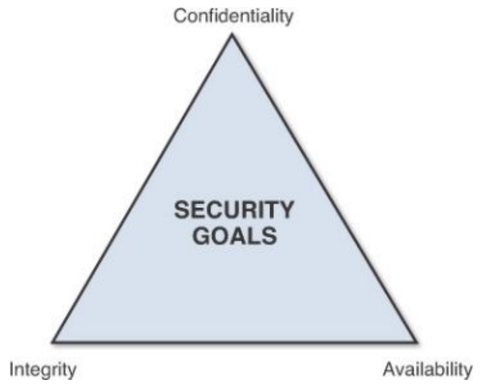
Designing for Security
(Access Control)

Chester Rebeiro

Indian Institute of Technology Madras

Security Goals

Any security system must address the following goals

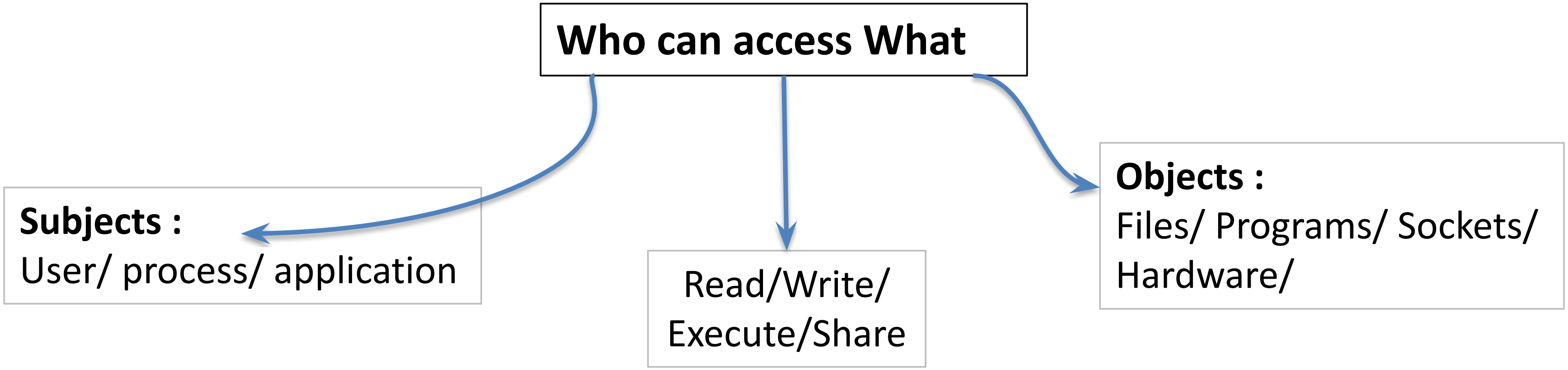


- Confidentiality
  keep data secret except to authorized users
- Integrity
  - prevent unauthorized users from making modifications
  - Prevent authorized users from making improper modifications
- Availability of data to unauthorized users
  - Handle Denial of Service, loss due to natural disasters, equipment failure

eg. Moodle, facebook

Access Control

(the tao of achieving confidentiality and integrity)



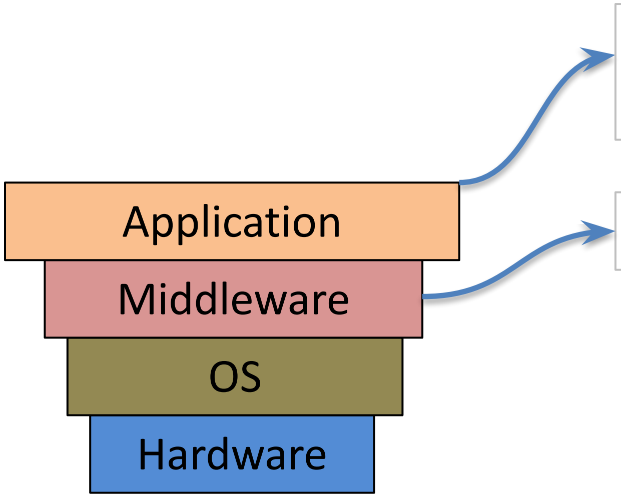
Who can access What

Subjects :
User/ process/ application

Read/Write/
Execute/Share

Objects :
Files/ Programs/ Sockets/
Hardware/

Access Control
(number of levels)



Elaborate and complex. Many people may be involved
Multiple roles. Hundreds of transactions feasible

Eg. DBMS. Who gets to access what fields in the DB

Moving from Hardware to Application

- More aspects to control
  - More subjects and objects involved
  - Inter-relationship becomes increasingly difficult
- Complexity increases
- Reliability Decreases
  - More prone to loopholes that can be exploited

Hardware Access Control

- **Policies**
  - Must protect OS from applications
  - Must protect applications from others
  - Must prevent one application hogging the system

  (first two ensure confidentiality and integrity, the third ensures availability)

- **Mechanisms**
  - Paging unit
  - Privilege rings
  - Interrupts

Access Control at OS Level

Policies

- Only authenticated users should be able to use the system
- One user's files should be protected from other users
  (not present in older versions of Windows)
- A Process should be protected from others
- Fair allocation of resources (CPU, disk, RAM, network) without starvation

Mechanisms

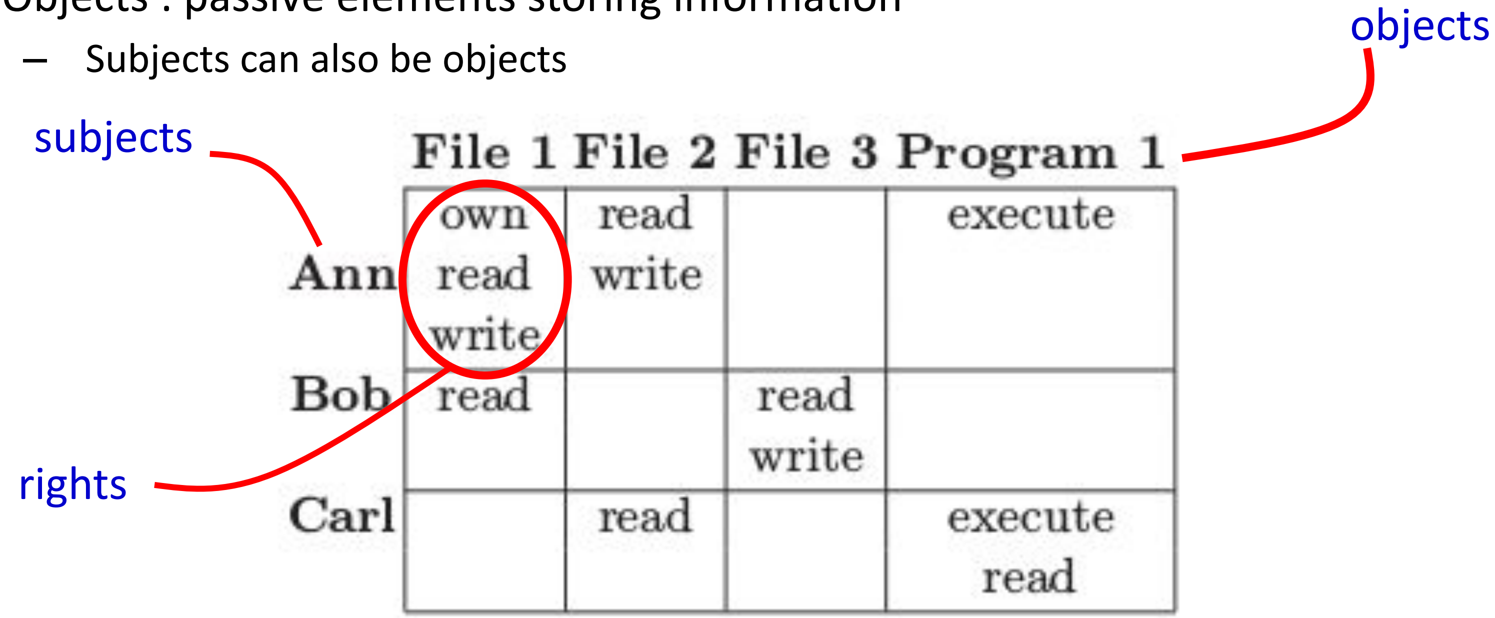
- User authentication
- **Access Control Mechanisms for Files (and other objects)**
- For process protection leverage hardware features (paging etc.)
- Scheduling, deadlock detection / prevention to prevent starvation

Access Control for Objects in the OS

- Discretionary (DAC)
  - Access based on
    - Identity of requestor
    - Access rules state what requestors are (or are not) allowed to do
  - Privileges granted or revoked by an administrator
  - Users can pass on their privileges to other users
  - The earliest form called Access Matrix Model

Access Matrix Model

- By Butler Lampson, 1971 (Earliest Form)
- Subjects : active elements requesting information
- Objects : passive elements storing information
  - Subjects can also be objects



objects

subjects

rights

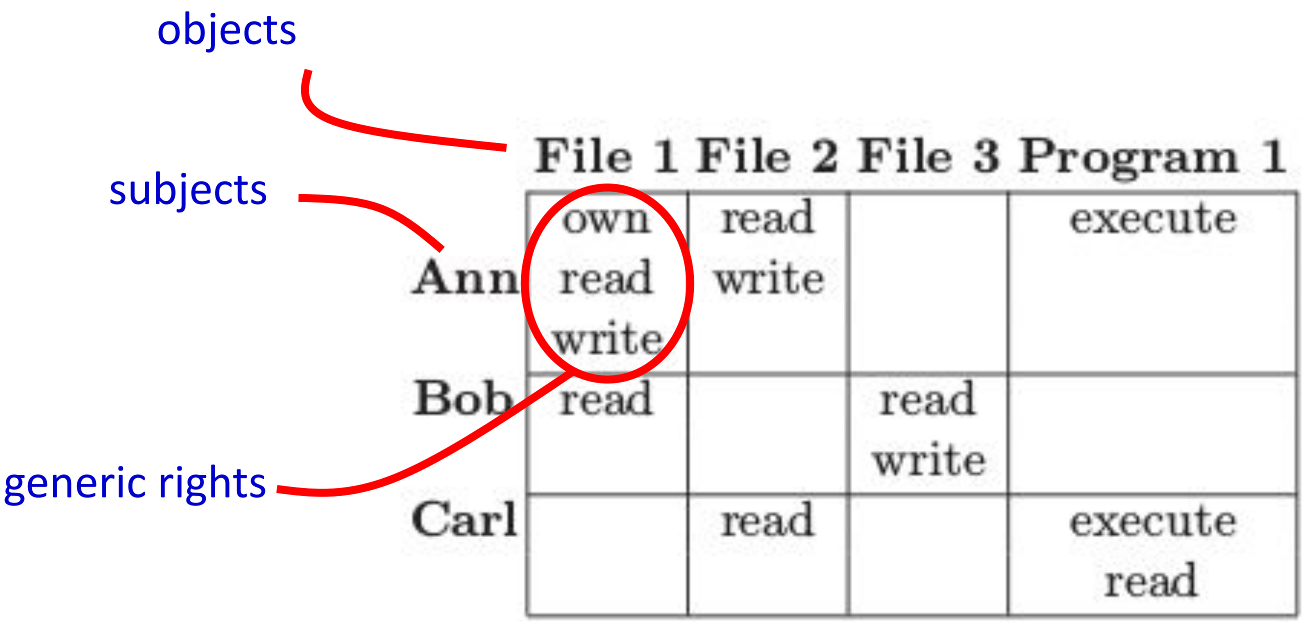
	File 1	File 2	File 3	Program 1
Ann	own read write	read write		execute
Bob	read		read write	
Carl		read		execute read

Other actions : ownership (property of objects by a subject),
control (father-children relationships between processes)

Butler Lampson, “Protection”, 1971

A Formal Representation of Access Matrix

- Define an access matrix : $\boxed{A[X_{s_i}, X_{o_j}]}$
- Protection System consists of
  - **Generic rights :** $\boxed{R = \{r_1, r_2, \cdots, r_k\}}$ **thus** $A[X_{s_i}, X_{o_j}] \subseteq R$
  - **Primitive Operations** $\boxed{O = \{op_1, op_2, \cdots, op_n\}}$



objects

subjects

generic rights

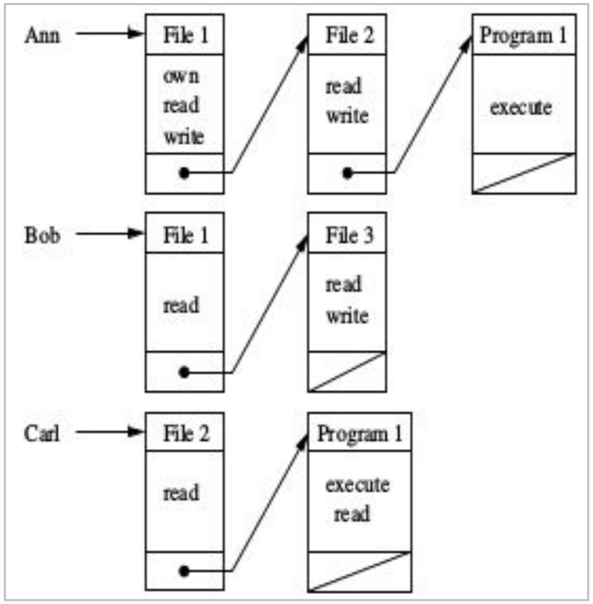
	File 1	File 2	File 3	Program 1
Ann	own read write	read write		execute
Bob	read		read write	
Carl		read		execute read

enter r into $A[X_{s_i}, X_{o_j}]$
delete r from $A[X_{s_i}, X_{o_j}]$
create subject X_s
create object X_o
destroy subject X_s
destroy object X_o

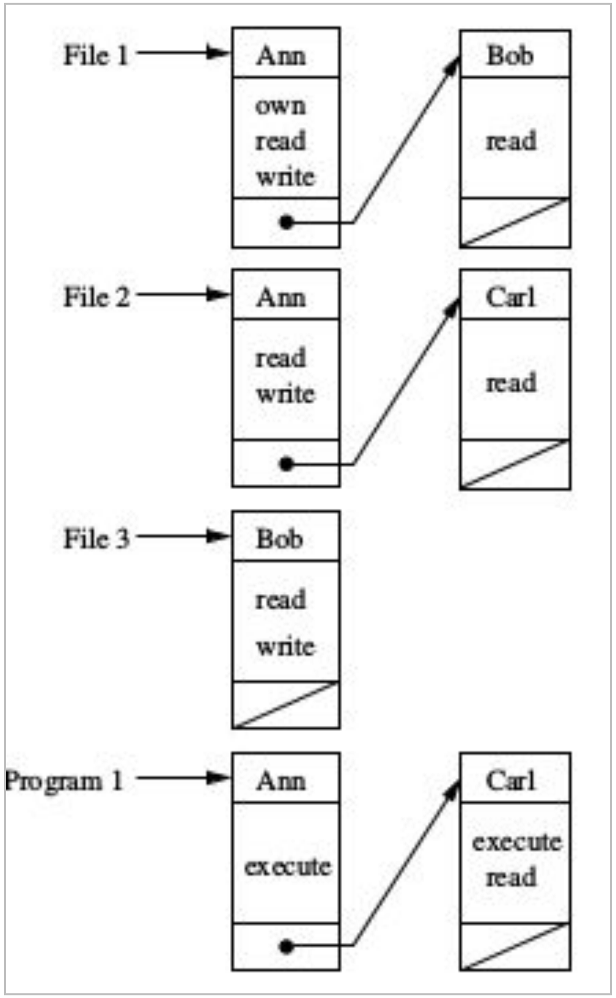
Michael A. Harrison, Walter L. Ruzzo, Jeffrey D. Ullman, Protection in Operating Systems, 1974

