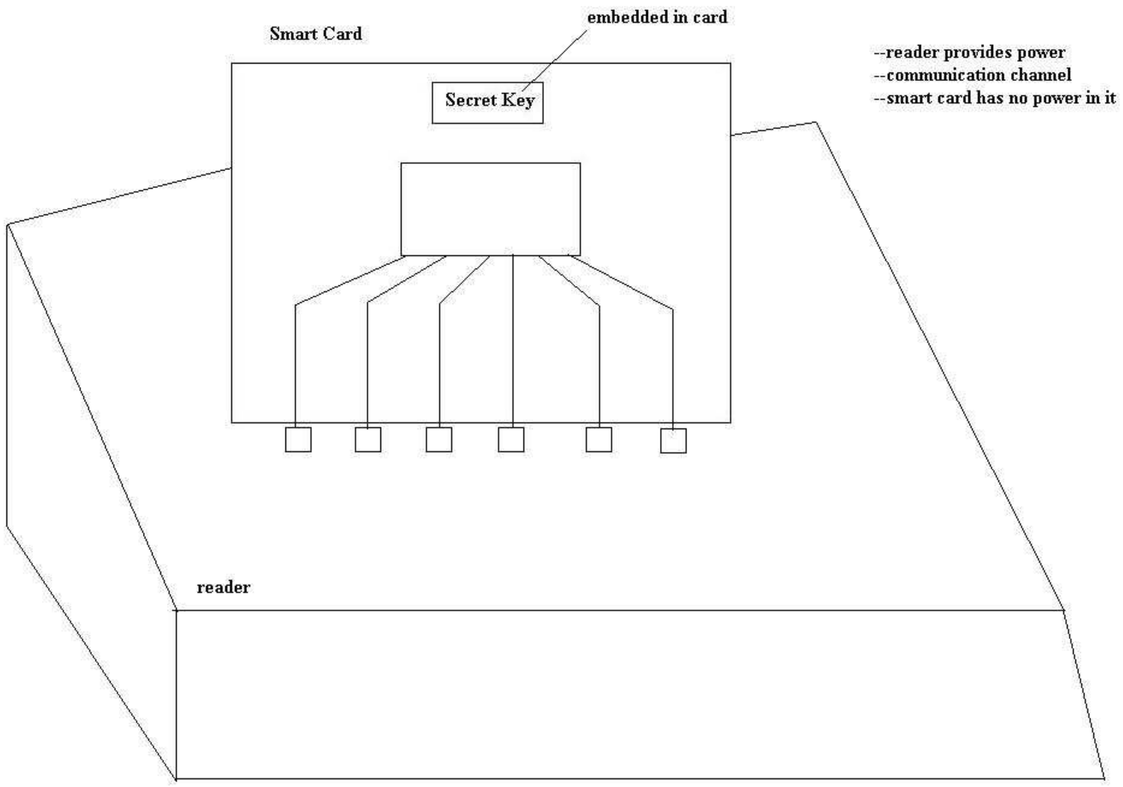
Figure 3: Design of Terra

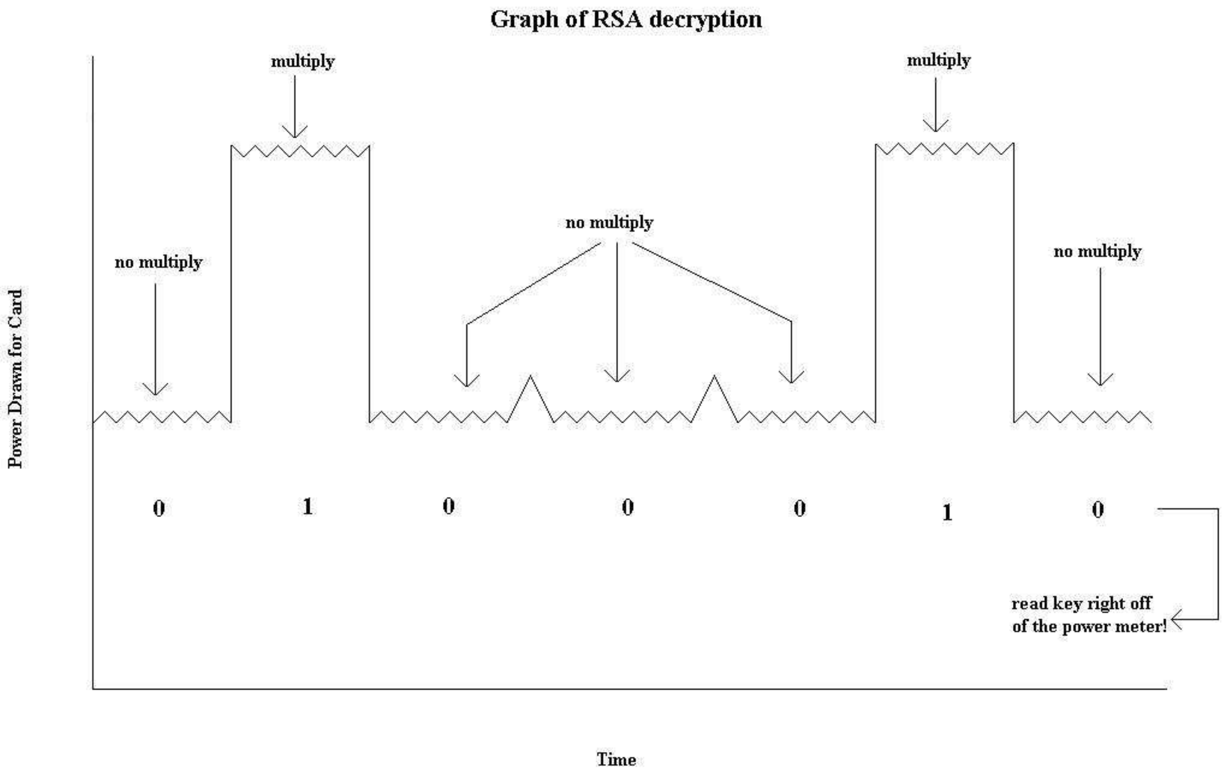
CSE 509 Notes
5/4/2006

Side Channel Attacks

Side Channel – other side communication channels that exist other than the main channel (examples: electricity, amount of time of a connection, etc.)

-Remote Timing Attacks – originally for smart cards (replaces credit cards with a microchip with wires going into the edge of the card, connected to a reader box, then a wire from the reader box to the credit card company).





Threat Model – Nowadays with a credit card, it is used at a merchant, the account number (data) is read from the card, then the account is charged (suppose a waiter at a restaurant can use the card number for identity theft). Assume the merchant is bad, but we want to purchase from him, but not allow them to make any other purchases. This is a Power Analysis Attack (done a lot with remote timing attacks).

When a smart card is connected to a reader, it gives the exact amount of time it takes to make a decryption.

-Local Timing Attack – a type of Remote Timing Attack in which a custom reader is created to do thousands of encryptions/decryptions instead of only one. Create pairs (C1, t1), (C2, t2), (C3,t3), …, (Cn, tn), figure out how long it takes to compute each decryption pair, then <u>guess</u> each bit independently.

<u>Attack:</u>

```
For i = 0 to L
    Guess bit i of the key is 0.
    Divide samples into slow and fast groups.
    If timing differs
        Bit i = 0
    Else
        Bit i = 1
```

Example:

(C1, t1) is slow
(C2, t2) is slow
(C3, t3) is fast
…
(Cn, tn) is slow

If key bits are guessed correctly and divided correctly, then the key is obtained. Otherwise, the average timing of each decryption is about the same. This can be done within thousands of messages/transactions.

