TEL AVIV UNIVERSITY

Introduction to Information Security

0368-3065, Spring 2015

Lecture 2:
Control Hijacking (2/2), Secure Architecture Principles, Access Control (1/2)

Eran Tromer

Slides credit:
Avishai Wool, Tel Aviv University

Preventing control-hijacking attacks

1. Fix bugs:
   - Static analysis
     - Tools: Coverity, Purify, PREfast/PREfix. ..
   - Runtime analysis
     - Tools: Valgrind
   - Rewrite software in a type-safe languange (Java, ML)
     - Difficult for existing (legacy) code …

2. Concede overflow, but prevent code execution

3. Add runtime code to detect overflows exploits
   - Halt process when overflow exploit detected
   - StackGuard, LibSafe, …

Control Hijacking

Platform defense

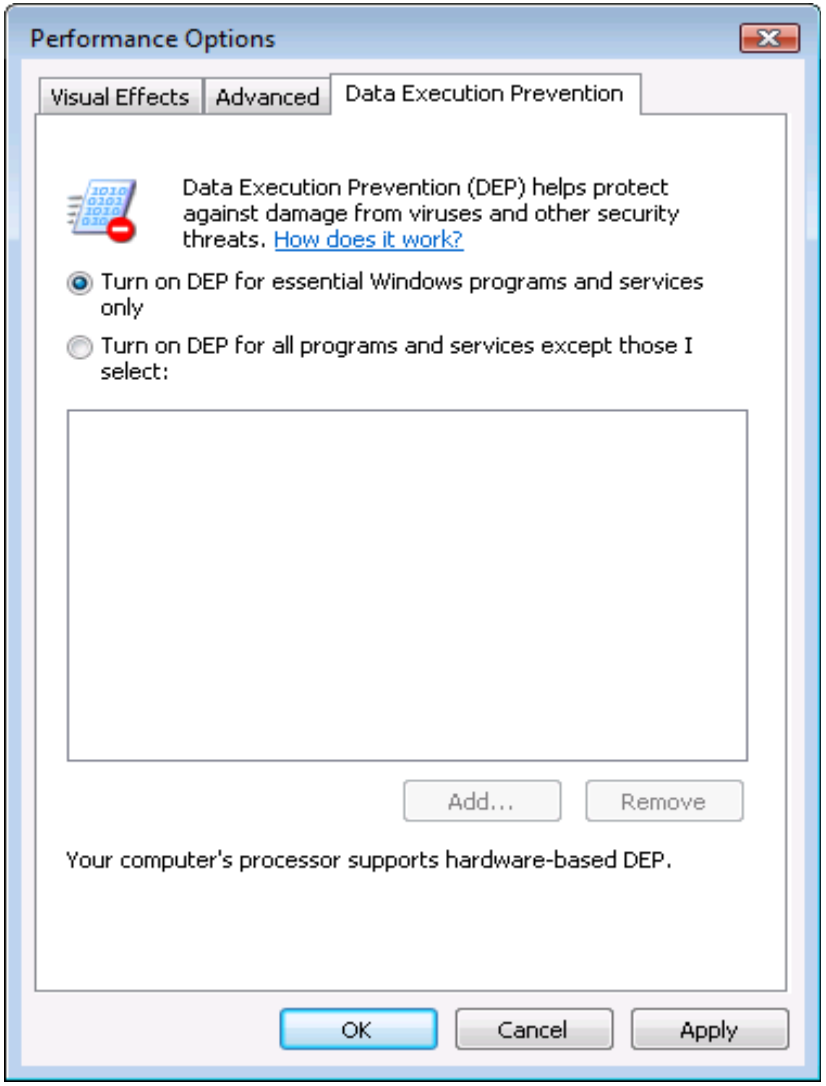
Marking memory as non-execute (W^X)

Prevent attack code execution by marking stack and heap as **non-executable.**
(von Neumann architecture ⇒ Harvard architecture)

- AMD: NX-bit ("No Execute", from AMD Athlon 64)
  Intel: XD-bit ("Executable Disable", from Intel P4 Prescott)
  - NX bit in every Page Table Entry (PTE)
- Deployment:
  - Linux (via PaX project); OpenBSD
  - Windows: since XP SP2 ("DEP")
    - Boot.ini : **/noexecute=OptIn** or **AlwaysOn**
    - Visual Studio: **/NXCompat[:NO]**
- Limitations:
  - Some apps need executable heap (e.g. JITs).
  - Does not defend against `**return-to-libc**' exploits

Examples: DEP control in Windows

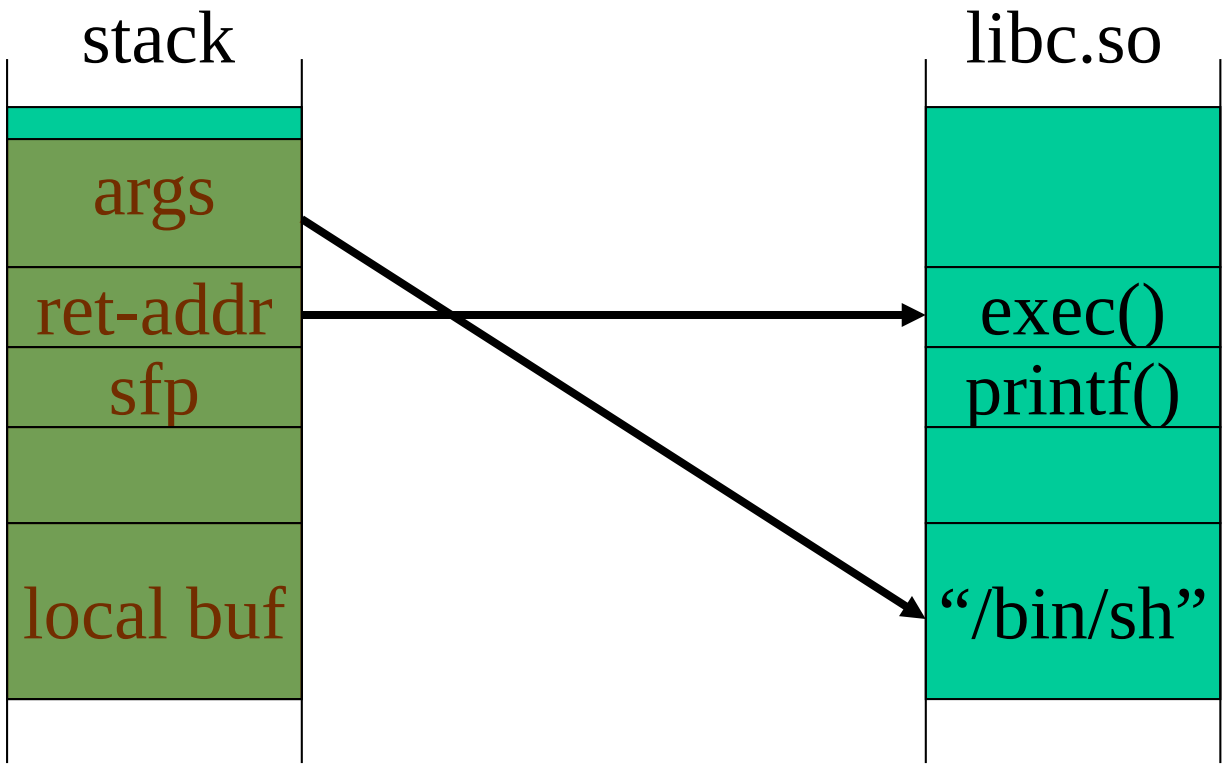
Computer > right-click Properties > Advanced system settings > Performance (Settings) > Data Execution Prevention