Implementation Aspects

Capabilities



Access Control List



Capabilities : ticket
ACL : My name is in the list

Railway Reservation

Capability vs ACL

- Delegation

  CAP: easily achieved

  For example “Ann” can create a certificate stating that she delegates to “Ted” all her activities from 4:00PM to 10:00PM

  ACL: The owner of the file should add permissions to ensure delegation

- Revocation

  ACL: Easily done, parse list for file, remove user / group from list

  CAP: Get capability back from process
  If one capability is used for multiple files, then revoke all or nothing

Unix Security Mechanism

- **Subject:** process
- **Objects:** files, directories, sockets, process, process memory, file descriptors

- Each process is associated with a user ID (32 bit integer) and group ID (32 bit integer)
- The privileges of a process depends on the user ID and group ID

File Operations in Unix

Operations for a file

- Create
- Read
- Write
- Execute (does this imply read?)
- Ownership
- Change permissions

Operations for a directory

- Create
- Unlink / link
- Rename a file
- lookup

Permissions for files and directories

In inode :
uid, gid

	R	W	X
Owner	1	1	0
Group	1	0	0
Other	1	0	0

Change permissions by owner (same uid as the file)

For directories almost similar: linking / unlinking write permissions

X permission on a directory implies look up. You can look up a name but not read the contents of the directory

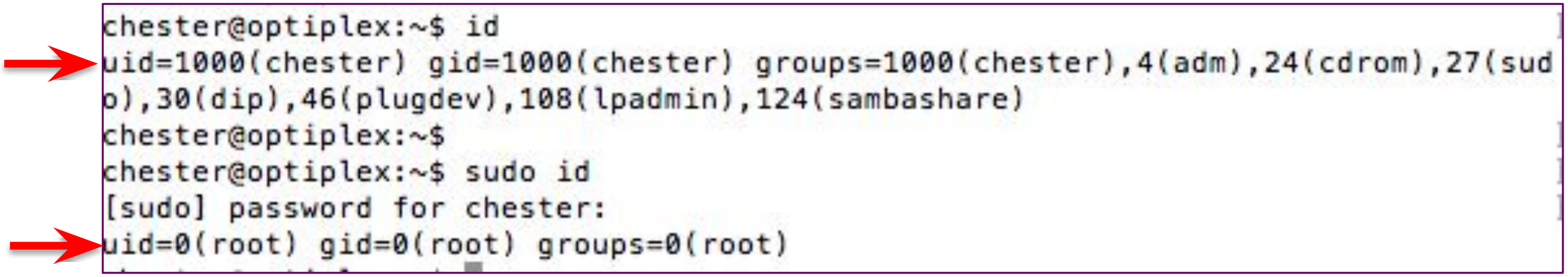
Additionally bits are present to specify type of file (like directory, symbolic link, etc.)

User IDs

- UID = 0 is root permissions
- setuid(user ID) □set the user id of a process. Can be executed only by processes with UID = 0
- setgid(group iD) □ set the group id of a process
- Login process
  - At the time of login, the login process runs with uid=0
  - If user name and password is verified,
    - Use uid stored in /etc/passwd file to invoke setuid()
    - Invoke shell with the user's process ID
- setuid bit in inode
  - Allows a program to execute with the privileges of the owner of the file.

sudo / su

- used to elevate privileges
  - If permitted, switches uid of a process to 0 temporarily
  - Remove variables that control dynamic linking
  - Ensure that timestamp directories (/var/lib/sudo) are only writeable by root



```
chester@optiplex:~$ id
uid=1000(chester) gid=1000(chester) groups=1000(chester),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),108(lpadmin),124(sambashare)
chester@optiplex:~$
chester@optiplex:~$ sudo id
[sudo] password for chester:
uid=0(root) gid=0(root) groups=0(root)
```

File Descriptors

- Represents an open file
- Two ways of obtaining a file descriptor
  - Open a file
  - Get it from another process
    - for example a parent process
    - Through shared memory or sockets
- If you have a file descriptor, no more explicit checks

Processes

- Operations
  - Create
  - kill
  - Debug (ptrace system call that allows one process to observe the control the other)

- Permissions
  - Child process gets the same uid and gid as the parent
  - ptrace can debug other processes with the same uid

Network Permissions in Unix

- Operations
  - Connect
  - Listening
  - Send/Receive data
- Permissions
  - Not related to UIDs. Any one can connect to a machine
  - Any process can listen to ports > 1024
  - If you have a descriptor for a socket, then you can send/receive data without further permissions

Problems with the Unix Access Control

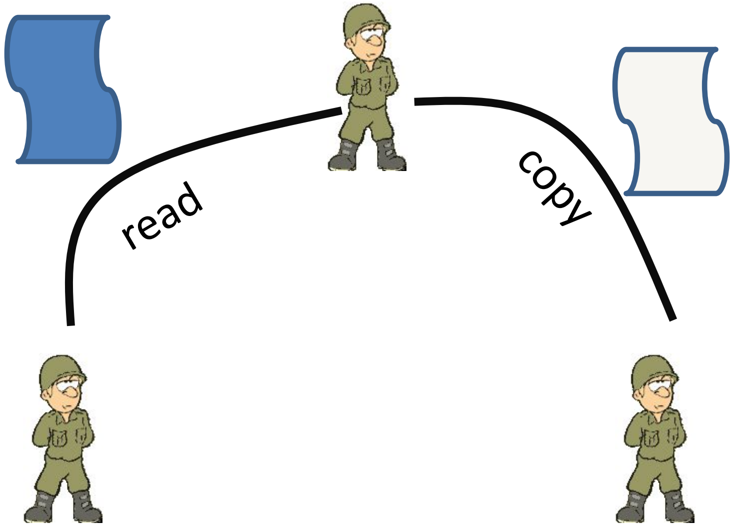
- **Root can do anything (has complete access)**
  - Can delete / modify files
    (FreeBSD, OSX, prevent this by having flags called append-only, undeletable, system □ preventing even the root to delete)
  - Problem comes when (a) the system administrator is untrustable
    (b) if root login is compromised
- **Permissions based on uid are coarse-grained**
  - a user cannot easily defend himself against allegations
  - Cannot obtain more intricate access control such as
    “X user can run program Y to write to file Z”
  - Only one user and one group can be specified for a file.

Vulnerabilities in Discretionary Policies

- Discretionary policies only authenticate a user
- Once authenticated, the user can do anything
- Subjected to Trojan Horse attacks
  - A Trojan horse can inherit all the user's privileges
  - Why?
    - A trojan horse process started by a user sends requests to OS on the user's behalf

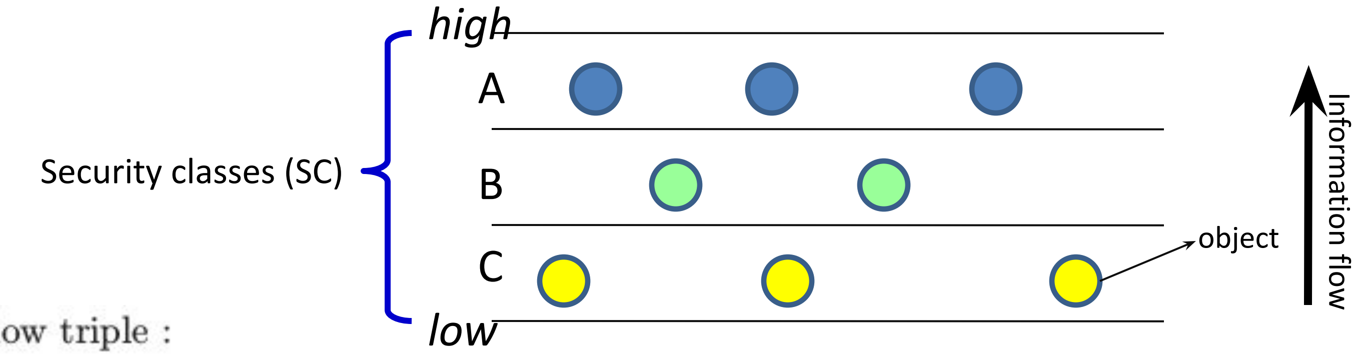
Drawback of Discretionary Policies

- It is not concerned with information flow
  - Anyone with access can propagate information
- Information flow policies
  - Restrict how information flows between subjects and objects



Information Flow Policies

- Every object in the system assigned to a security class (SC)



- Information flow triple :

  $\langle SC, \rightarrow, \oplus \rangle$

  $\rightarrow$ is the can flow relation

  - $B \rightarrow A$: Information from B can flow to A
  - $C \rightarrow B \rightarrow A$: Information flow
  - $C \leq B \leq A$: Dominance relation

  $\oplus$ is the join relation

  - defines how to label information obtained by combining information from two classes
  - $\oplus : SC \times SC \rightarrow SC$.

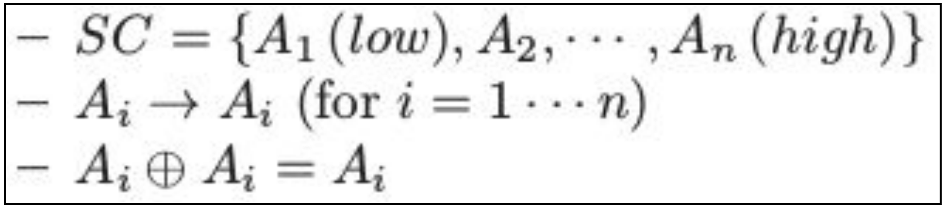
  SC, $\rightarrow$, and $\oplus$ are fixed and do not change with time.

  The SC of an object may vary with time

Ravi Sandhu, *Lattice Based Access Control Models*, 1993

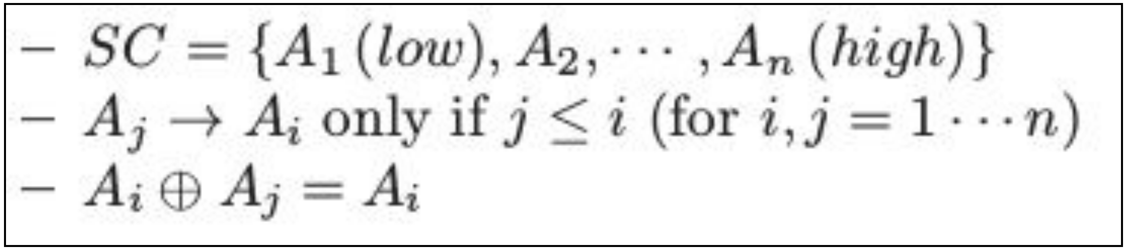
Examples

- Trivial case (also the most secure)
  - No information flow between classes



- $SC = \{A_1\,(low), A_2, \cdots, A_n\,(high)\}$
- $A_i \rightarrow A_i$ (for $i = 1 \cdots n$)
- $A_i \oplus A_i = A_i$

- Low to High flows only



- $SC = \{A_1\,(low), A_2, \cdots, A_n\,(high)\}$
- $A_j \rightarrow A_i$ only if $j \leq i$ (for $i, j = 1 \cdots n$)
- $A_i \oplus A_j = A_i$

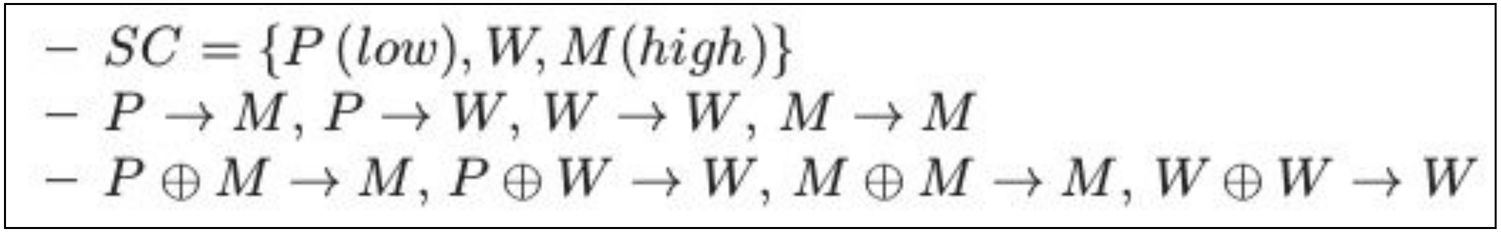
Exercises

- A company has the following security policy
  - A document made by a manager can be read by other managers but no workers
  - A document made by a worker can be read by other workers but no managers
  - Public documents can be read by both Managers and Workers

- What are the security classes?
- What is the flow operator?
- What is the join operator?

Exercises

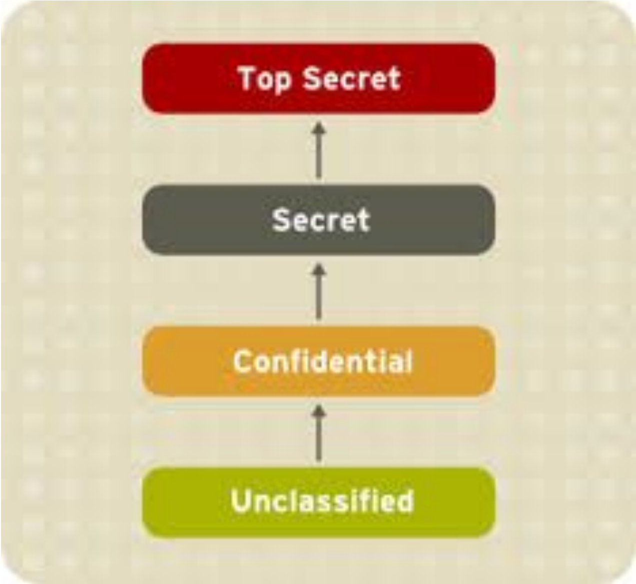
- A company has the following security policy
  - A document made by a manager can be read by other managers but no workers
  - A document made by a worker can be read by other workers but no managers
  - Public documents can be read by both Managers and Workers



- $SC = \{P\,(low), W, M\,(high)\}$
- $P \rightarrow M$, $P \rightarrow W$, $W \rightarrow W$, $M \rightarrow M$
- $P \oplus M \rightarrow M$, $P \oplus W \rightarrow W$, $M \oplus M \rightarrow M$, $W \oplus W \rightarrow W$

Mandatory Access Control

- Most common form is multilevel security (MLS) policy
  - Access Class
    - Objects need a **classification level**
    - Subjects needed a **clearance level**
  - A subject with *X* clearance can access all objects in X and below X but not vice-versa
  - Information only flows upwards and cannot flow downwards