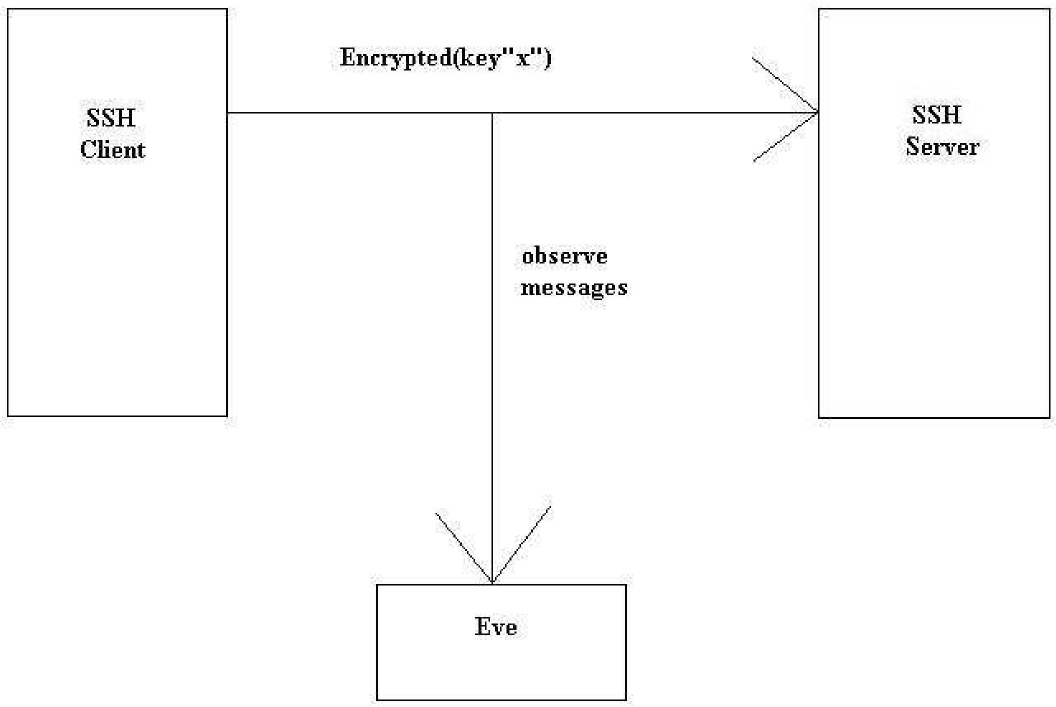
Assuming timing measures are good, and exponentiation has certain properties, if SSL does the same type of algorithm with timing, it can be broken too, but people didn't consider this possibility. However, SSL uses other algorithms besides simple multiplication and exponentiation (as opposed to the smart card).

SSL has broken simply by doing a handshake and watching the timing of RSA decryptions. It was also done behind wiring and an entire network! It took roughly between 400 and 800 messages per bit of the key to be guessed, and since the key was about a thousand bits, it took roughly one million messages to accomplish this!

This type of remote timing attack extracted the key from a <u>real</u> SSL server with one million messages. Thus, the attacker was now able to impersonate the server with the SSL key!

<u>Another Side Channel Attack</u>

- Keystroke Timing
    - From Packet Timings,
        - One can infer keystroke timing
        - There is bias in interkey typing (consider how you type letters on the keyboard using your hand, and the different lengths in time when typing a pair of letters such as x and y).
        - Thus, one can infer what a person is typing from interkey timings!