DEP terminating a program

“return to libc” attack

- Control hijacking without executing code



Defense against “return to libc”: randomization

- **ASLR**: (Address Space Layout Randomization)
  - Map shared libraries to random location in process memory
    ⇒ Attacker cannot jump directly to exec function
  - Deployment on 32-bit OS:
    - **Windows** Vista: 8 bits of randomness for DLLs
      - aligned to 64K page in a 16MB region ⇒ 256 choices
    - **Linux** (via PaX): 16 bits of randomness for libraries
  - More effective on 64-bit architectures

- Other randomization methods:
  - System-call randomization: randomize system-call IDs
  - Instruction Set Randomization (ISR)

Generally: Software Diversity

ASLR Example

Booting twice loads libraries into different locations:

ntlanman.dll	0x6D7F0000	Microsoft® Lan Manager
ntmarta.dll	0x75370000	Windows NT MARTA provider
ntshrui.dll	0x6F2C0000	Shell extensions for sharing
ole32.dll	0x76160000	Microsoft OLE for Windows

ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

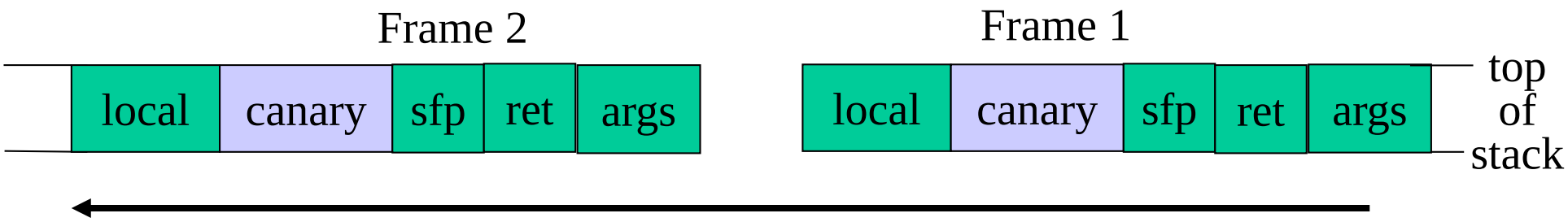
Control Hijacking

Runtime defense

Run time checking: StackGuard

- Many run-time checking techniques …
  - we only discuss methods relevant to overflow protection

- Solution 1: StackGuard
  - Run time tests for stack integrity.
  - Embed "canaries" in stack frames and verify their integrity prior to function return.



Canary Types

- Random canary:
  - Random string chosen at program startup.
  - Insert canary string into every stack frame.
  - Verify canary before returning from function.
    - Exit program if canary changed. Turns potential exploit into DoS.
  - To corrupt, attacker must learn current random string.

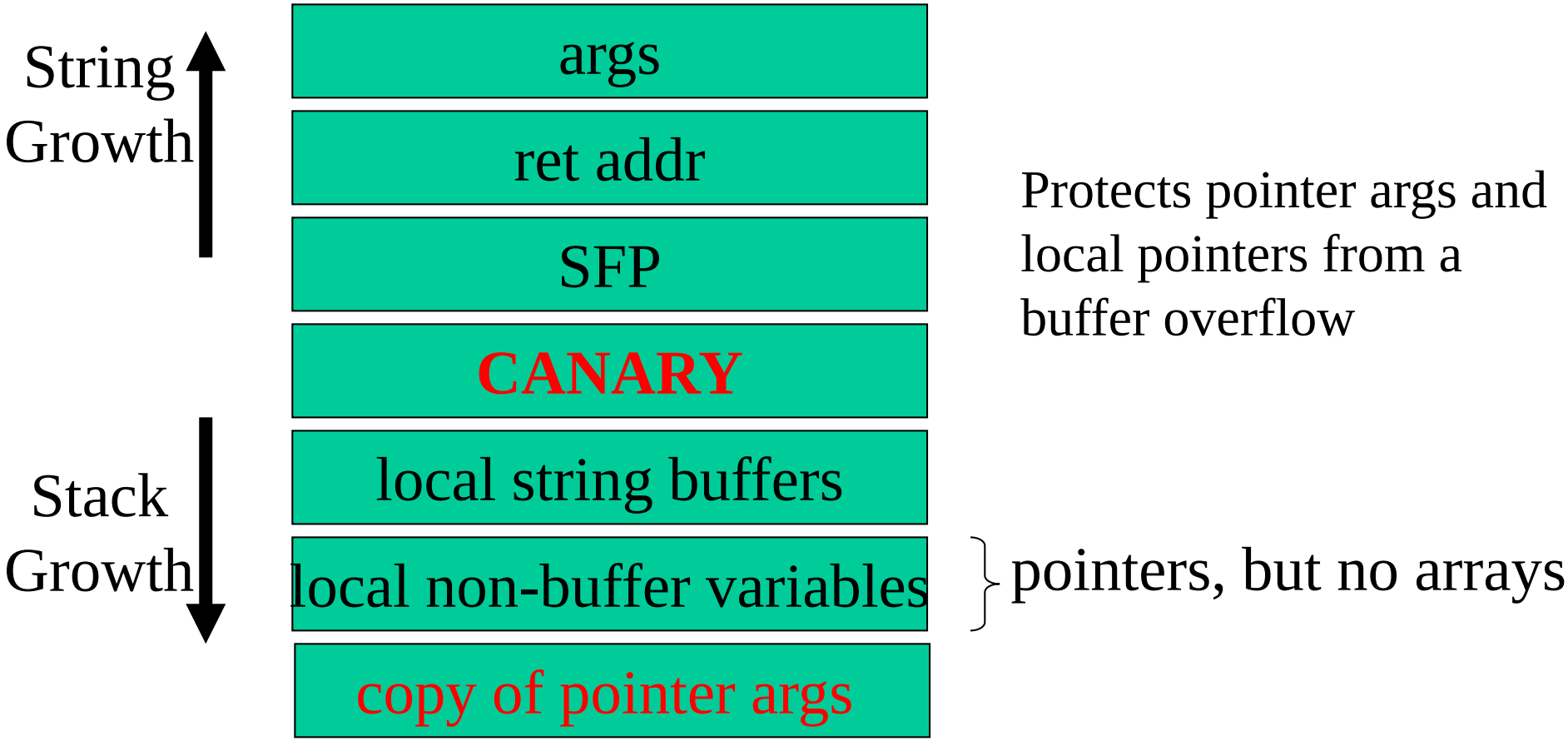
- Terminator canary: Canary = {0, newline, linefeed, EOF}
  - String functions will not copy beyond terminator.
  - Attacker cannot use string functions to corrupt stack.

StackGuard (Cont.)

- StackGuard implemented as a GCC patch.
  - Program must be recompiled.
- Small performance effects: (8% for Apache)
- Note: Canaries don’t provide full proof protection.
  - Some stack smashing attacks leave canaries unchanged

StackGuard enhancements: ProPolice

- ProPolice (IBM) - gcc 3.4.1. (**-fstack-protector**)
  - Rearrange stack layout to prevent overflow into pointers.