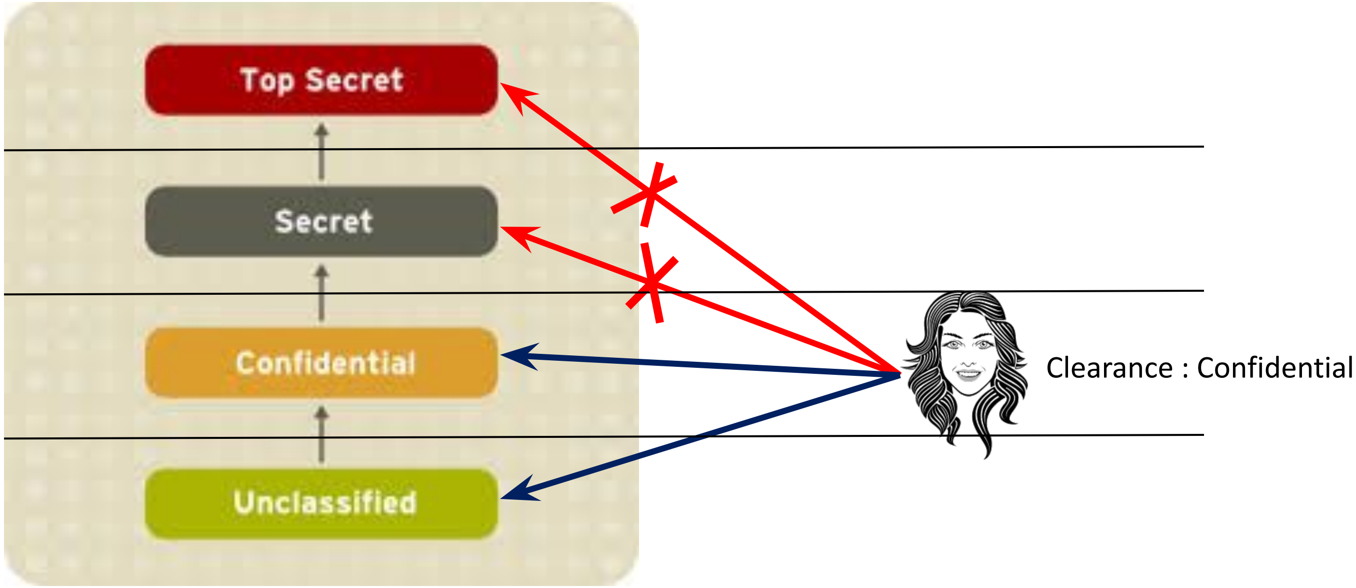


Bell-LaPadula Model

- Developed in 1974
- Objective : Ensure that information does not flow to those not cleared for that level
- Formal model for access control
  - allows formally prove security
- Four access modes:
  - read, write, append, execute
- Three properties (MAC rules)
  - No read up (simple security property (ss-property))
  - No write down (*-property)
  - ds property : discretionary security property (every access must be allowed by the access matrix)

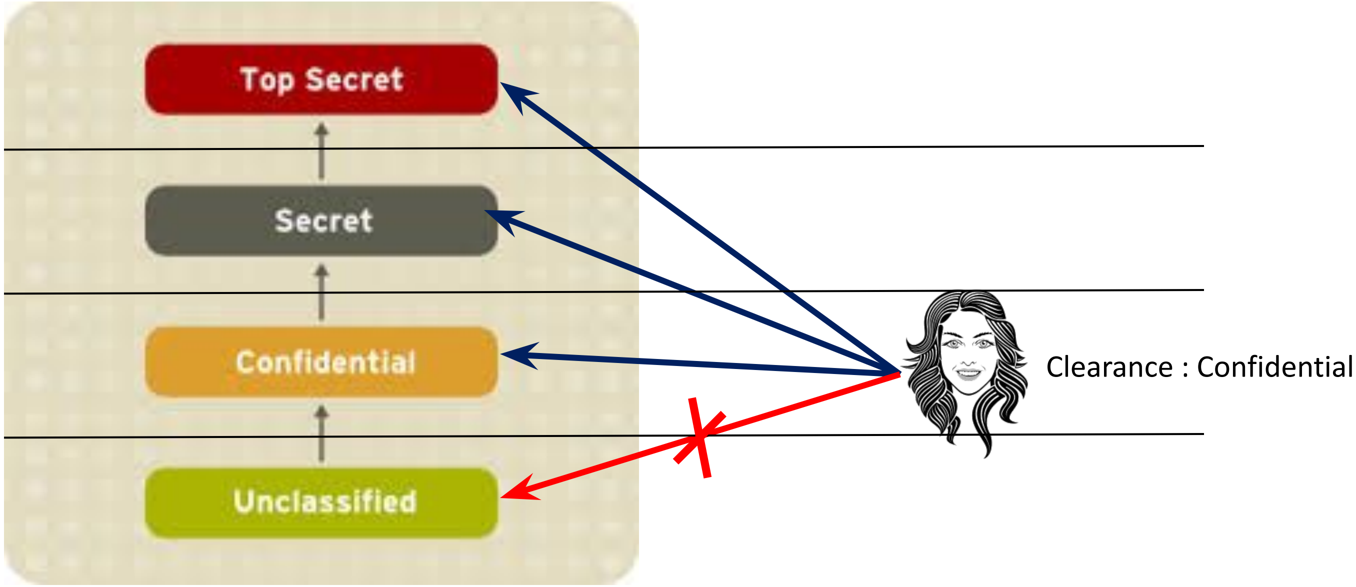
D. E. Bell and L. J. LaPadula, *Secure Computer System: Unified*

No read up



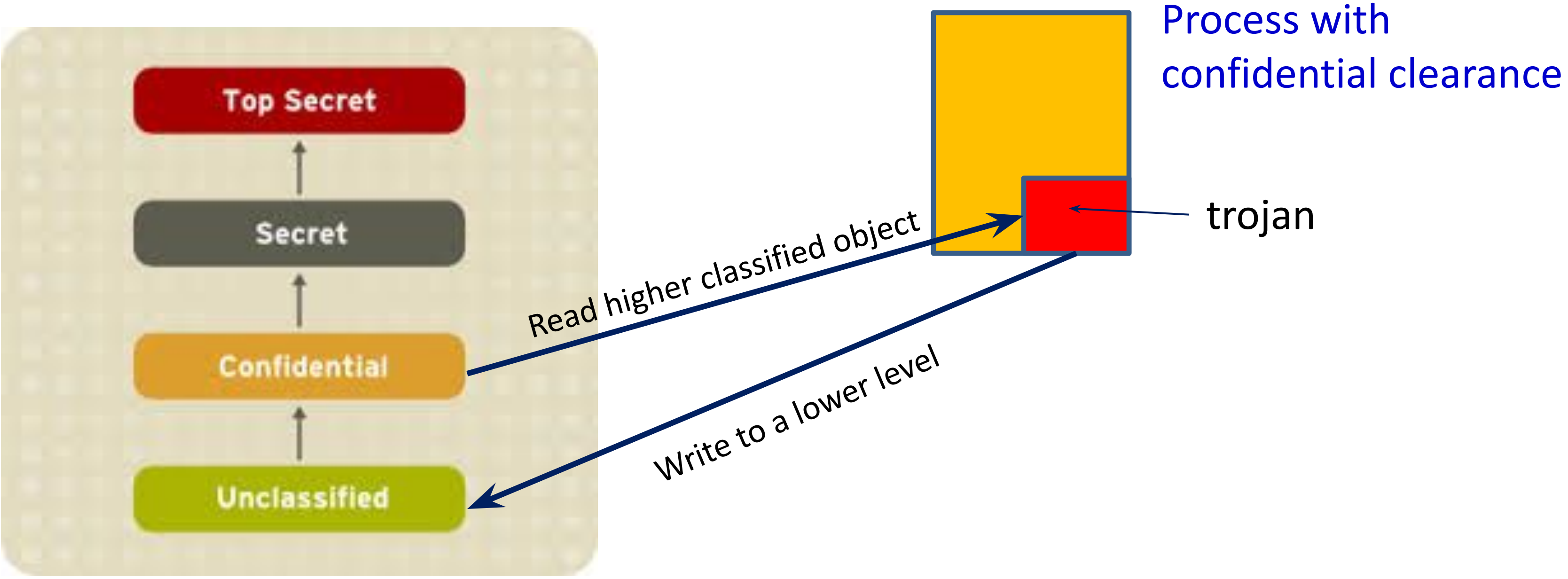
- Can only read confidential and unclassified files

No Write Down



- Cannot write into an unclassfied object

Why No Write Down?



Top Secret

Secret

Confidential

Unclassified

Read higher classified object

Write to a lower level

Process with
confidential clearance

trojan

- A process inflected with a trojan, could read confidential data and write it down to unclassified
- We trust users but not subjects (like programs and processes)

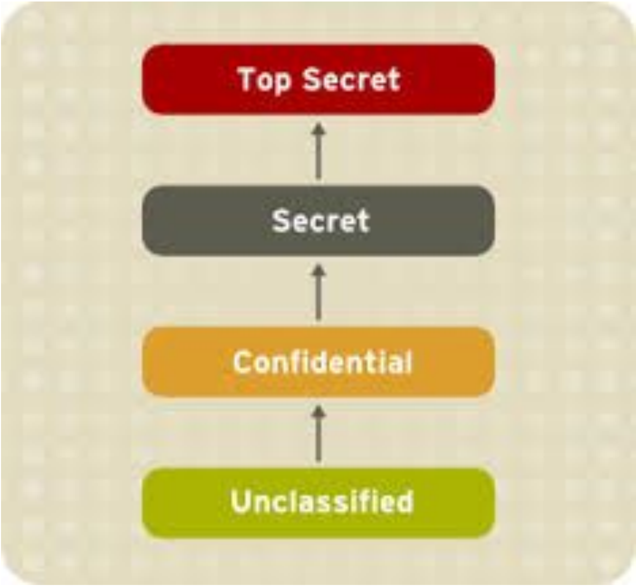
ds-property

- Discretionary Access Control
  - An individual may grant access to a document he/she owns to another individual.
  - However the MAC rules must be met

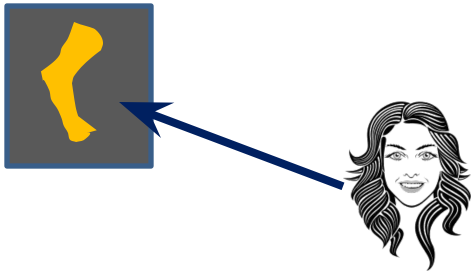
MAC rules over rides any discretionary access control. A user cannot give away data to unauthorized persons.

Limitations of BLP

- Write up is possible with BLP
- Does not address Integrity Issues



file with classification *secret*



Clearance : Confidential

User with clearance can modify a secret document
BLP only deals with confidentiality. Does not take care of integrity.

Limitation of BLP (changing levels)

- Suppose someone changes an object labeled ***top secret*** to ***unclassified***.
  - breach of confidentiality
  - Will BLP detect this breach?
- Suppose someone moves from clearance level top secret to unclassified
  - Will BLP detect this breach?

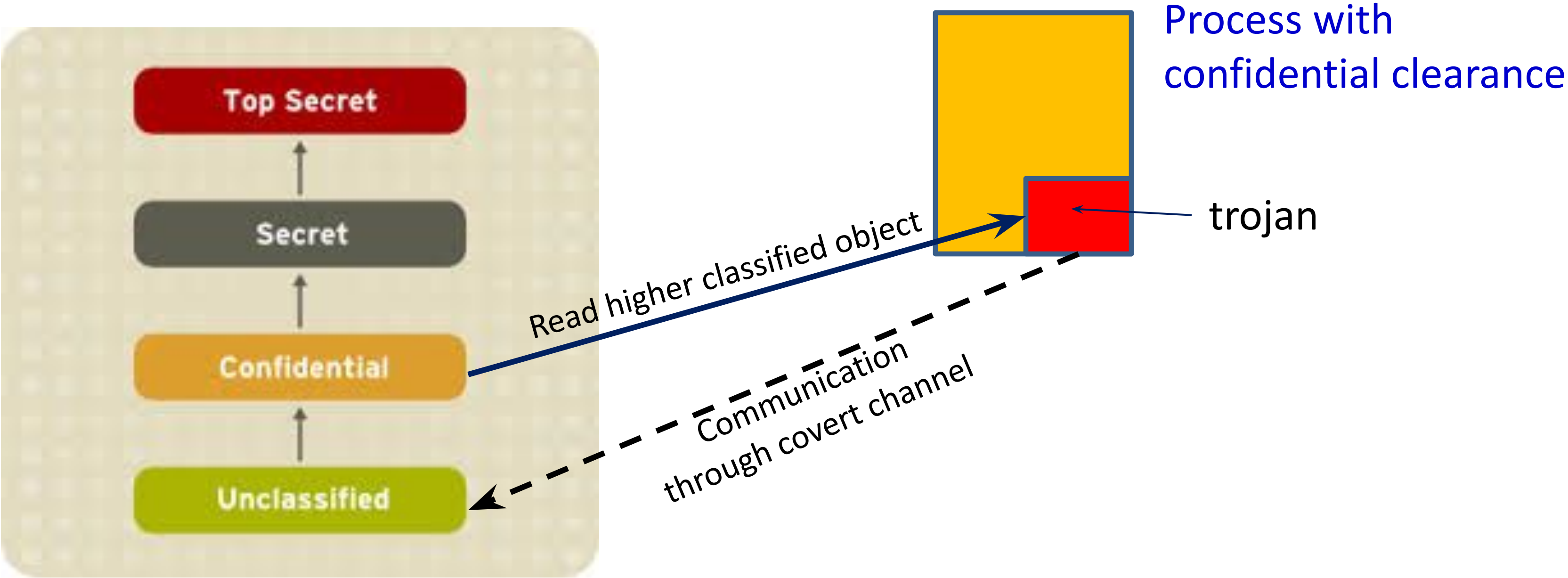
Need an additional rule about changing levels

Tranquility

- **Strong Tranquility Property:**
  - Subjects and objects do not change label during lifetime of the system

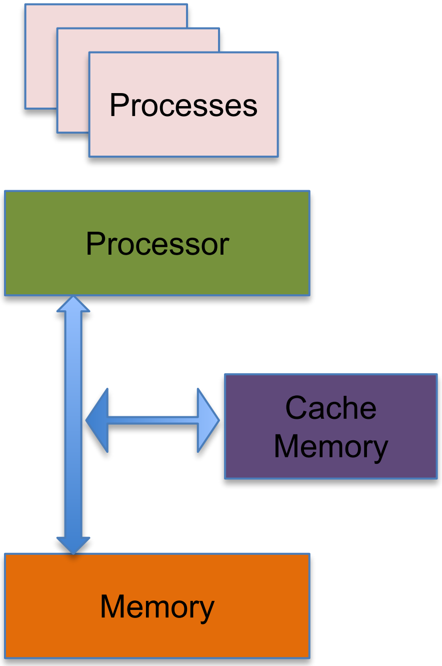
- **Weak Tranquility Property:**
  - Subjects and objects do not change label in a way that violates the *spirit* of the security policy.
  - Should define
    - How can subjects change clearance level?
    - How can objects change levels?