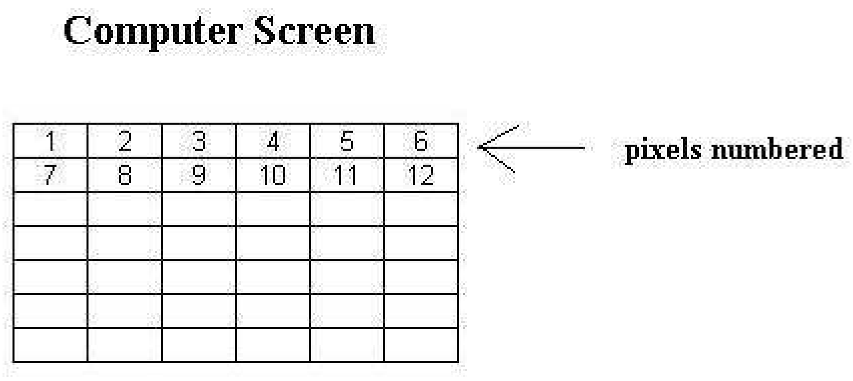


Model for obtaining keyboard information

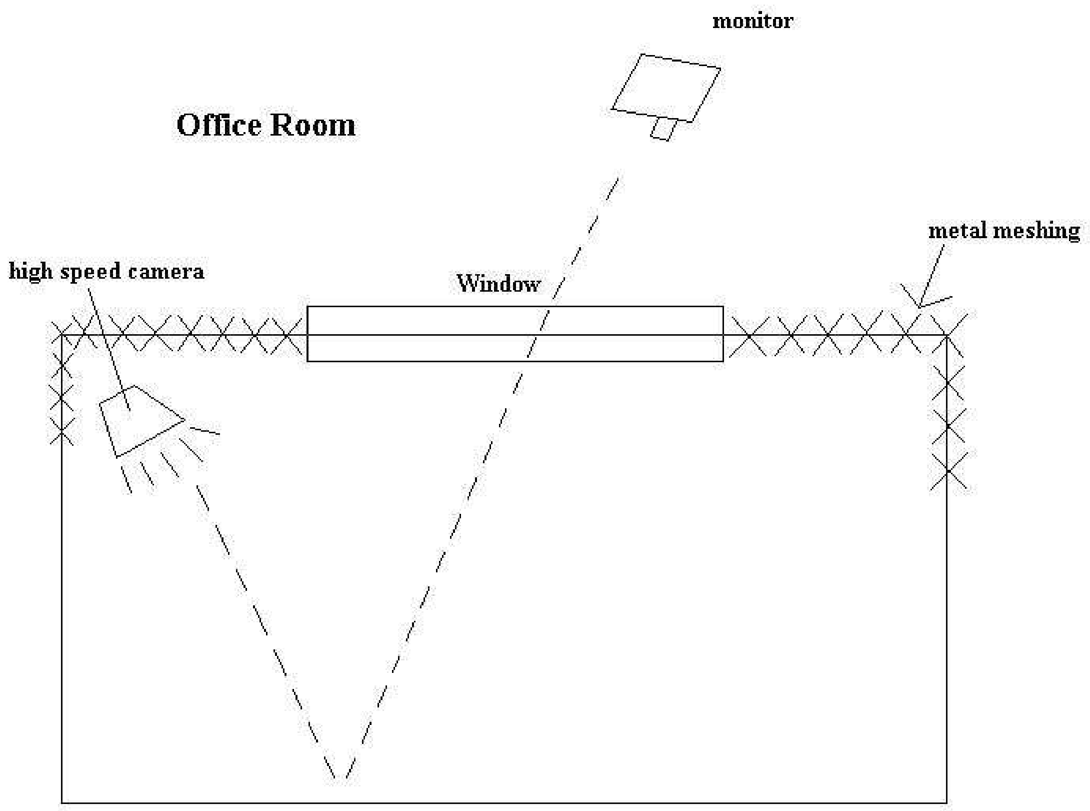
1. Develop Model of Interkey Timing
   Probability[KeyPair = "xy" given the interkey timing is 37 ms] (conditional probability)
2. Viterbi's Algorithm: Takes observed timings and outputs the most likely key sequence!

If there is a microphone in the room, one not only knows the timing, but even the frequency/sound of each unique key. Then, figure out each unique keyboard sound (based on the type of keyboard), and infer what a user is typing!

There is another type of side attack, but the technology that this is based on is a CRT monitor: an Optical-Domain Tempest Attack



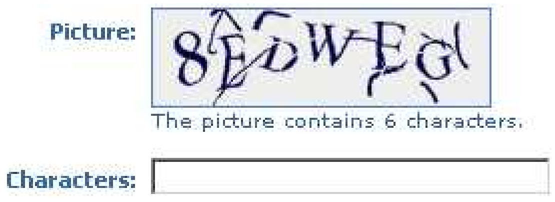
One pixel lights up at a time if it needs to be on or not. If so, an electron is shot at the pixel to turn it on, but this is done so fast for the entire screen, the human eye only sees the end result. However, place a very high speed camera at the screen and one can obtain the information. This is particularly useful if the monitor is facing a window, and an attacker is not at the same desk or even outside in a different building! Thus, agencies such as NSA force monitors to point away from windows.