MS Visual Studio /GS [since 2003]

Compiler /GS option:

- Combination of ProPolice and Random canary.
- If cookie mismatch, default behavior is to call **_exit(3)**

Function prolog:

```
sub   esp, 8      // allocate 8 bytes for cookie
mov   eax, DWORD PTR ___security_cookie
xor   eax, esp    // xor cookie with current esp
mov   DWORD PTR [esp+8], eax  // save in stack
```

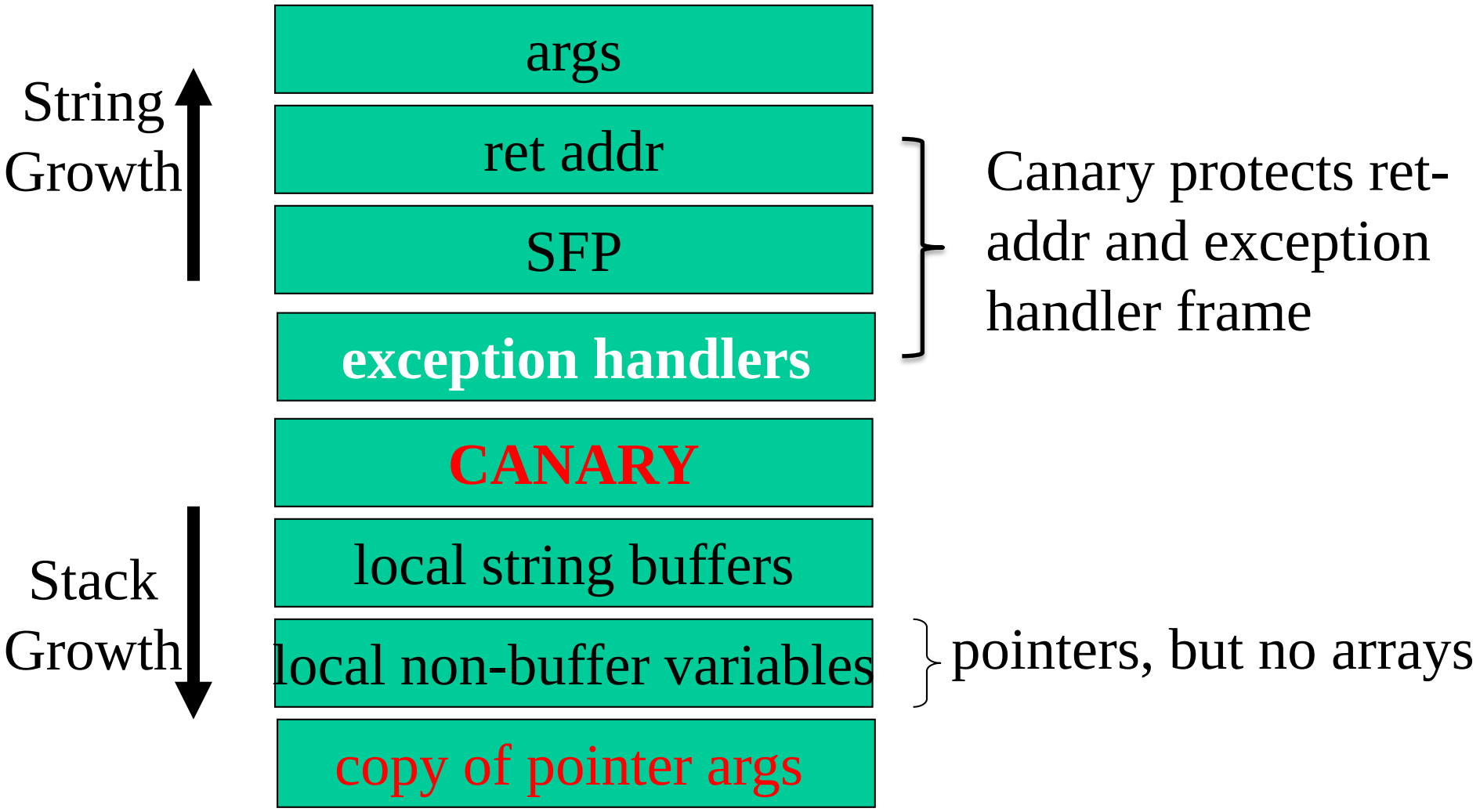
Function epilog:

```
mov   ecx, DWORD PTR  [esp+8]
xor   ecx, esp
call  @__security_check_cookie@4
add   esp, 8
```

Enhanced /GS in Visual Studio 2010:

- /GS protection added to all functions, unless can be proven unnecessary

/GS stack frame

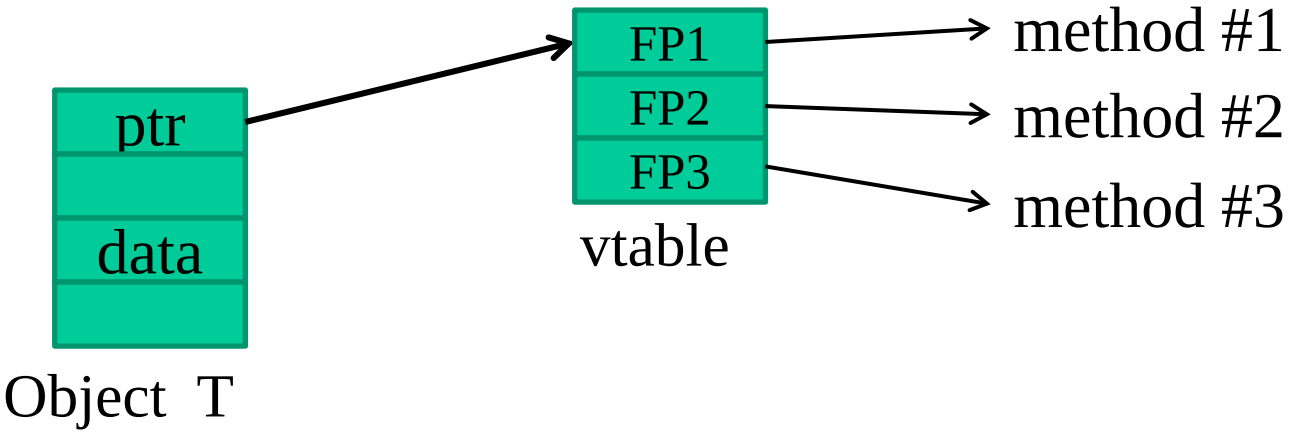


Control Hijacking

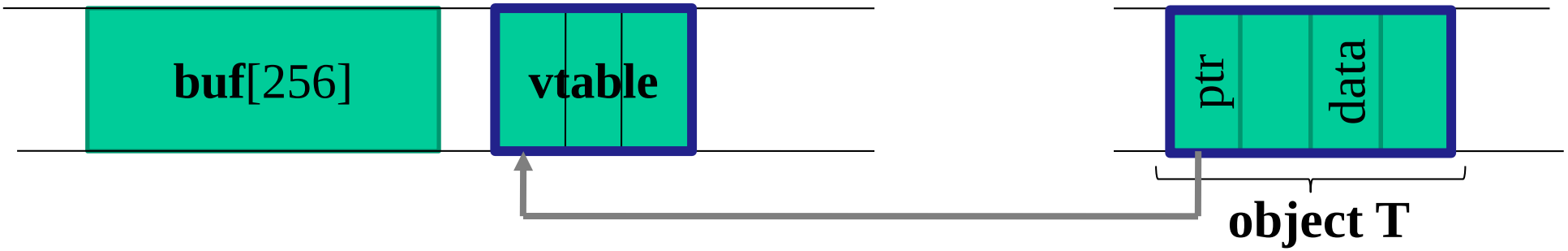
Heap spray attacks

Heap-based control hijacking

- Compiler generated function pointers (e.g. C++ code)