Limitations of BLP (Covert Channels)

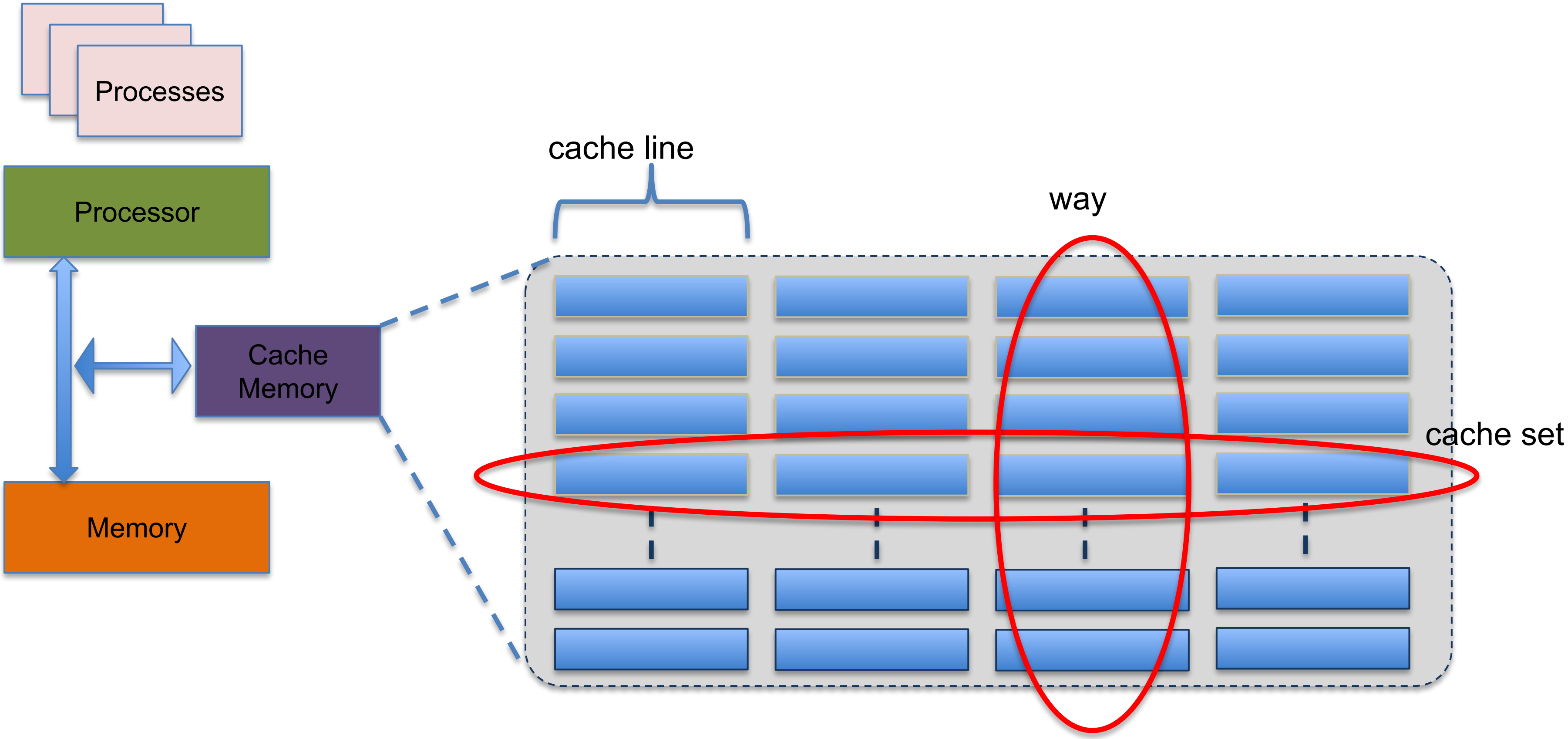

Top Secret
Secret
Confidential
Unclassified
Read higher classified object
Communication through covert channel
Process with confidential clearance
trojan

- Covert channels through system resources that normally not intended for communication.
- covert channel examples:
  page faults, file lock, cache memory, branch predictors , rate of computing, sockets
- Highly noisy, but can use coding theory to encode / decode information through noisy channels

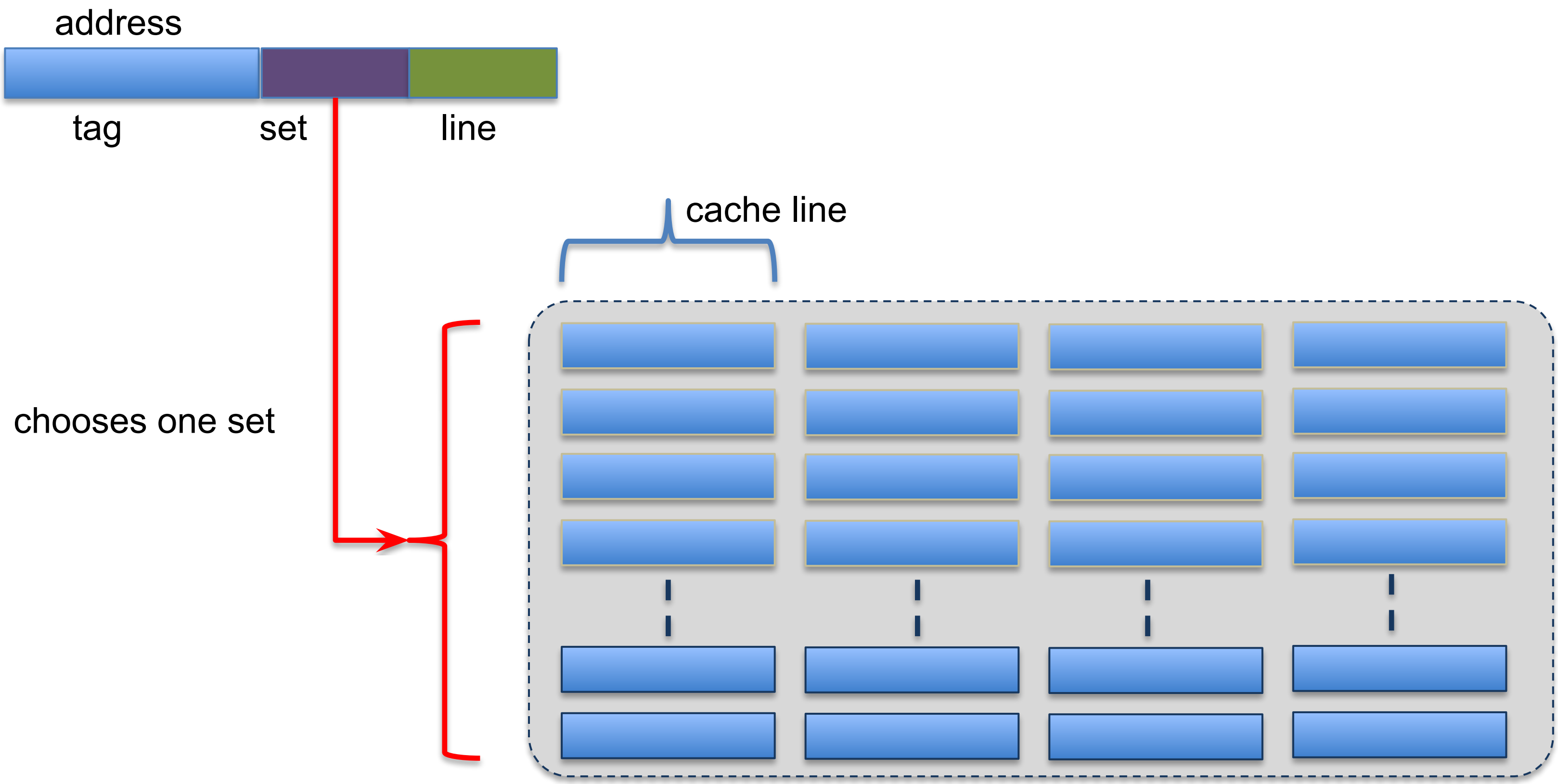
Cache Covert Channel



Cache Covert Channel



Cache Covert Channel



Cache Covert Channel

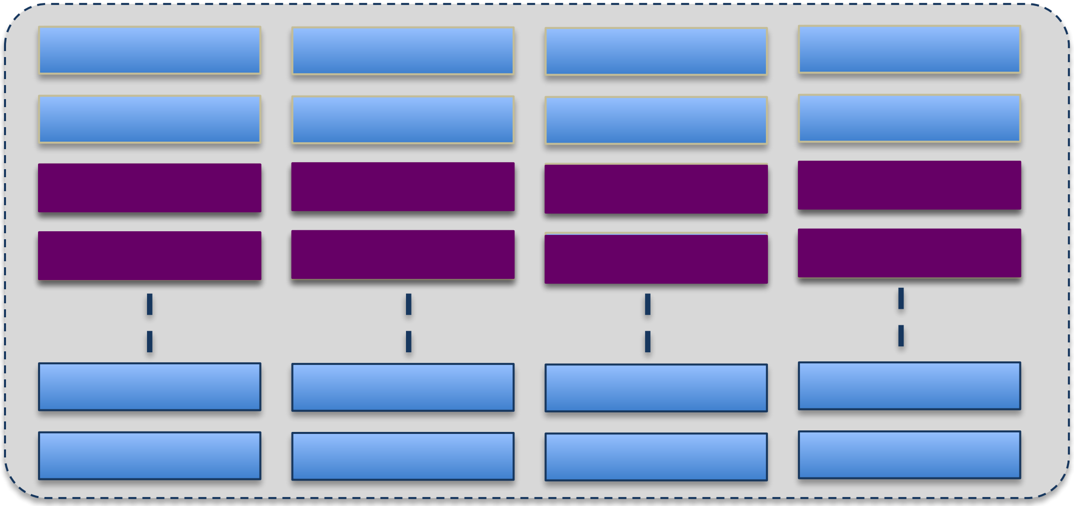
statistically
time A ~ time B

```
while(1){
  load A1; load A2
  load A3; load A4
  load B1; load B2
  load B3; load B4
}
```

Process
P2

A Set

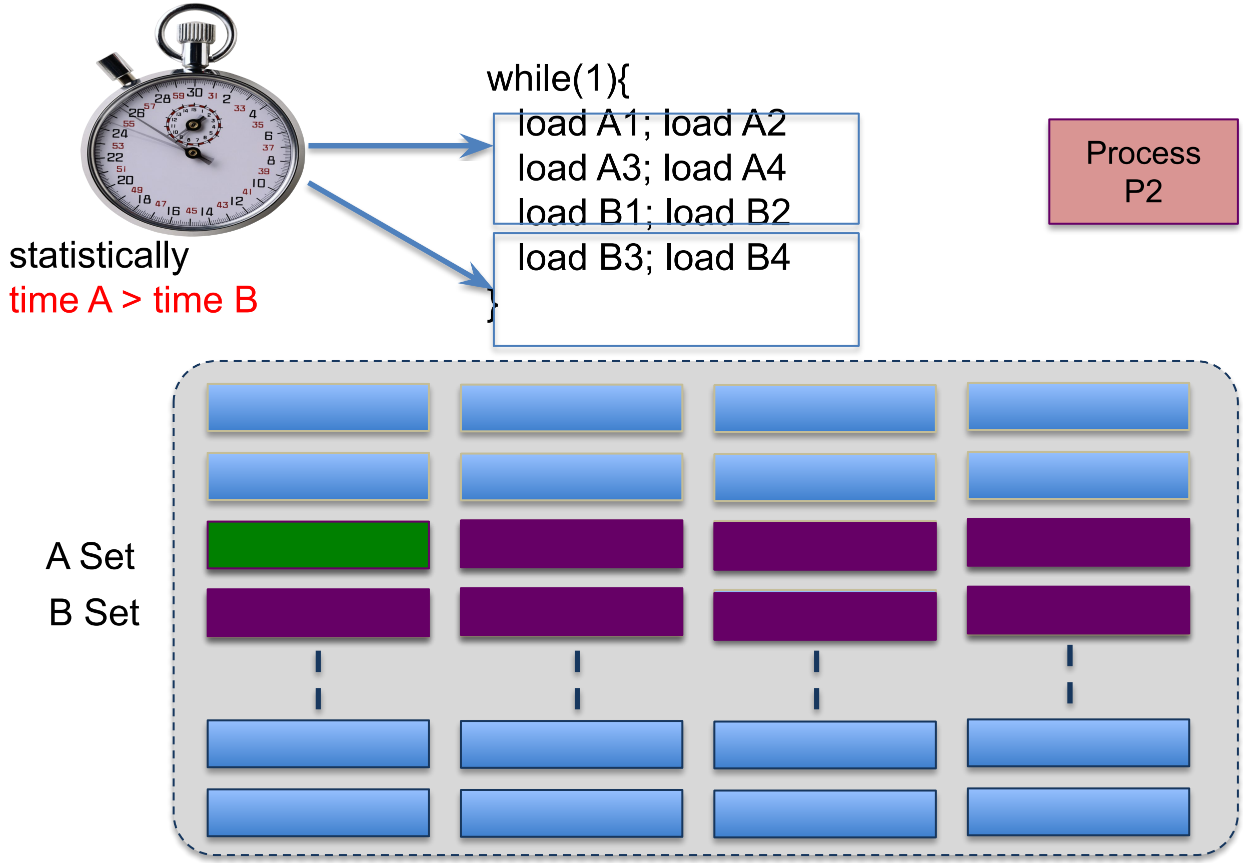
B Set



Cache Covert Channel

Process
P1

```
If (bit == 1)
    load $A_{P1}$
Else
    load $B_{P1}$
```



statistically
time A > time B

```
while(1){
    load A1; load A2
    load A3; load A4
    load B1; load B2
    load B3; load B4
}
```

Process
P2

Cache Covert Channel

Process
P1

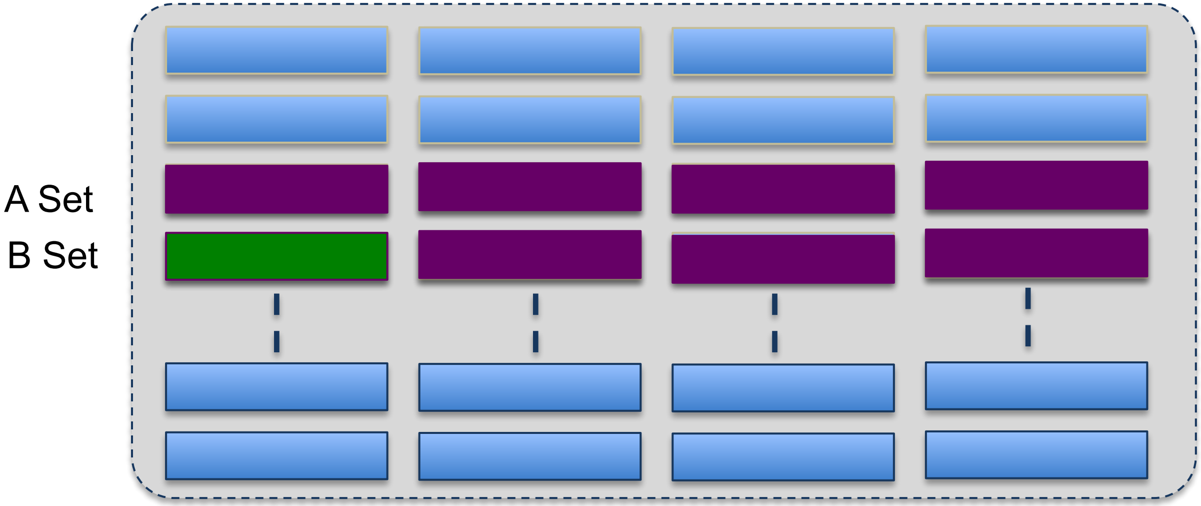
```
If (bit == 1)
    load A_P1
Else
    load B_P1
```

statistically

time A < time B

```
while(1){
  load A1; load A2
  load A3; load A4
  load B1; load B2
  load B3; load B4
}
```

Process
P2



Cache Covert Channel

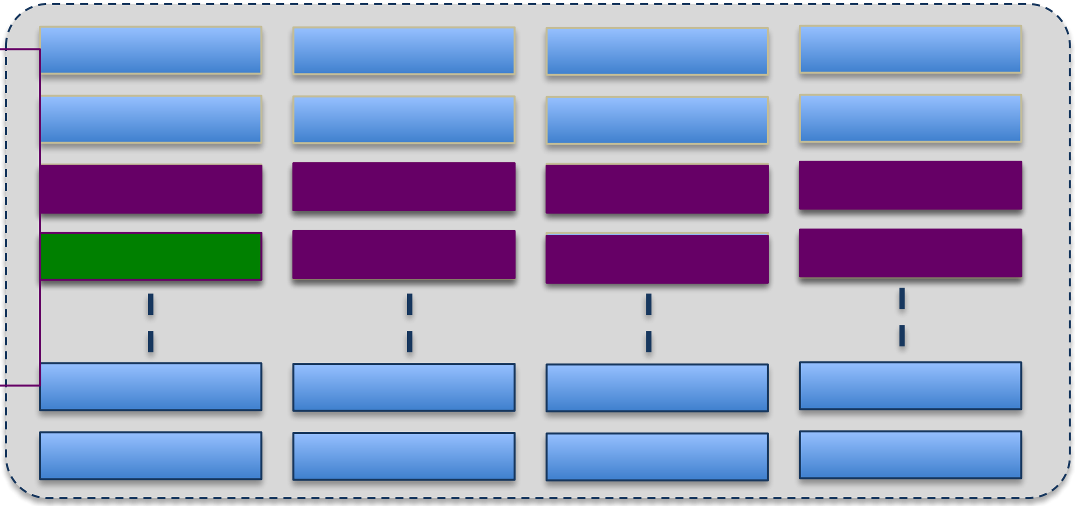
Process
P1

statistically
time A < time B

```
while(1){
  load A1; load A2
  load A3; load A4
  load B1; load B2
  load B3; load B4
}
```

Process
P2

```
bit = message
while(bit[i] != ‘\0’)
    for(some number of iterations)
      If (bit[i] == 1)
        load $A_{P1}$
      else
        load $B_{P1}$
```



Covert Channels

- Identifying: Not easy because simple things like the existence of a file, time, etc. could be a source for a covert channel.