Shine a high speed camera at the wall, and when the monitor shines each pixel, it is reflected on the wall and caught by the camera.

LCD monitors do not allow this! Another option that agencies use is a Ferraday Cage, which blocks electromagnetic rays by using metal meshing around the walls and windows of high security places (see the diagram with the office).

CAPTCHA’s



These are used to make sure that humans were at the keyboard and not a script (example: Yahoo’s free email accounts).

Yahoo needed

-puzzle

-humans can pass with extremely high probability (but later on, this was a problem for illiterate or blind people, foreigners using different alphabets and characters, and even kids).

-computers cannot pass (only passed with a very low probability)

-computer can create the puzzle

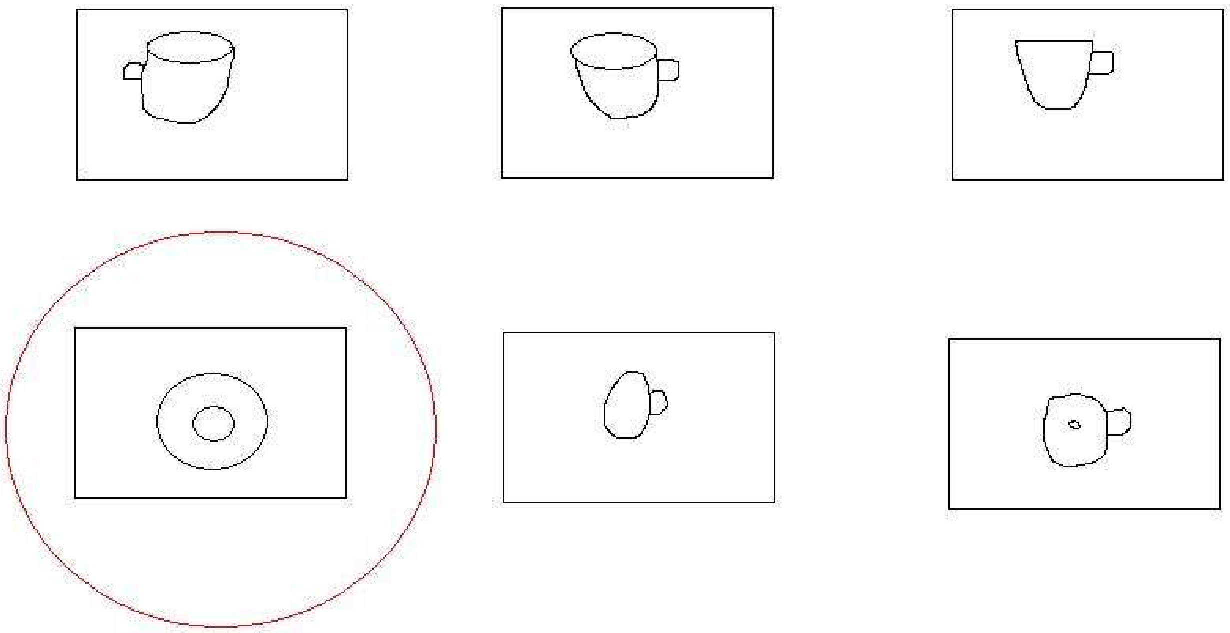
-computer can grade the puzzle (the computer picks a word from the dictionary or a set of characters, then puts them into a image distorter, then checks if the user’s input matches the word used to create it).

Attempts to break this lead to:

Fundamental AI problems

-OCR (attacks: better OCR algorithms than used in the past)

-Image recognition (example: two words are picked from the dictionary, use Google image searches on the two words, then picks 5 images of one word and one image from the other word. Pick the odd picture).



<u>Proxy Attack</u>

One can run their own website that everyone wants to visit. Then, require visitors to the website to solve a CAPTCHA. This CAPTCHA is obtained by running a script against Yahoo, then the CAPTCHA is sent from Yahoo to the website, let the <u>visitors</u> solve the CAPTCHAs, and then send their exact response to Yahoo.

The CAPTCHA goal to break this: Humans are the cheapest solution.