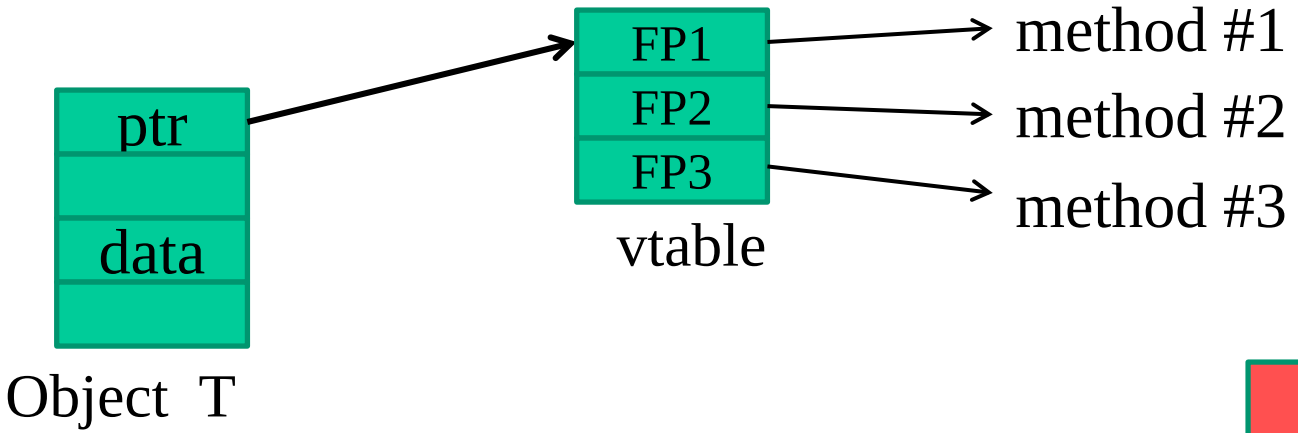


- Suppose vtable is on the heap next to a string object:

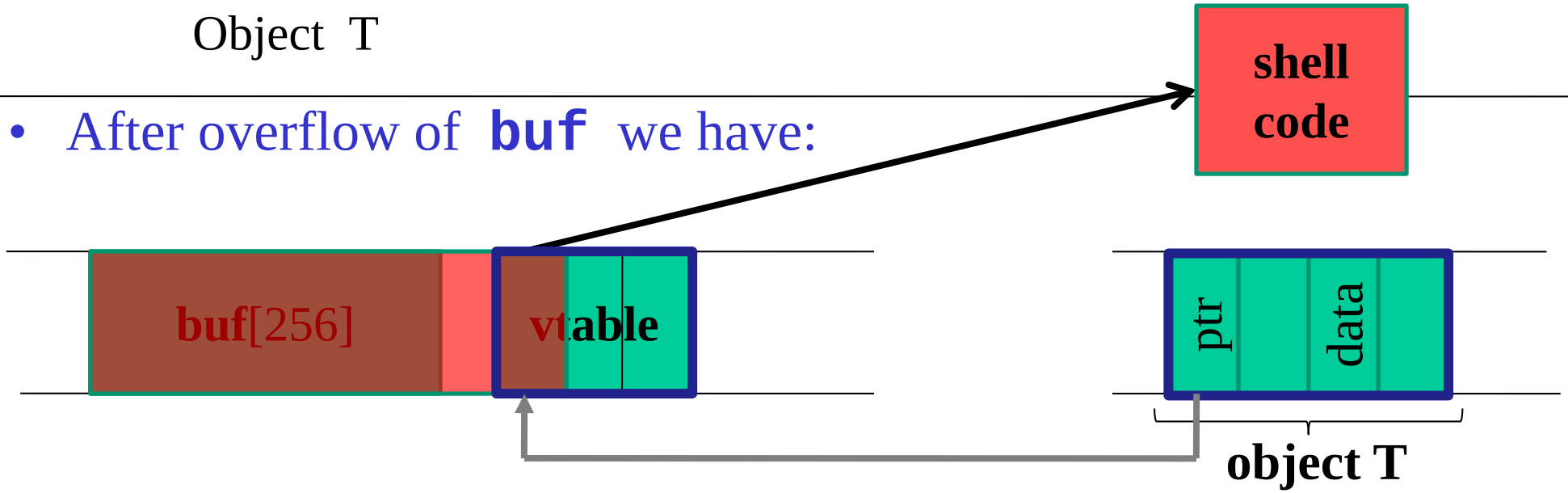


Heap-based control hijacking

- Compiler generated function pointers (e.g. C++ code)



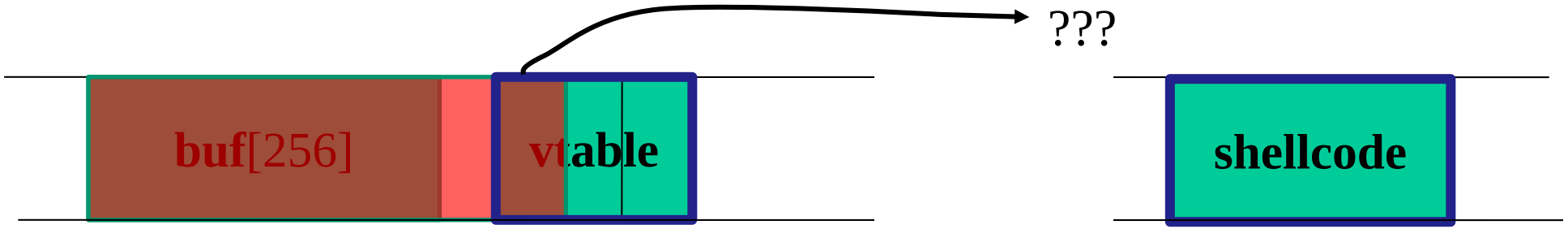
- After overflow of **`buf`** we have:



A reliable exploit?

```
<SCRIPT language="text/javascript">
 shellcode = unescape("%u4343%u4343%...");
overflow-string = unescape(“%u2332%u4276%...”);
cause-overflow( overflow-string );       // overflow  buf[ ]
</SCRIPT>
```