- Quantification: communication rate (bps)

- Elimination: Careful design, separation, characteristics of operation (eg. rate of opening / closing a file)

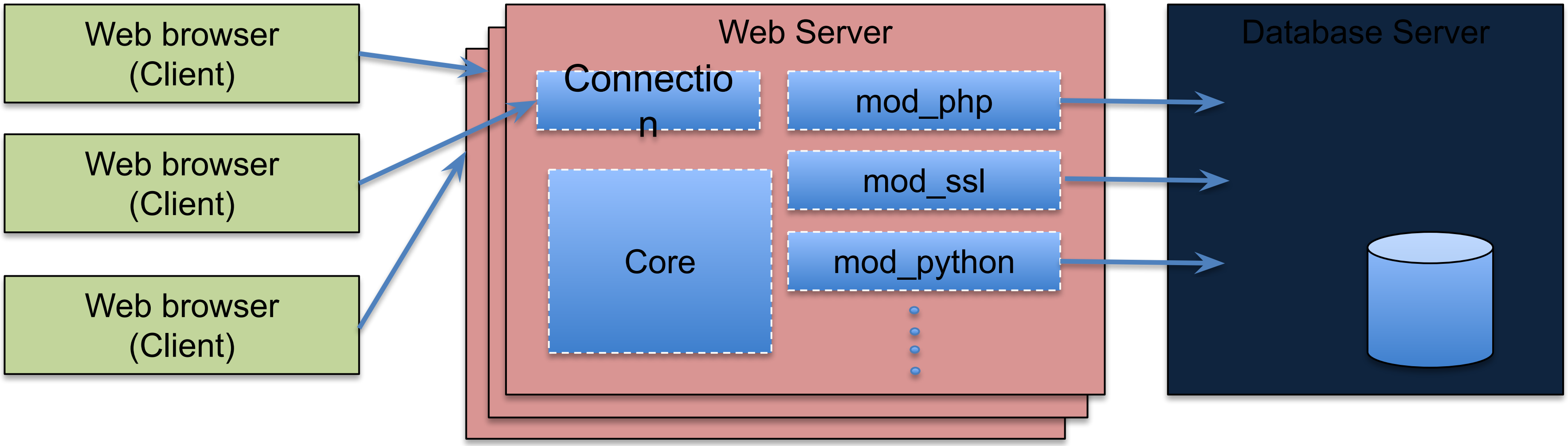
Principle of Least Privilege

- Every subject has access to the minimum amount of information and resources that are necessary
- Useful for implementing weak tranquility.

Case Study OKWS Web Server

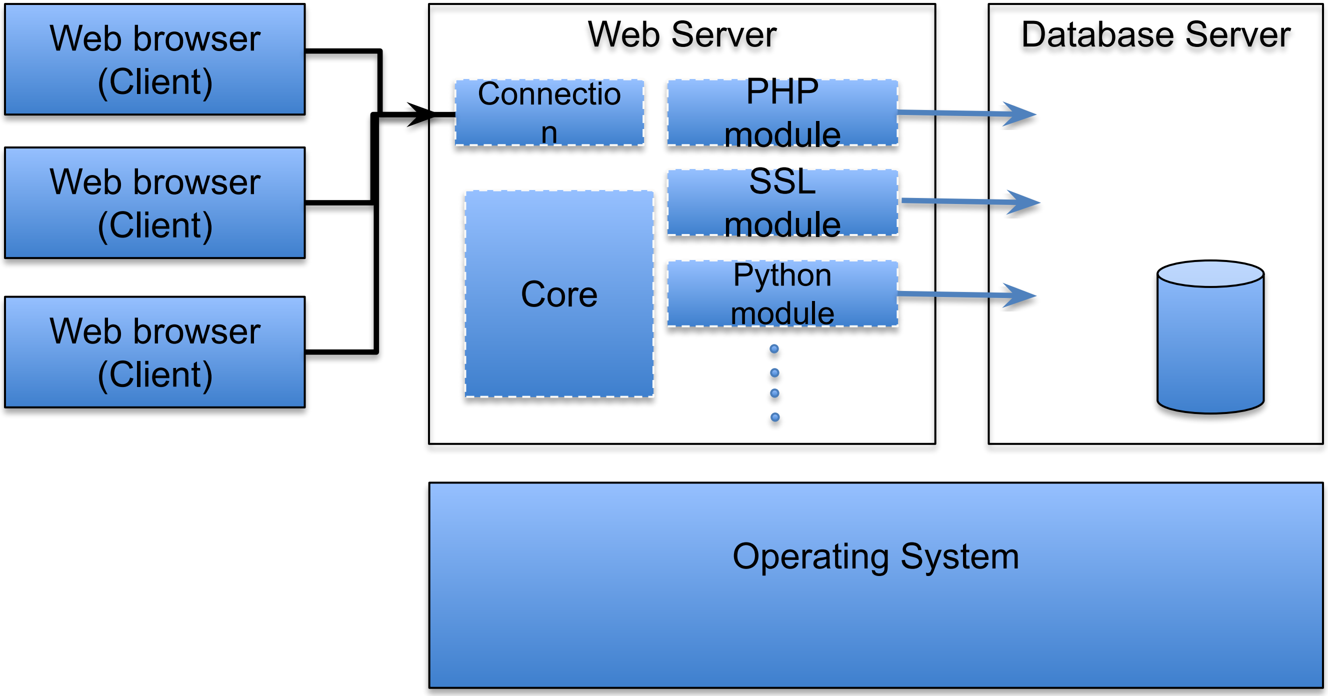
Case Study 1
(Secure Web Server)

Typical Web Server

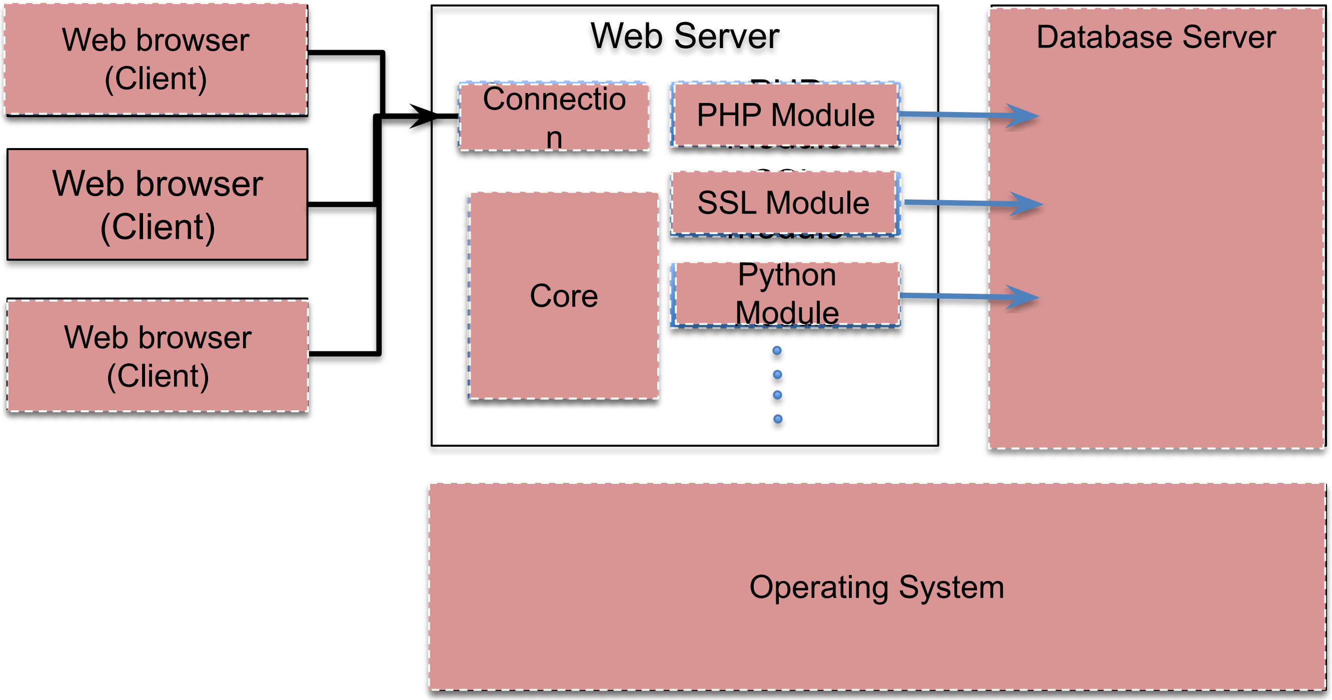


- single address space holds multiple web servers
- Every new client creates a new process
- HTTP interfaces restrict access to the database server
- Security achieved by coarse grained access control mechanisms in the data base server
- A vulnerability in any component can ripple through the entire system

Apache Webserver



Apache Webserver



Known attacks on Web Servers

- A bug in one website can lead to an attack in another website
  example: Amazon holds credit card numbers. If it happens to share the same web server as other users this could lead to trouble.

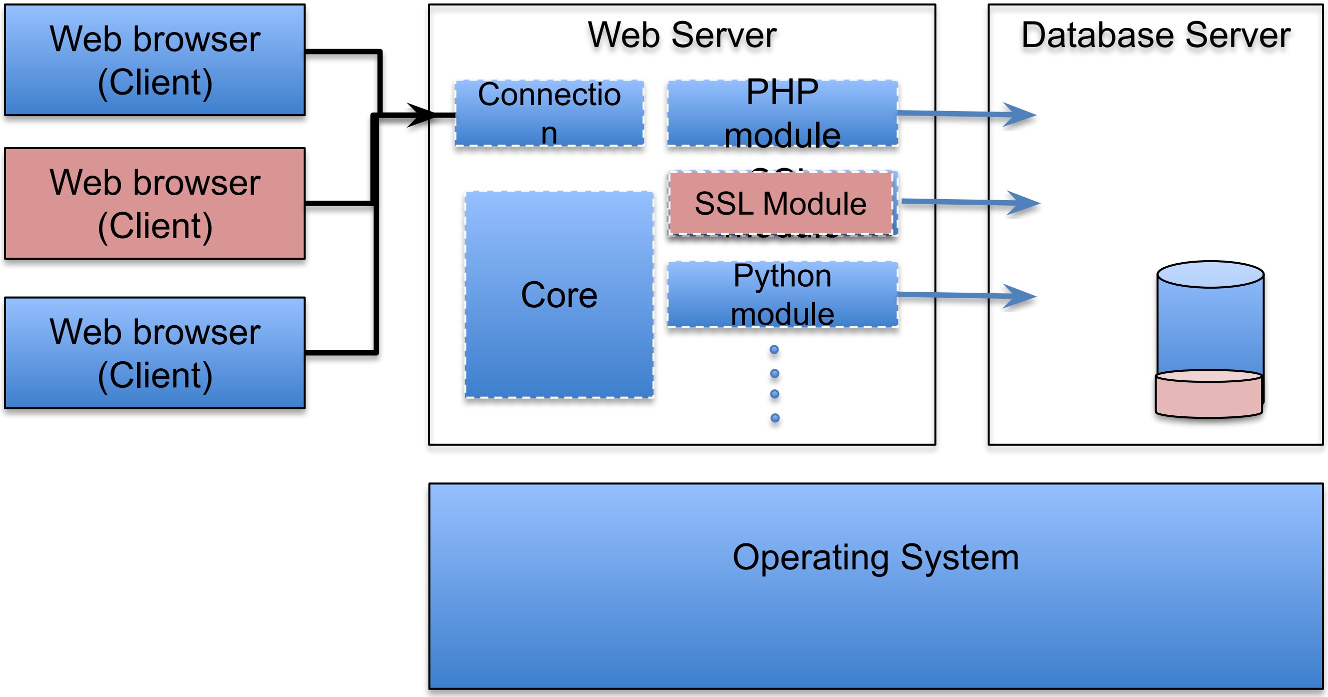
- Some known attacks on Apache's webserver and its standard modules
  - Unintended data disclosure (2002)
    users get access to sensitive log information
  - Buffer overflows and remote code execution (2002)
  - Denial of service attacks (2003)
  - Due to scripting extensions to Apache

Principle of Least Privileges

Aspects of the system most vulnerable to attack are the least useful to attackers.

- Decompose system into subsystems
- Grant privileges in fine grained manner
- Minimal access given to subsystems to access system data and resources
- Narrow interfaces between subsystems that only allow necessary operations
- Assume exploit more likely to occur in subsystems closer to the user (eg. network interfaces)
- Security enforcement done outside the system (eg. by OS)

Apache Webserver



OKWS Webserver (designed for least privileges)

each independent service runs in an independent process

Do not expose more code/services than required! Tradeoff security vs performance

Each service should run in a separate chroot jail

Allow access to only necessary files.

Each process should run as a different unprivileged user.

Prevent interfering with other processes

Narrow set of database access privileges

Prevent unrequired access to the DB service

https://www.usenix.org/event/usenix04/tech/general/full_papers/krohn/krohn.pdf

Achieving Confinement

Through Unix Tools

- **chroot:** define the file system a process can see

  if system is compromised, the attacker has limited access to the files. Therefore, cannot get further privileges

- **setuid:** set the uid of a process to confine what it can do

  if system runs as privileged user and is compromised, the attacker can manipulate other system processes, bind to system ports, trace system calls, etc.

- **Passing file descriptors:** a privileged parent process can open a file and pass the descriptor to an unprivileged child (don’t have to raise the privilege of a child, to permit it to access a specific high privileged file)

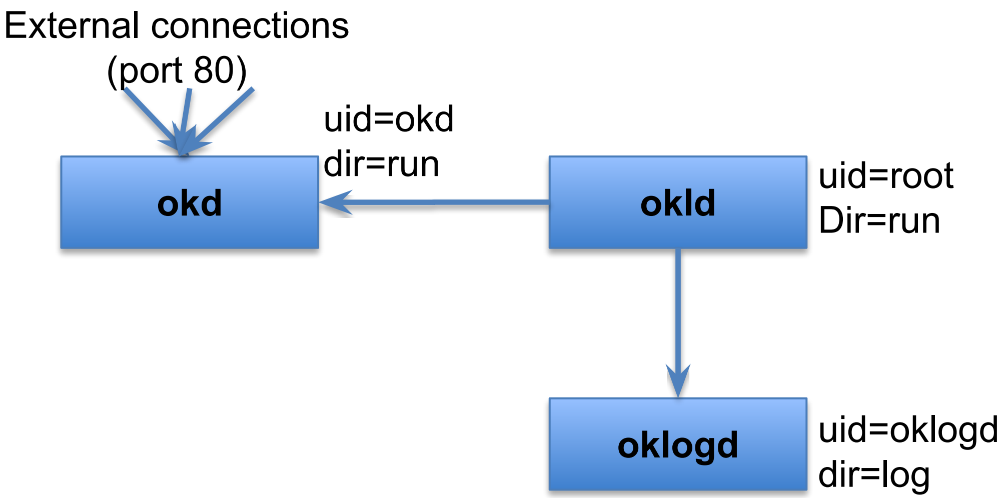
OKWS Design

okld

uid=root
dir=run

runs as superuser; bootstrapping; chroot directory is run monitors processes; relaunches them if they crash

OKWS Design



External connections (port 80)

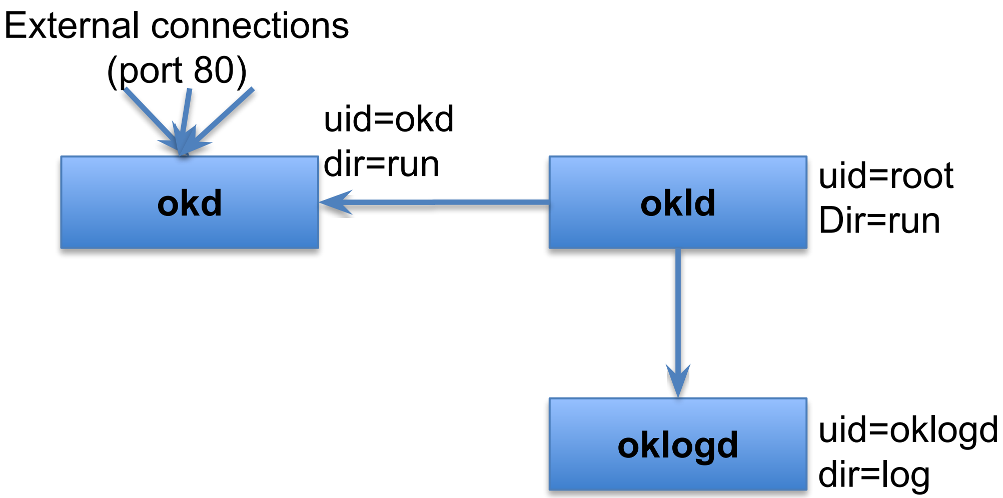
Launch okd (demux daemon) to route traffice to appropriate service ;

If request is valid, forwards the request to the appropriate service

If request is invalid, send HTTP 404 error to the remote client

If request is broken, send HTTP 500 error to the remote client

OKWS Design

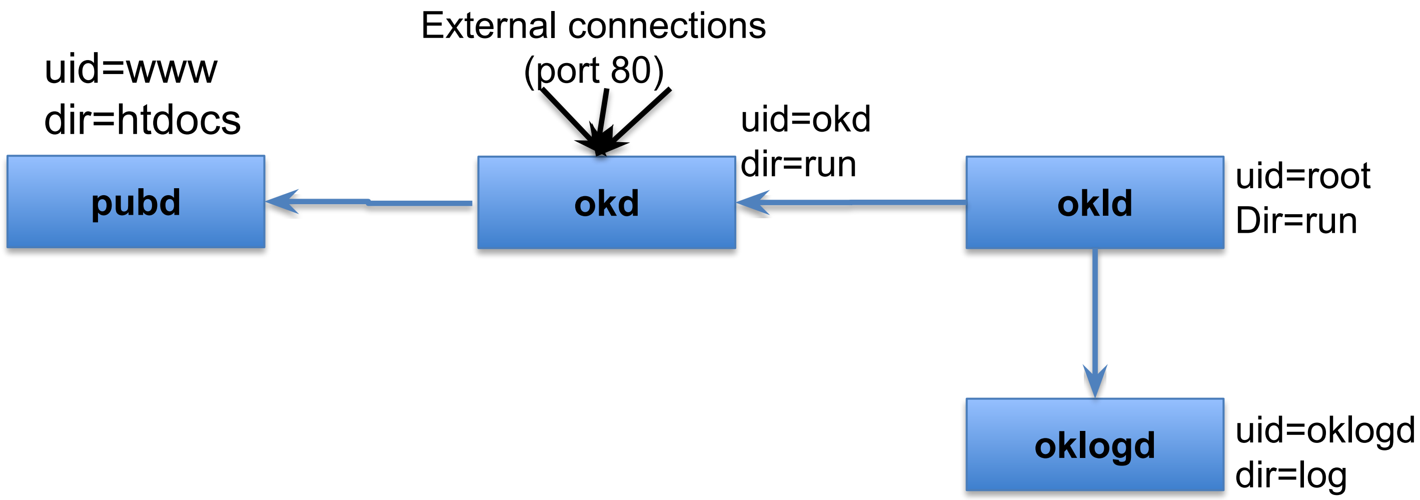


oklogd daemon to write log entries to disk

chroot into their own runtime jail (within a jail, each process has just enough access privileges to read shared libraries on startup, dump core files if crash)

Each service runs as an unprivileged user

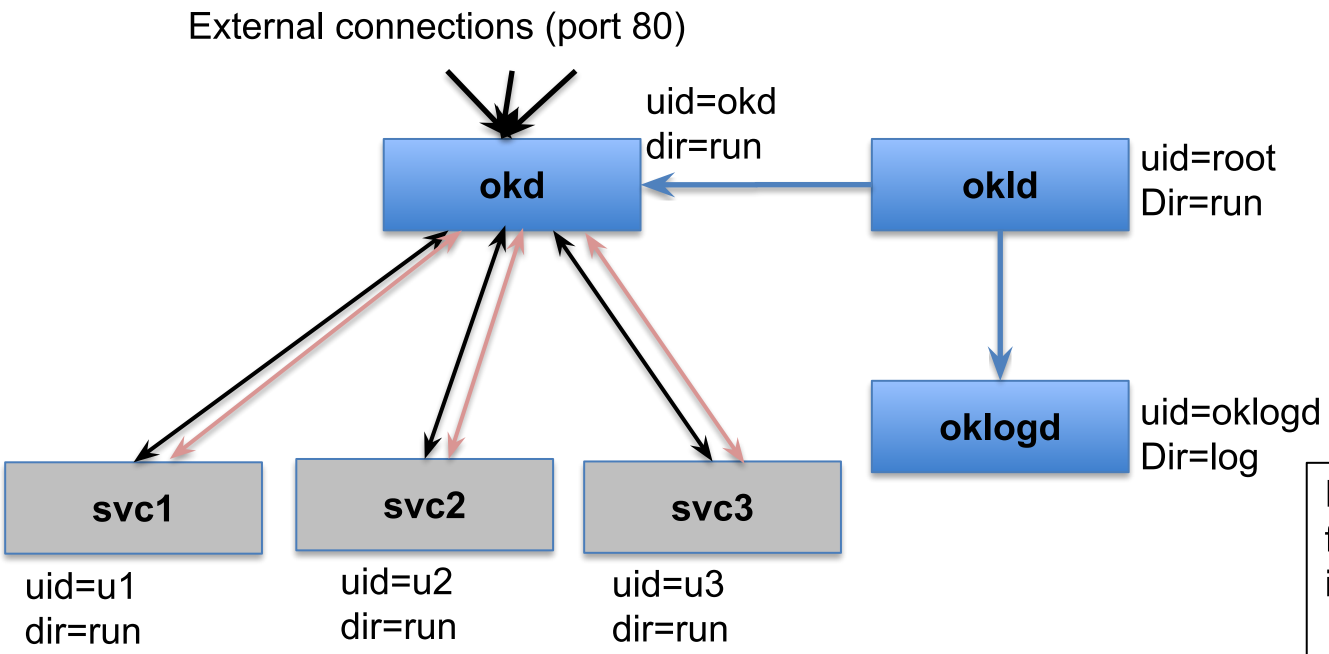
OKWS Design



pubd: provides minimal access to local configuration files, html files

Read only access to the files

OKWS Design



```
Request 2 sockets
fork()
if (child process){
    setuid()
    chroot()
    exec()
}
```

okld launch services; each service in its chroot with its own uid

Services owned by root with permissions 0410 (can only be executed by user)

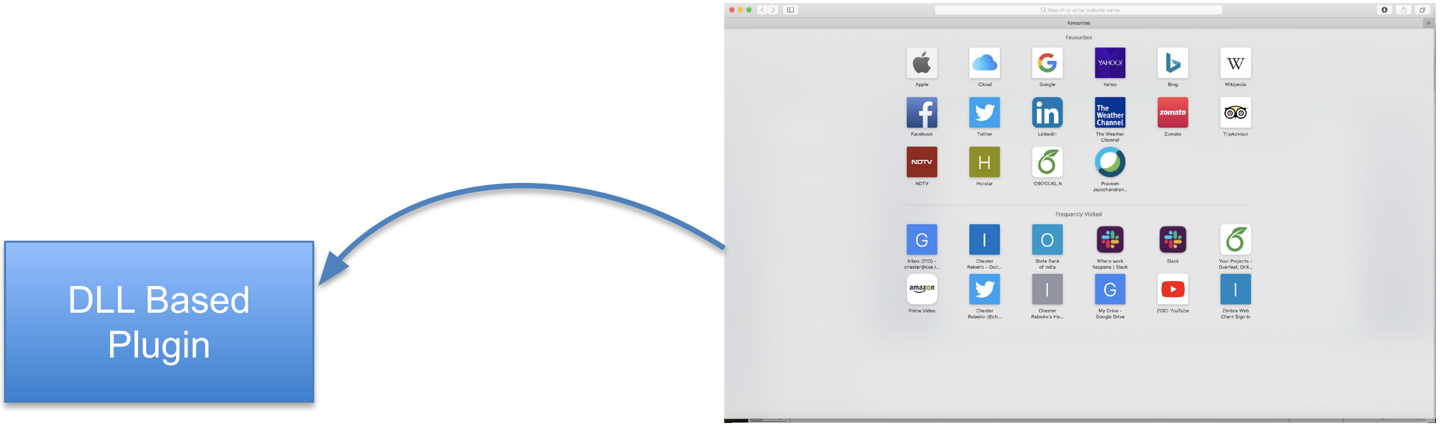
okld catches SIGCHILD and restarts services if they crash

Logging

- Each service uses the same logging file
  - They use the oklogd to write into the file via RPCs
  - oklogd runs in its own chroot jail
    - Any compromised service will not be able to modify / read the log
    - A compromised service may be able to write arbitrary messages to the log (noise)

Case Study 2
Secure Web Browser

Access Control in a Web Browser



DLL Based Plugin

eg. PDF viewer

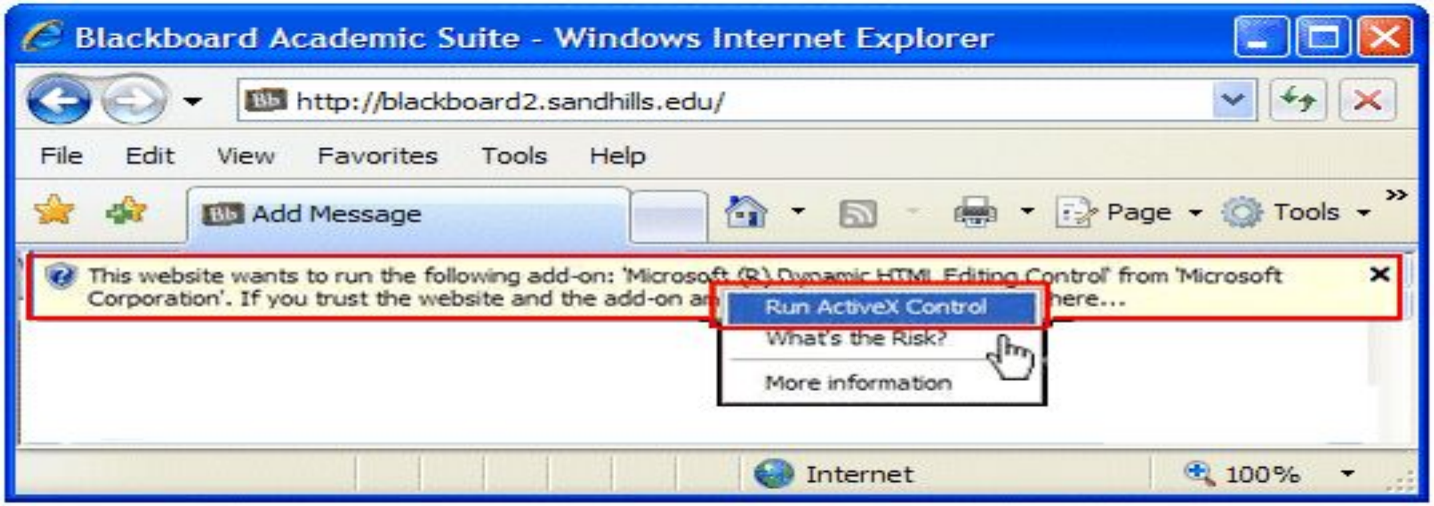
Web browser

Typically very buggy

Web Browser Confinement

- How to allow an untrusted module to load into a web-browser?
  - Trust the developer / User decides

    Active X



This website wants to run the following add-on: 'Microsoft (R) Dynamic HTML Editing Control' from 'Microsoft Corporation'. If you trust the website and the add-on ar here...