Problem: attacker does not know where browser places **shellcode** on the heap

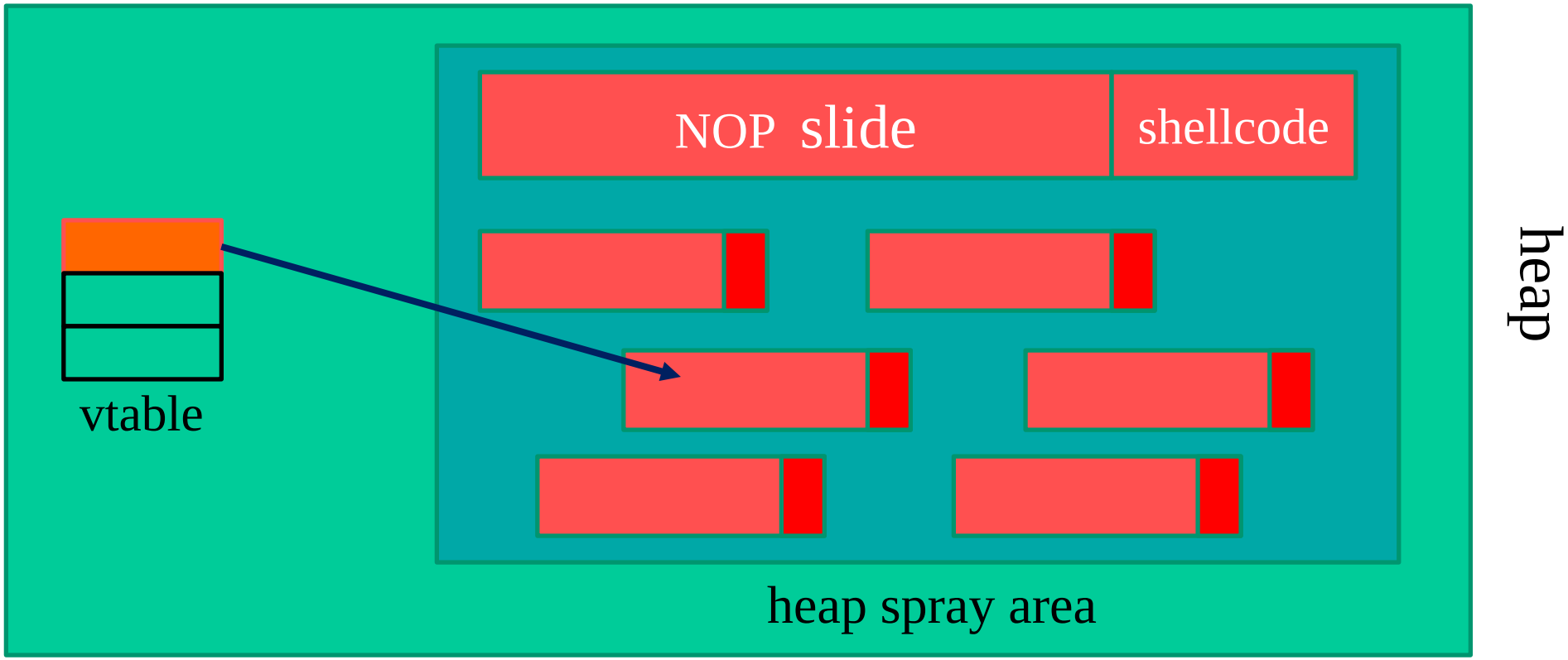


Heap Spraying [SkyLined 2004]

Idea:

1. use Javascript to spray heap with shellcode (and NOP slides)

2. then point vtable ptr anywhere in spray area



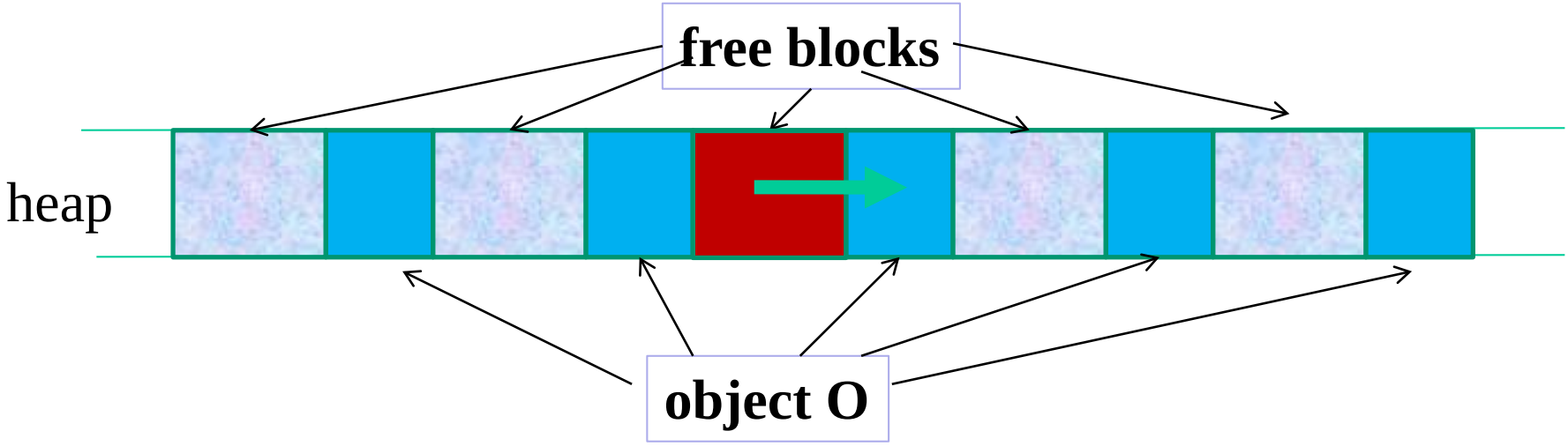
JavaScript heap spraying

```
var  nop = unescape(“%u9090%u9090”)
while (nop.length < 0x100000)  nop += nop

var shellcode = unescape("%u4343%u4343%...");

var x = new Array ()
for (i=0;  i<1000;  i++) {
      x[i] = nop + shellcode;
}
```

- Pointing func-ptr almost anywhere in heap will cause shellcode to execute.

Vulnerable buffer placement

- Placing vulnerable **`buf[256]`** next to object O:
  - By sequence of JavaScript allocations and frees make heap look as follows:



  - Allocate vulnerable buffer in Javascript and cause overflow
  - Successfully used against a Safari PCRE overflow [DHM'08]

Many heap spray exploits

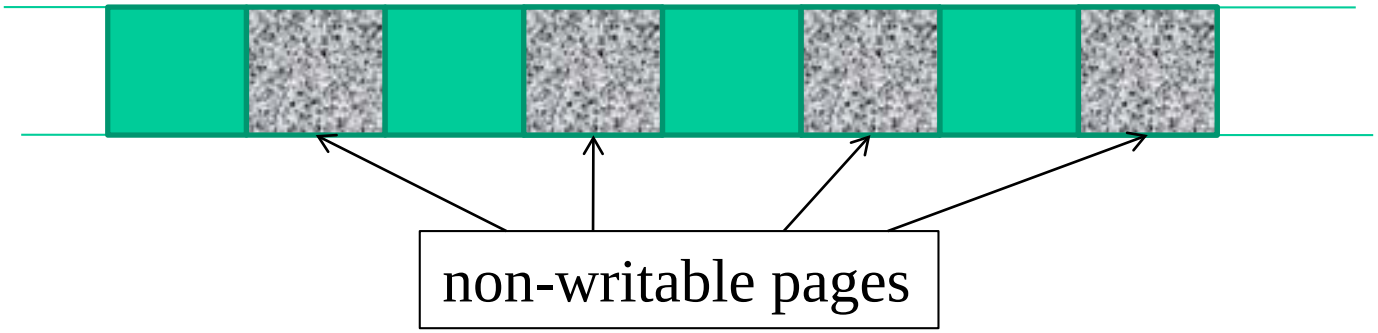
Date	Browser	Description
11/2004	IE	IFRAME Tag BO
04/2005	IE	DHTML Objects Corruption
01/2005	IE	.ANI Remote Stack BO
07/2005	IE	`javaprxy.dll` COM Object
03/2006	IE	`createTextRang` RE
09/2006	IE	VML Remote BO
03/2007	IE	ADODB Double Free
09/2006	IE	WebViewFolderIcon `setSlice`
09/2005	FF	0xAD Remote Heap BO
12/2005	FF	`compareTo()` RE
07/2006	FF	`Navigator` Object RE
07/2008	Safari	Quicktime Content-Type BO

[RLZ'08]

- Improvements: Heap Feng Shui [S'07]
  - Reliable heap exploits **on IE** without spraying
  - Gives attacker full control of IE heap from Javascript

(partial) Defenses

- Protect heap function pointers (e.g. PointGuard)
- Better browser architecture:
  - Store JavaScript strings in a separate heap from browser heap
- OpenBSD heap overflow protection:

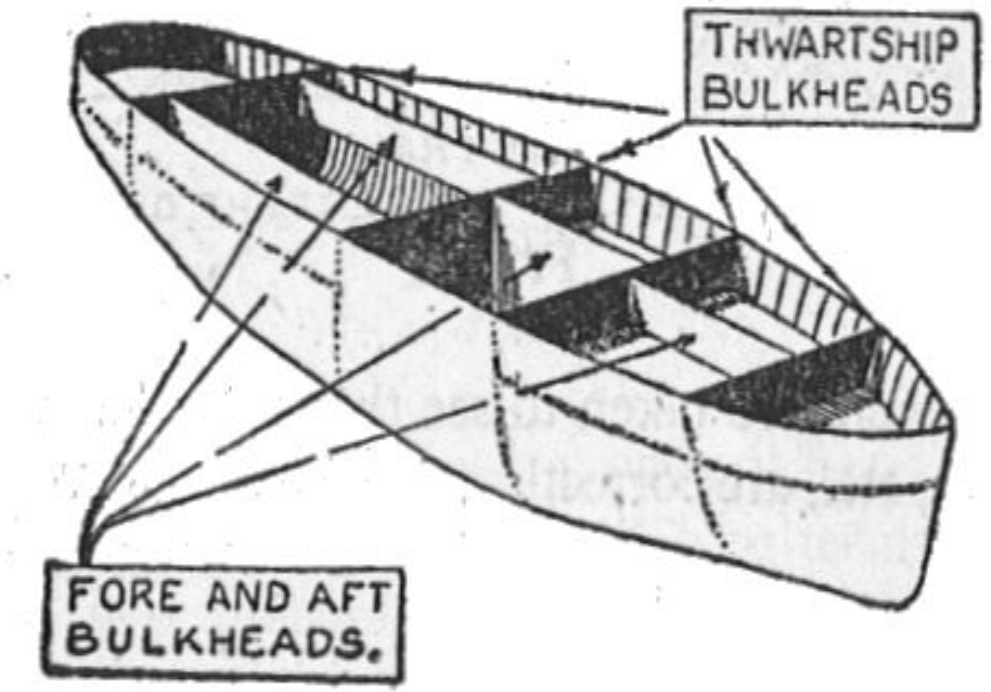


prevents cross-page overflows

- Nozzle [RLZ'08] : detect sprays by prevalence of code on heap

Secure Architecture Principles: Isolation and Least Privilege

Basic idea: Isolation



A Seaman's Pocket-Book, 1943 (public domain)

Principles of Secure Design

- Compartmentalization
  - Isolation
  - Principle of least privilege
- Defense in depth
  - Use more than one security mechanism
  - Secure the weakest link
  - Fail securely
- Keep it simple

Principle of Least Privilege

- Privilege
  - Ability to access or modify a resource
- Principle of Least Privilege
  - A system module should only have the minimal privileges needed for intended purposes
- Requires compartmentalization and isolation
  - Separate the system into independent modules
  - Limit interaction between modules

Example: Android process isolation

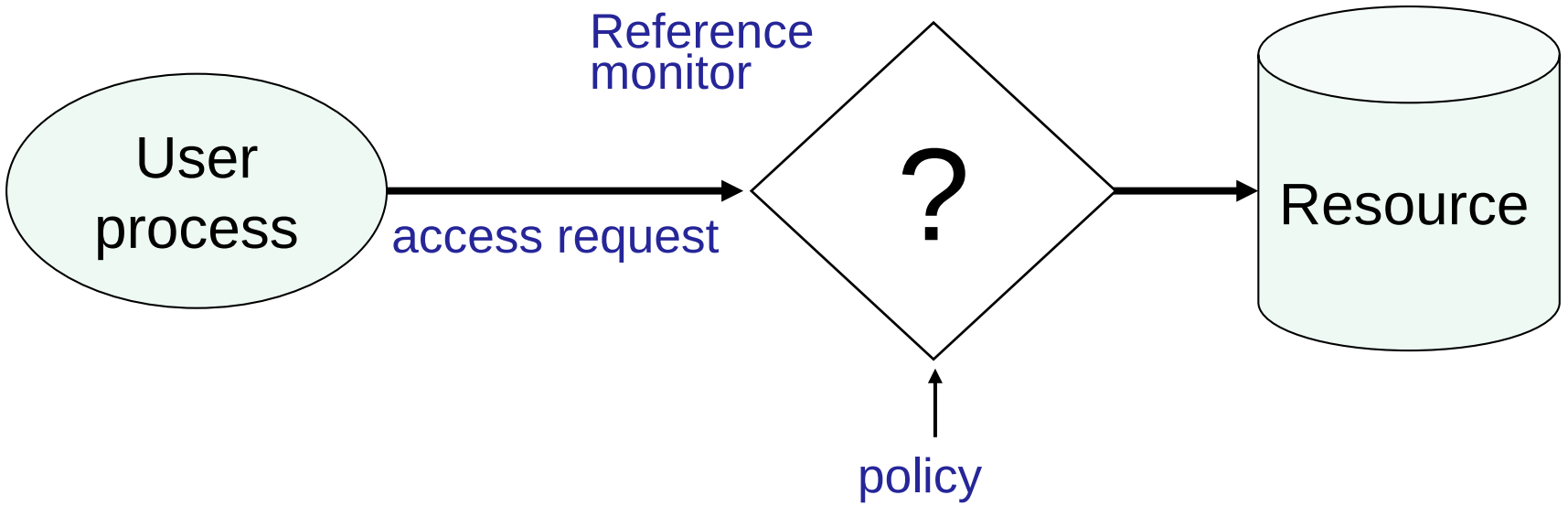
- Android application sandbox
  - Isolation: Each application runs with its own UID in own VM
    - Provides memory protection
    - Communication protected using Unix domain sockets
    - Only ping, zygote (spawn another process) run as root
  - Interaction: reference monitor checks permissions on inter-component communication
  - Least Privilege: Applications announces permission
    - Whitelist model – user grants access
      - Questions asked at install time, to reduce user interruption

Access control

Concepts

Access control

- Assumptions
  - System knows who the user is
    - Authentication via name and password, other credential
  - Access requests pass through gatekeeper (reference monitor)
    - System must not allow monitor to be bypassed



Access control matrix

[Lampson]

Objects

files, sockets, resources

Subjects

users, processes

	File 1	File 2	File 3	...	File n
User 1	read	write	-	-	read
User 2	write	write	write	-	-
User 3	-	-	-	read	read
...					
User m	read	write	read	write	read

Two implementation concepts

- Access control list (ACL)
  - Store column of matrix with the resource
- Capability
  - User holds a “ticket” for each resource
  - Two variations
    - store row of matrix with user, under OS control
    - unforgeable ticket in user space

	File 1	File 2	...
User 1	read	write	-
User 2	write	write	-
User 3	-	-	read
...			
User m	Read	write	write

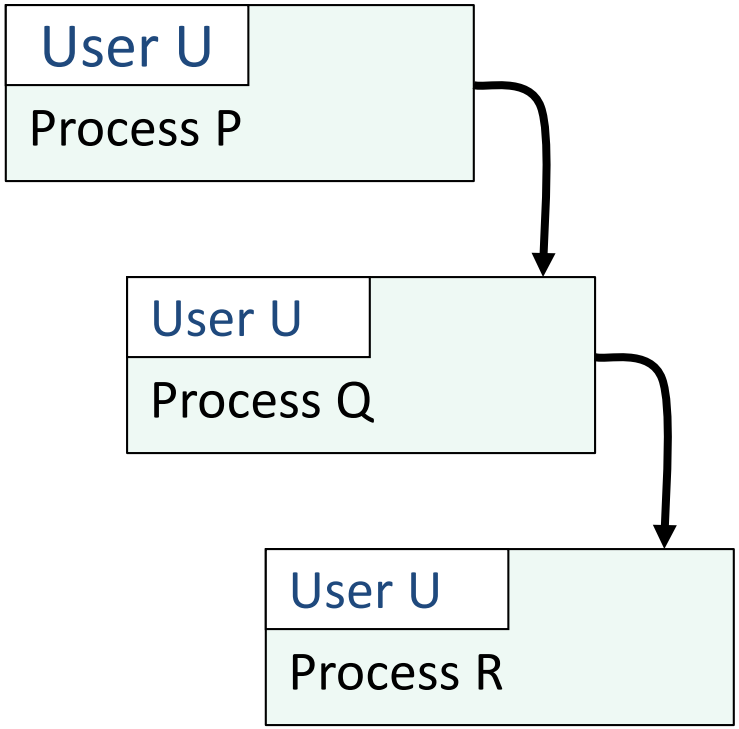
Access control lists are widely used, often with groups

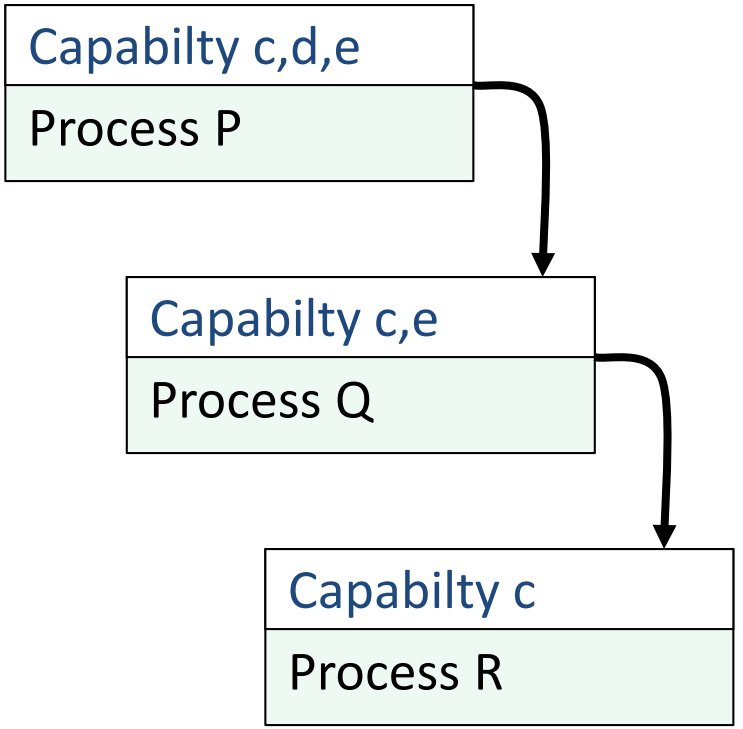
Some aspects of capability concept are used in many systems

ACL vs Capabilities

- Access control list
  - Associate list with each object
  - Check user/group against list
  - Relies on authentication: need to know user
- Capabilities
  - Capability is unforgeable ticket
    - Random bit sequence, or managed by OS
    - Can be passed from one process to another
  - Reference monitor checks ticket
    - Does not need to know identify of user/process

ACL vs Capabilities



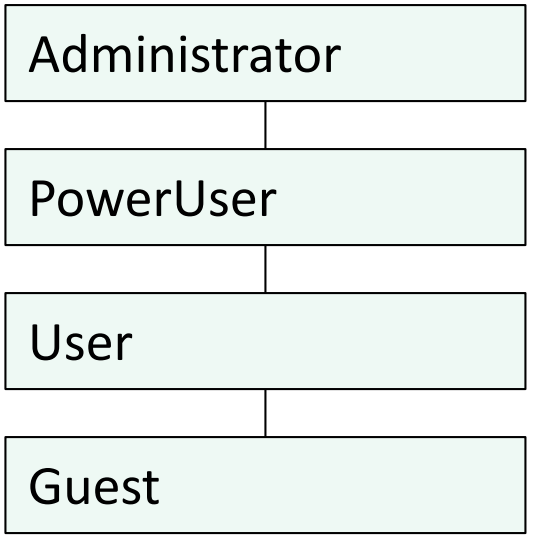


ACL vs Capabilities

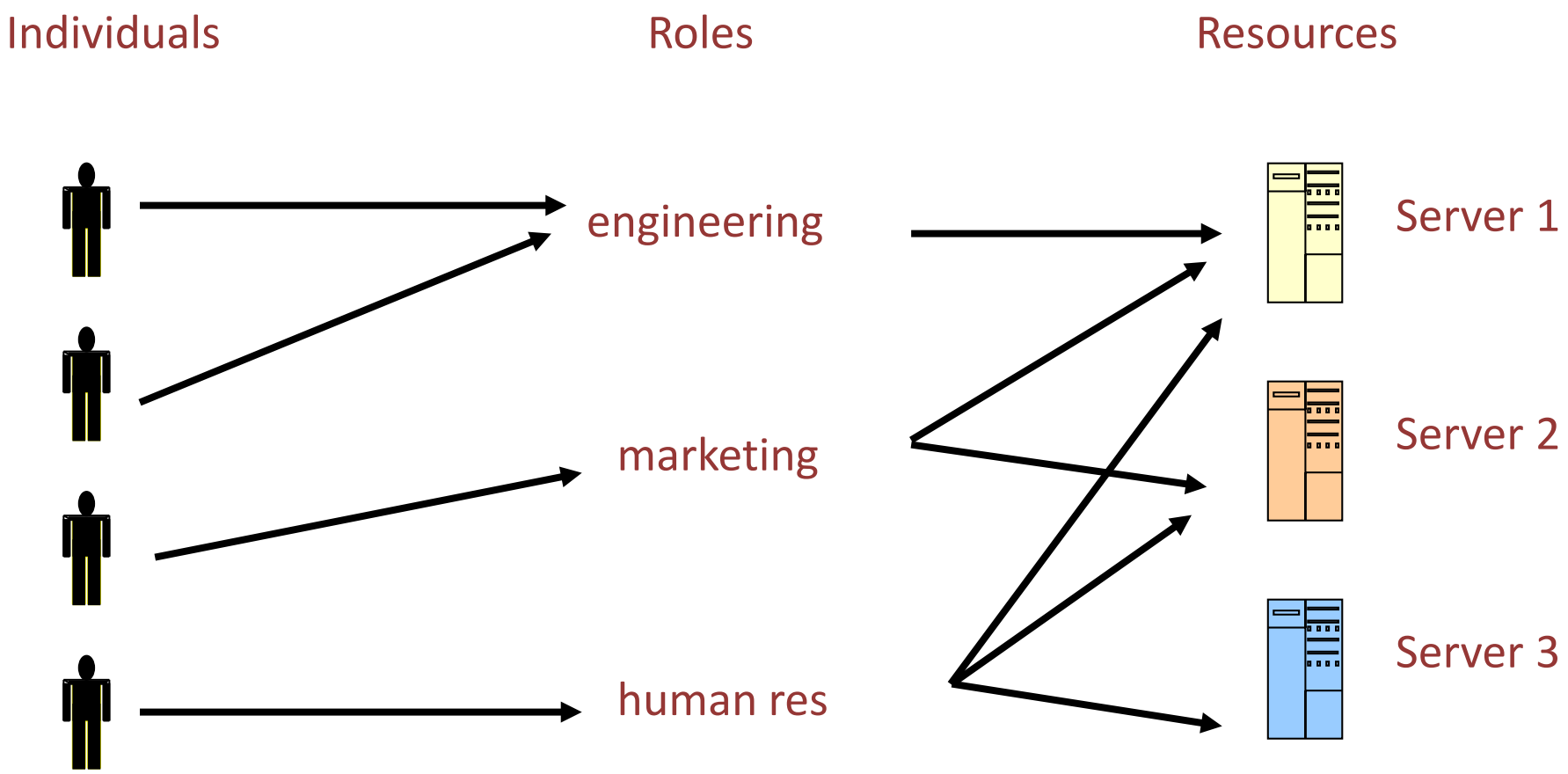
- Delegation
  - Cap: Process can pass capability at run time
  - ACL: Try to get owner to add permission to list?
    - More common: let other process act under current user
- Revocation
  - ACL: Remove user or group from list
  - Cap: Try to get capability back from process?
    - Possible in some systems if appropriate bookkeeping
      - OS knows which data is capability
      - If capability is used for multiple resources, have to revoke all or none …
    - Indirection: capability points to pointer to resource
      - If C $\rightarrow$ P $\rightarrow$ R, then revoke capability C by setting P=0