Web Browser Confinement

- How to allow an C/C++ in a web-browser?
  - Trust the developer / User decides

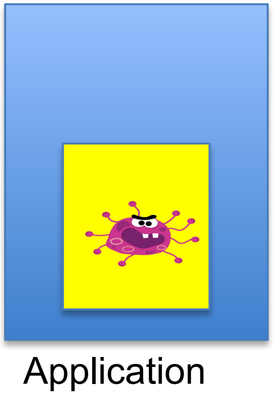
    Active X
  - Fine grained confinement
    - (eg. NACL from Google)
    - Uses Software Fault Isolation
  - Restrictive languages
    - Web assembly

Fine Confinement within a Process

- How to
  - restrict a module from jumping outside its module
  - Restrict read/modification of data in another module

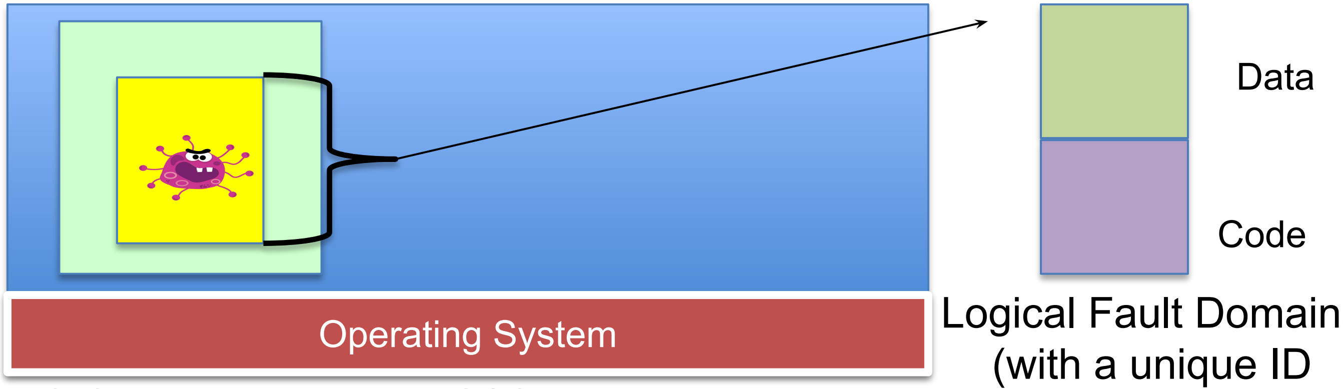
(jumping outside a module and access to data outside a module should be done only through prescribed interfaces)

(can use RPCs, but huge performance overheads)



Fine Grained Confinement
(Software Fault Isolation)

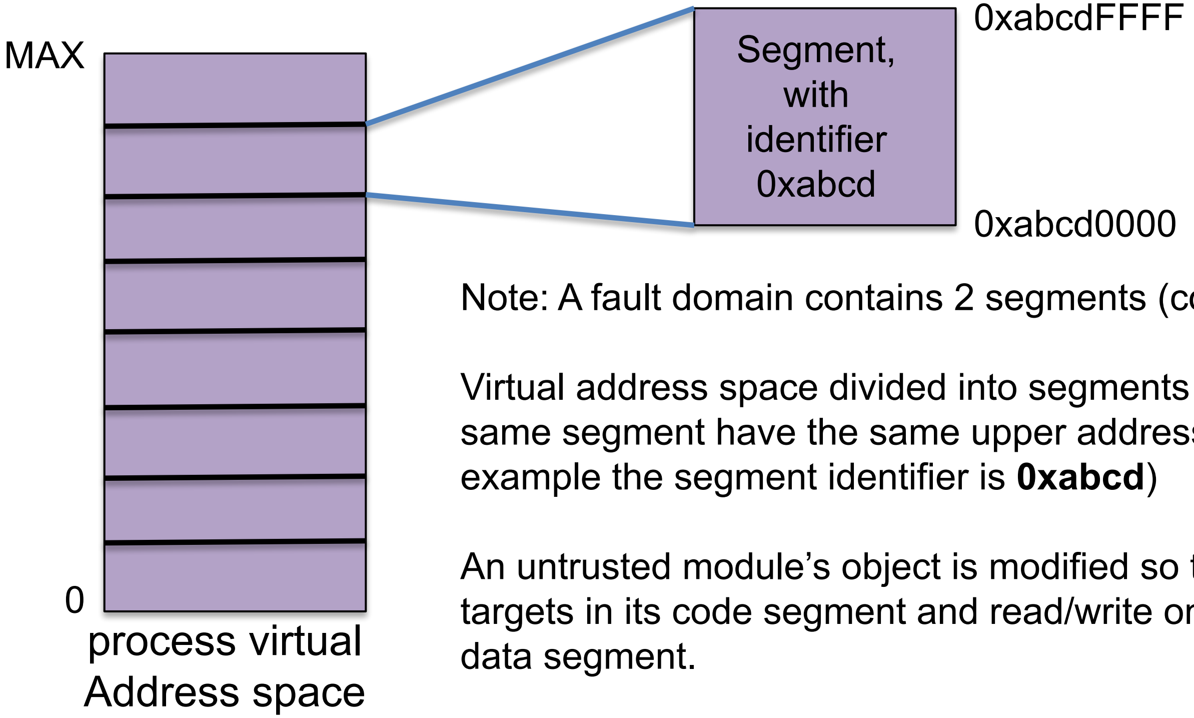
- process space partitioned into logical fault domains.
- Each fault domain contains data, code, and a unique ID
- Code in one domain not allowed to read/modify data in another domain.
- Code in one domain cannot jump to another domain.
- The only way is through a low cost cross-fault-domain RPC interface not involving the OS..



Logical Fault Domains (with a unique ID which is used for access control)

Wahbe et al. Efficient Software Fault Isolation, SOSP 93
https://dl.acm.org/citation.cfm?doid=168619.168635

Segments and Segment Identifier

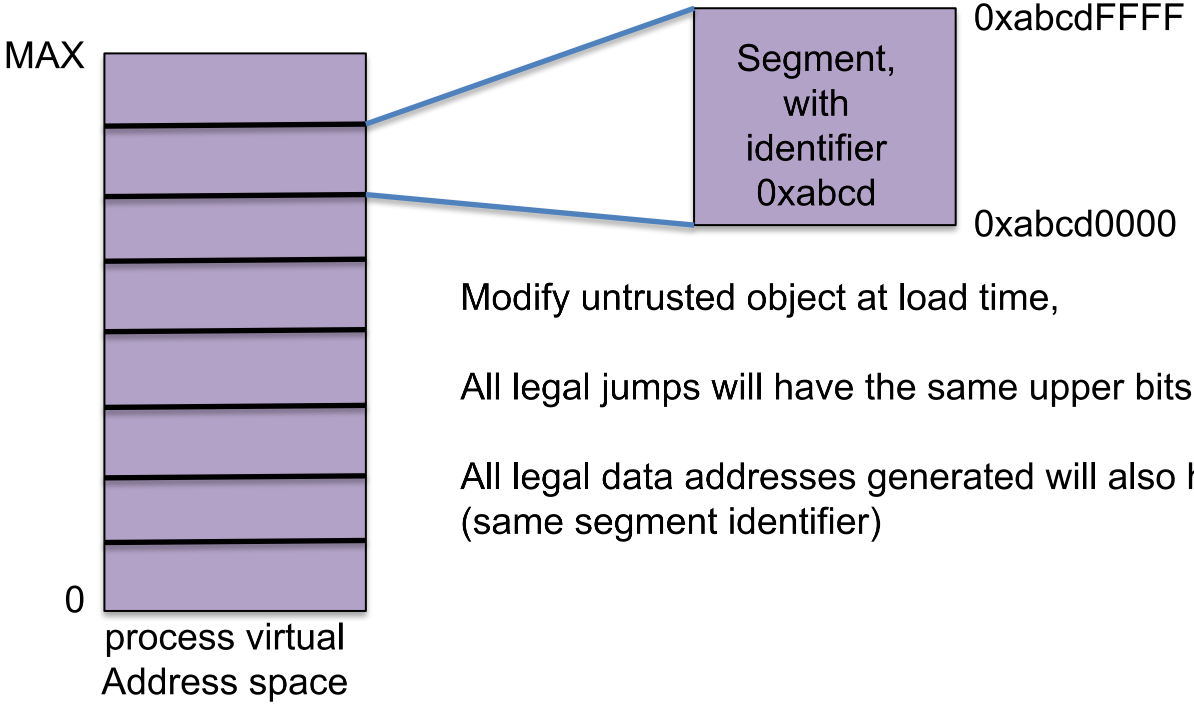


Note: A fault domain contains 2 segments (code; data+stack+heap)

Virtual address space divided into segments such that addresses in the same segment have the same upper address bits (eg in the above example the segment identifier is **0xabcd**)

An untrusted module’s object is modified so that it can jump only to targets in its code segment and read/write only to addresses within its data segment.

Segments and Segment Identifier



Modify untrusted object at load time,

All legal jumps will have the same upper bits (same segment identifier)

All legal data addresses generated will also have the same upper bits (same segment identifier)

Safe Instructions

- Compile time techniques / Load time techniques
  - Scan the binary from beginning to end.
  - Reliable disassembly: by scanning the executable linear
    - variable length instructions may be issues

```
25  CD  80  00  00                      CD  80  00  00
AND %eax, 0x000080CD                    INT $0x80
```

    - A jump may land in the middle of an instruction
    - Two ways to deal with this—
      - Ensure that all instructions are at 32 byte offsets
      - Ensure that all Jumps are to 32 byte offset

```
AND eax, 0xffffffe0
JMP *eax
```

Achieving Segmentation

- Binary rewriting statically
  - At the time of loading, parse through the untrusted module to determine all memory read and write instructions and jump instructions.
  - Use unique ID (upper bits) to determine if the target address is legal
  - Rewriting can be done either at compile time (modifying compiler) or at load time. (currently only compile time rewriting feasible)
  - A verifier also needed when the module is loaded into the fault domain.

Safe & Unsafe Instructions

Safe Instructions:

- Most instructions are safe (such as ALU instr)
- Many of the target addresses can be resolved statically
  (jumps and data addresses within the same segment id. These are also **safe instructions**)

Safe Instructions

- Compile time techniques / Load time techniques
  - Scan the binary from beginning to end.
  - Reliable disassembly: by scanning the executable linearl
    - variable length instructions may be issues

```
25  CD  80  00  00                       CD  80  00  00
AND %eax, 0x000080CD                     INT $0x80
```

- A jump may land in the middle of an instruction
- Two ways to deal with this—
  - Ensure that all instructions are at 32 byte offsets
  - Ensure that all Jumps are to 32 byte offset

```
AND eax, 0xffffffe0
JMP *eax
```

Unsafe Instructions

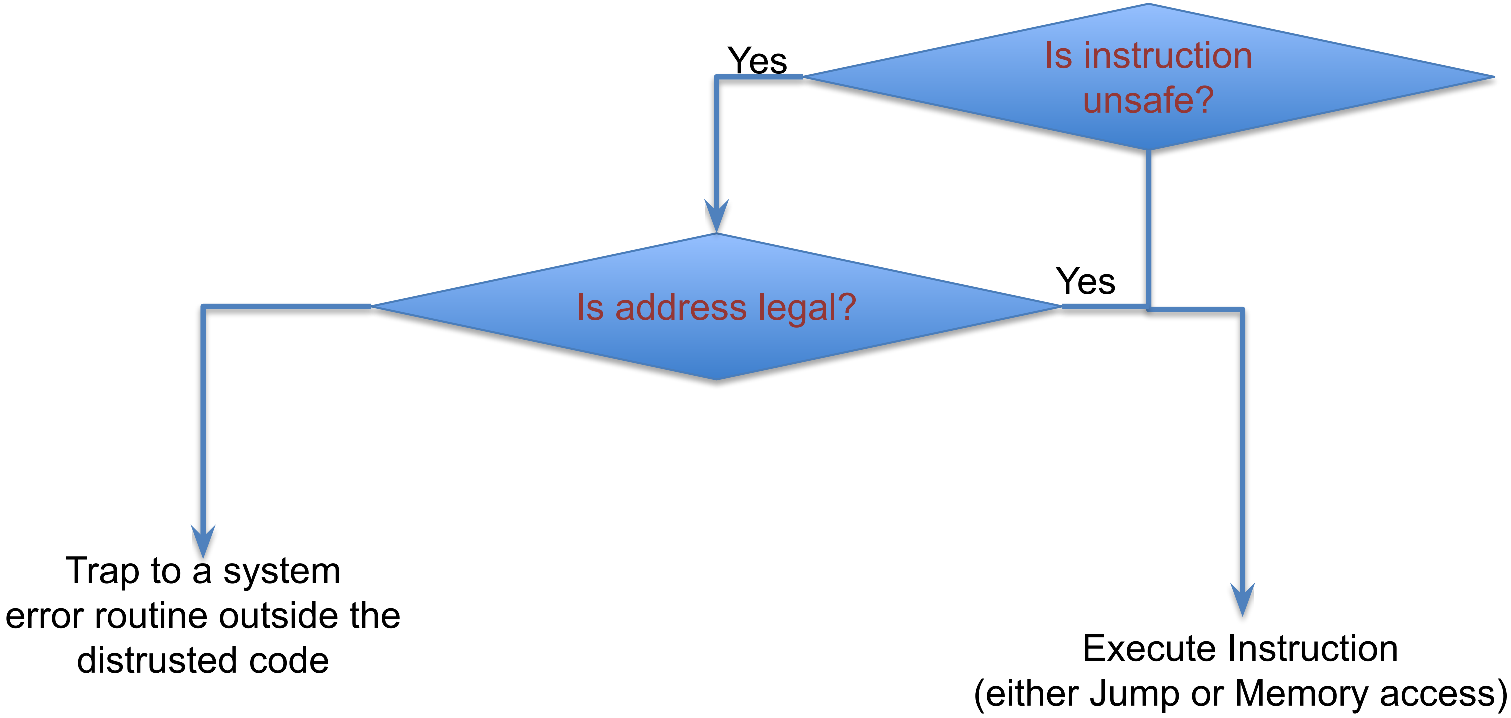
Prohibited Instructions:

- Eg. int, syscall, etc.

Unsafe Instructions: Cannot be resolved statically.

- For example *store 0x100, [r0]*
- Unsafe targets need to be validated at **runtime**
- Jumps based on registers (eg. Call *eax), and Load/stores that use indirect addressing are unsafe.
  Eg. JMP *eax

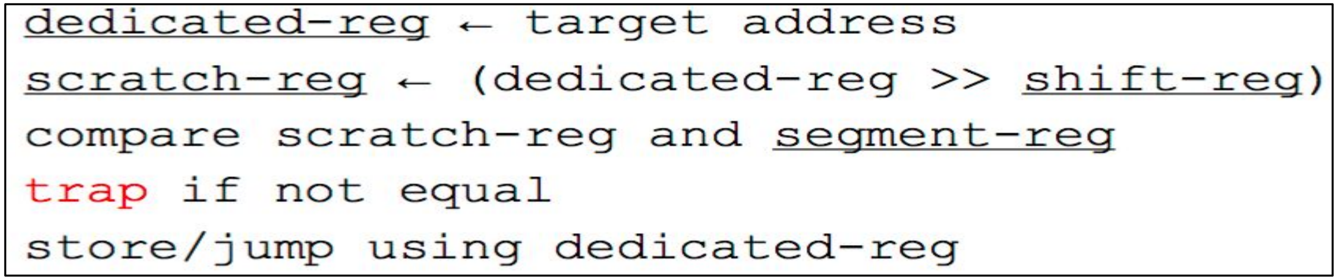
Runtime Checks for Unsafe Instructions (segment matching)


Is instruction unsafe?
Yes
Is address legal?
Yes
Trap to a system error routine outside the distrusted code
Execute Instruction (either Jump or Memory access)

Run Time Checks Segment Matching

Insert code for every unsafe instruction that would trap if the store is made outside of the segment

4 registers required (underlined registers)



```
dedicated-reg ← target address
scratch-reg ← (dedicated-reg >> shift-reg)
compare scratch-reg and segment-reg
trap if not equal
store/jump using dedicated-reg
```

Overheads increase due to additional instructions but the increase is not as high as with RPCs across memory modules.