Roles (also called Groups)

- Role = set of users
  - Administrator, PowerUser, User, Guest
  - Assign permissions to roles; each user gets permission
- Role hierarchy
  - Partial order of roles
  - Each role gets permissions of roles below
  - List only new permissions given to each role



Role-Based Access Control



Advantage: users change more frequently than roles

Access control

Unix

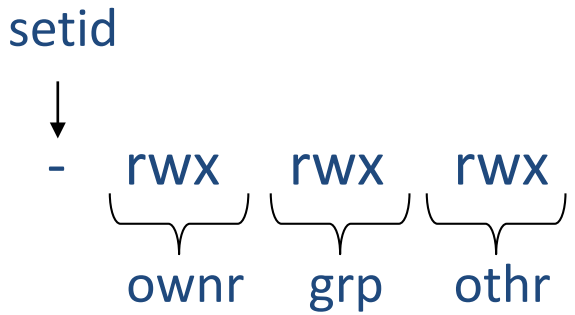
Unix access control

- File has access control list (ACL)
  - Grants permission to user ids
  - Owner, group, other
- Process has user id
  - Inherit from creating process
  - Process can change id
    - Restricted set of options
  - Special “root” id
    - Bypass access control restrictions

	File 1	File 2	...
User 1	read	write	-
User 2	write	write	-
User 3	-	-	read
...			
User m	Read	write	write

Unix file access control list

- Each file has owner and group
- Permissions set by owner
  - Read, write, execute
  - Owner, group, other
  - Represented by vector of four octal values



    0755 (rwxr-xr-x) → public directory or executable
    0644 (rw-r--r--) → public file
    0600 (rw-------) → private file
- Only owner, root can change permissions
  - This privilege cannot be delegated or shared
- Setid bits – Discuss in a few slides

Question

- Owner can have fewer privileges than other
  - What happens?
    - Owner gets access?
    - Owner does not?

Prioritized resolution of differences

if user = owner then owner permission
else if user in group then group permission
else other permission

Process effective user id (EUID)

- Each process has three Ids (+ more under Linux)
  - Real user ID (RUID)
    - same as the user ID of parent (unless changed)
    - used to determine which user started the process
  - Effective user ID (EUID)
    - from set user ID bit on the file being executed, or sys call
    - determines the permissions for process
      - file access and port binding
  - Saved user ID (SUID)
    - So previous EUID can be restored
- Real group ID, effective group ID, used similarly

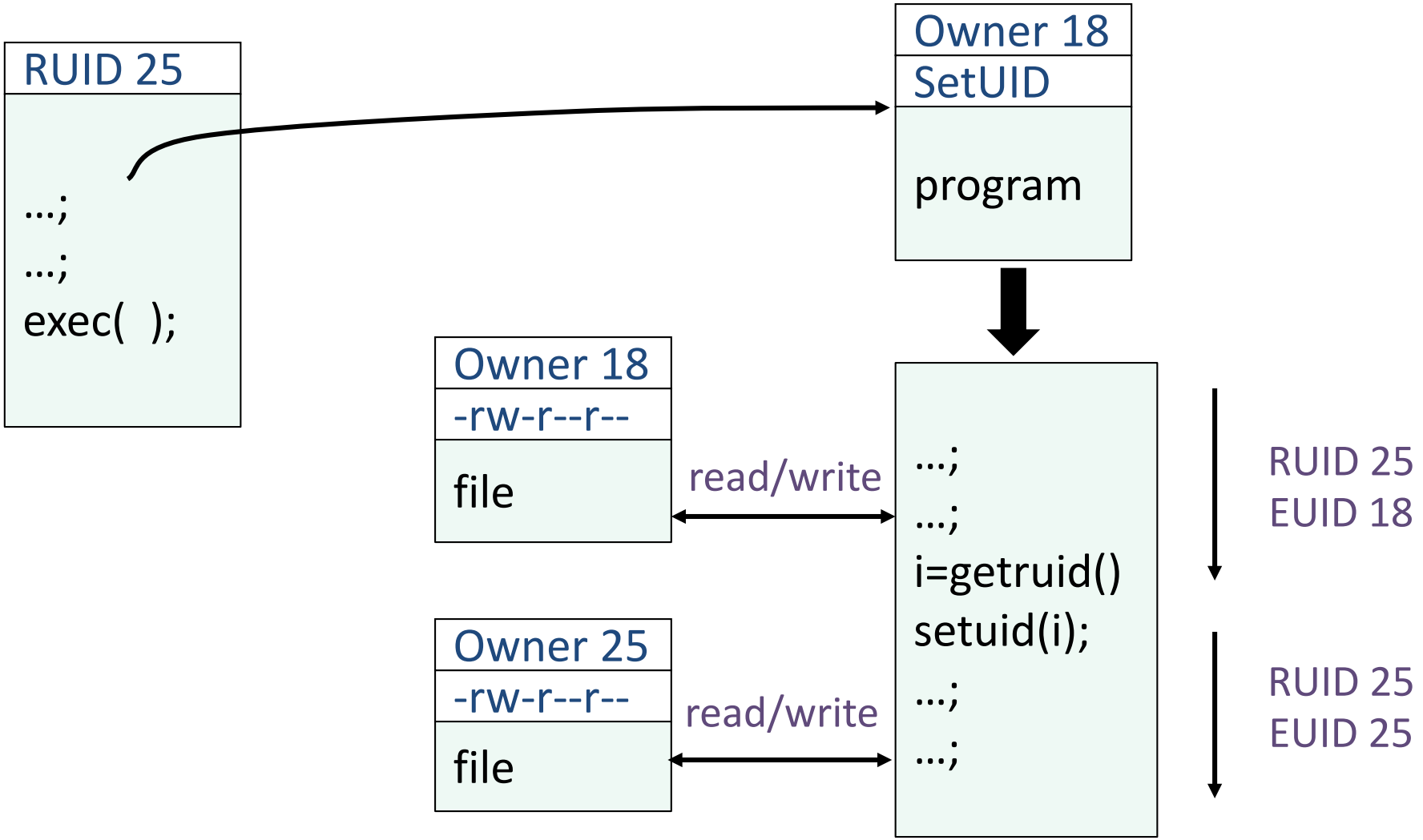
Process Operations and IDs

- Root
  - ID=0 for superuser root; can access any file
- Fork and Exec
  - Inherit three IDs, except exec of file with setuid bit
- Setuid system calls
  - seteuid(newid) can set EUID to
    - Real ID or saved ID, regardless of current EUID
    - Any ID, if EUID=0
- Details are actually more complicated
  - Several different calls: setuid, seteuid, setreuid

setid bits on executable Unix file

- Three setid bits
  - Setuid – set EUID of process to ID of file owner
  - Setgid – set EGID of process to GID of file
  - Sticky
    - Off: if user has write permission on directory, can rename or remove files, even if not owner
    - On: only file owner, directory owner, and root can rename or remove file in the directory

setuid example



setuid programming

- Be Careful with Setuid 0 !
  - Root can do anything; don’t get tricked
  - Principle of least privilege – change EUID when root privileges no longer needed