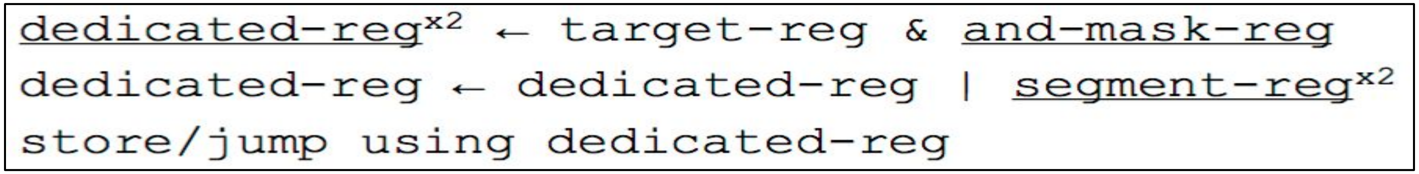
Address Sandboxing

- Segment matching is strong checking.
  - Able to detect the faulting instruction (via the trap)
- Address Sandboxing : Performance can be improved if this fault detection mechanism is dropped.
  - Performance improved by not making the comparison but forcing the upper bits of the target address to be equal to the segment ID
  - Cannot catch illegal addresses but prevents module from illegally accessing outside its fault domain.

Segment Matching : Check :: Address Sandboxing : Enforce

Address Sandboxing

Requires 5 dedicated registers



```
dedicated-regx2 ← target-reg & and-mask-reg
dedicated-reg ← dedicated-reg | segment-regx2
store/jump using dedicated-reg
```

Enforces that the upper bits of the dedicated-reg contains the segment identifier

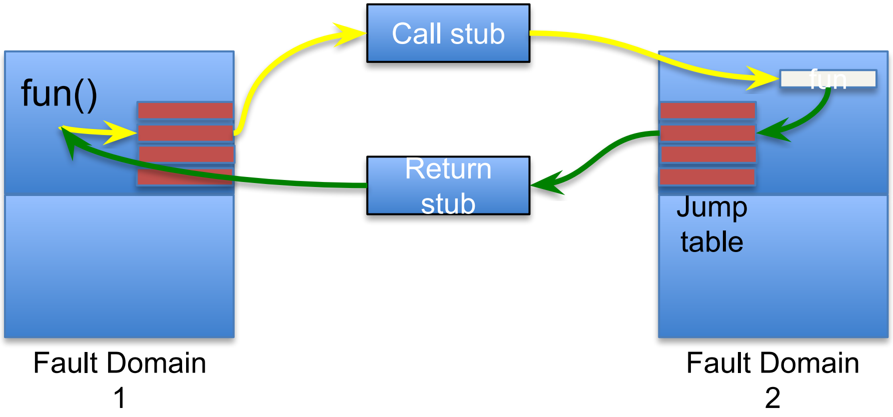
Ensure Valid Instructions

- How to ensure that jump targets are at valid instruction locations
  - Ensure that all instructions are at 32 byte offsets
  - Ensure that all Jumps are to 32 byte offset

```
AND eax, 0xffffffe0
JMP *eax
```

```
25  CD  80  00  00                                          CD  80  00  00
 AND %eax, 0x000080CD                                       INT $0x80
```

Calls between Fault Domains (light weight cross-fault-domain-RPC)



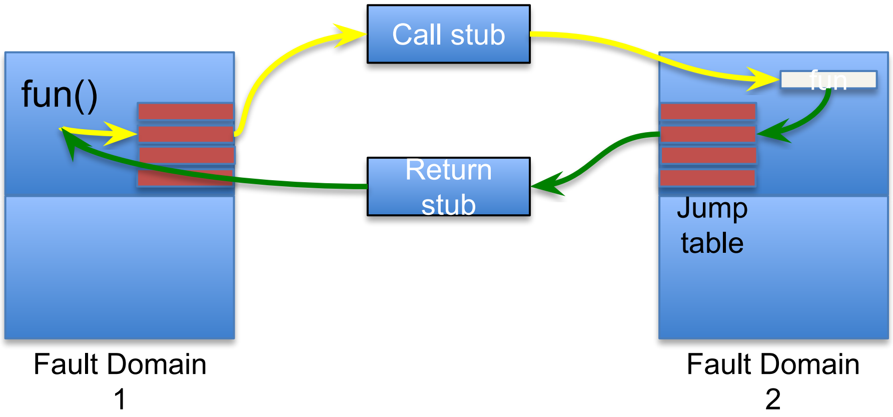
Safe calls outside a fault domain is by jump tables.

Each entry in jump table is a control transfer instruction whose target address is a legal entry point outside the domain.

Maintained in the read only segment of the program therefore cannot be modified.

Calls between Fault Domains (cross-fault-domain-RPC)

- A pair of stubs for each pair of fault domains
- Stubs are trusted
- Present outside the fault domains
- Responsible for
  - copying cross-domain arguments between domains
  - manages machine state (store/restore registers as required)
  - Switch execution stack
  - They can directly copy call arguments to the target domain



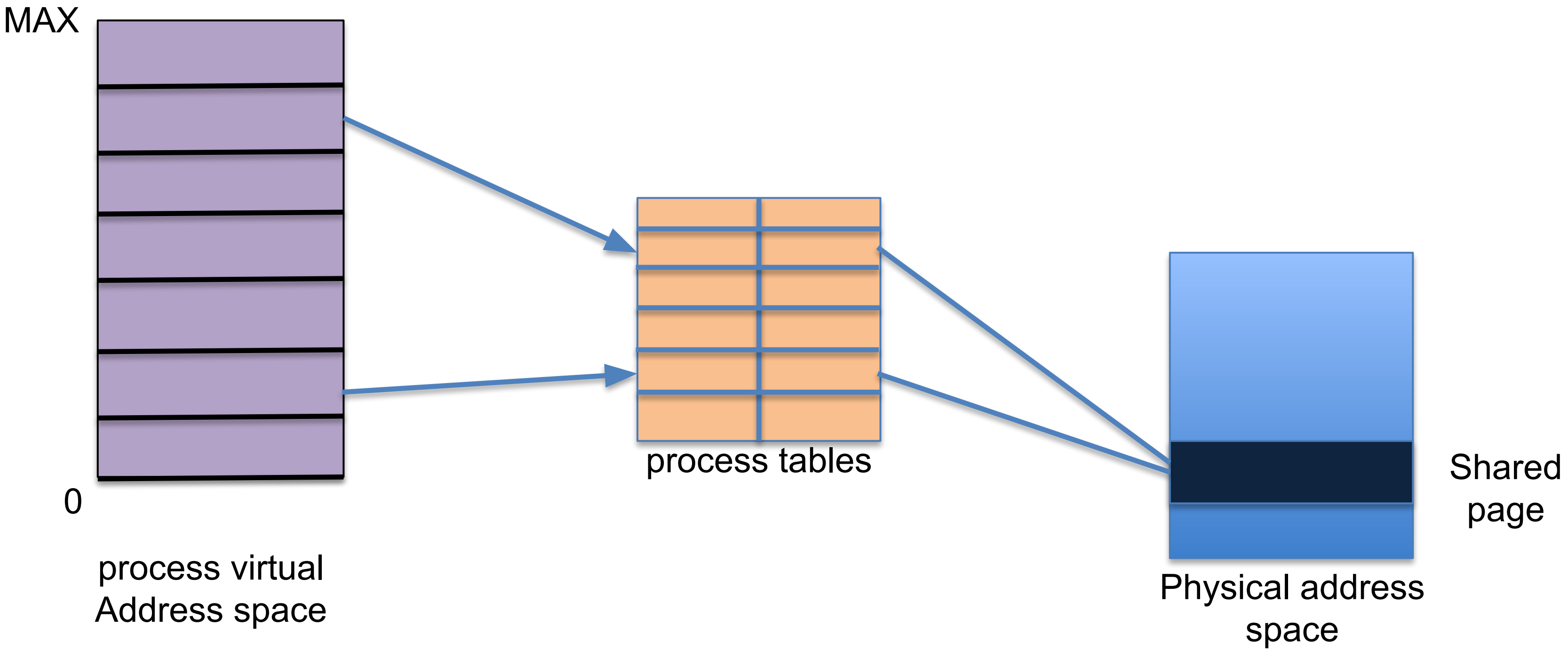
- Cheap
  - No traps, no context switches

System Resources

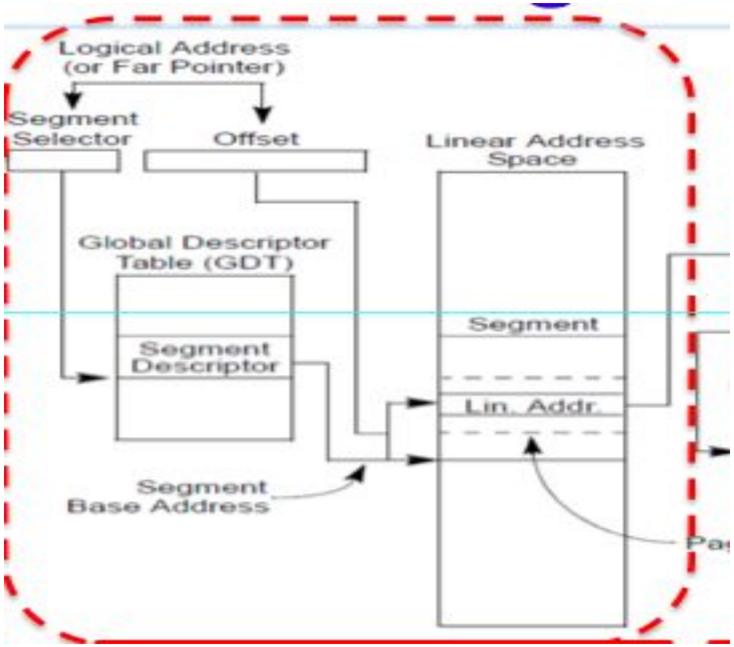
- How to ensure that one fault domain does not alter system resources used by another fault domain
  - For example, does not close the file opened by another domain
- One way,
  - Let the OS know about the fault domains
  - So, the OS keeps track if such violations are done at the system level
- Another (more portable way),
  - Modify the executable so that all system calls are made through a well defined interface called *cross-fault-domain-RPC.*
  - The cross-fault-domain-RPC will make the required checks.

Shared Data (Global / Heap Variables)

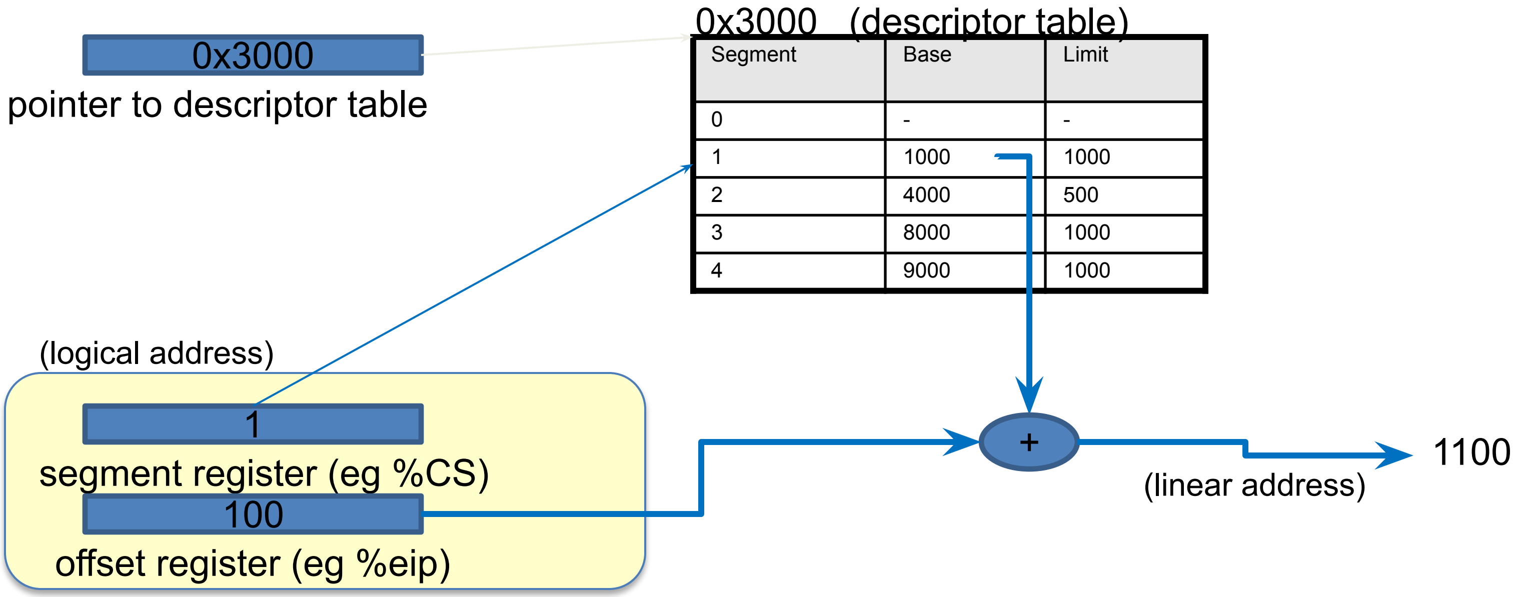
- Page tables in kernel modified so that shared memory mapped to every segment that needs access to it



Segmentation
(Hardware Support for Sandboxing)



Segmentation Example



0x3000

pointer to descriptor table

0x3000 (descriptor table)

Segment	Base	Limit
0	-	-
1	1000	1000
2	4000	500
3	8000	1000
4	9000	1000

(logical address)

1

segment register (eg %CS)

100

offset register (eg %eip)

+

1100

(linear address)

Segmentation In Sandboxing

- Create segments for each sandbox
- Make segment registers (CS, ES, DS, SS) point to these segments

- Need to ensure that the untrusted code does not modify the segment registers

- Jumping out of a segment: need to change segment registers appropriately

Usage

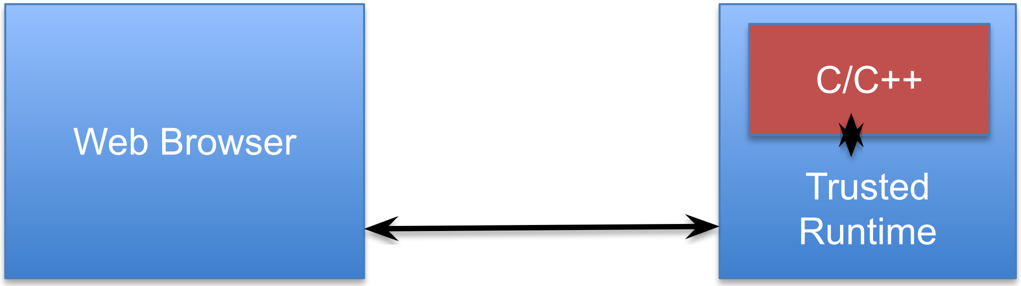
When to use it?

When you have an application with
- multiple tightly linked modules.
- a lot of shared data

If your application does not have these characteristics, then hardware based solutions are useful.

Native Client

- Used in Google Chrome till May 2017
- Uses SFI to run C/C++ code in a web browser (with support from Segmentation)
- A trusted environment for operations such as allocating memory, threading, message passing, etc



Native module always loaded in The range 0 to 256MB

https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/34913.pdf

Native Client Rules

1. Binary not writable
2. Start at mem 64K offset and extend to a max of 256MB
3. Indirect jumps protected by macro instructions
4. Pad memory after code with hlt instructions until page boundary
5. Direct jumps are to valid instructions
6. No instructions that span the 32-byte boundary
7. All instructions reachable by disassembly from the start