Low-level software vulnerability protection mechanisms

CS642: Computer Security

Professor Ristenpart

http://www.cs.wisc.edu/~rist/

rist at cs dot wisc dot edu

How can we help prevent exploitation of buffer overflows and other control flow hijacking?

Non-executable memory pages

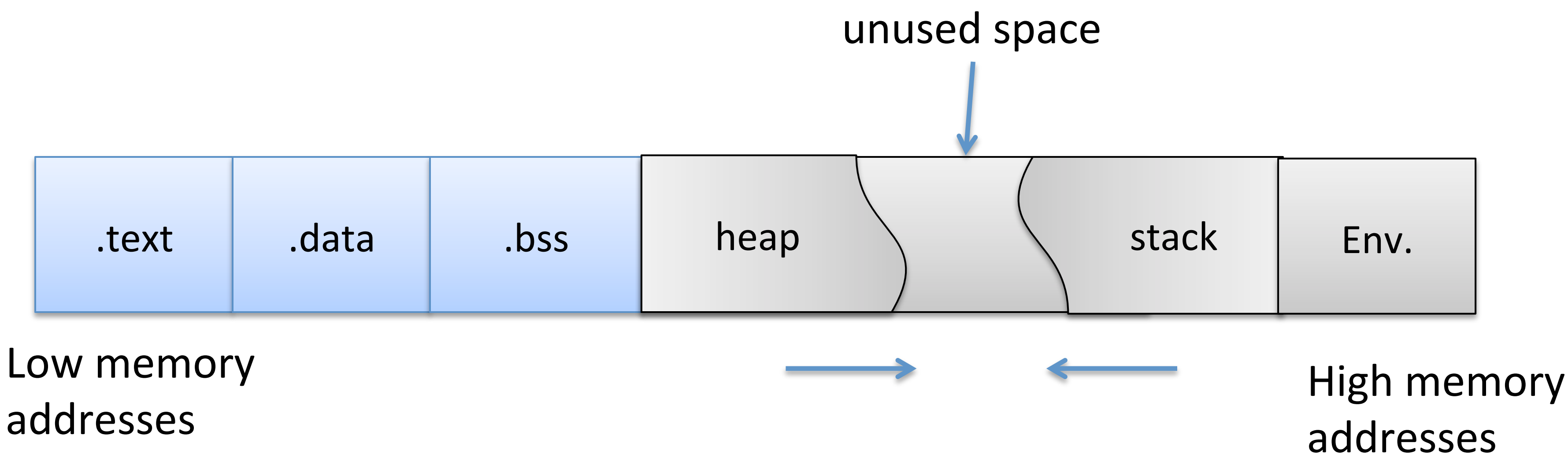
Return-into-libc exploits, Return-oriented programming

Address space layout randomization

StackGuard, StackShield

Software fault isolation

Process memory layout



.text:
- machine code of executable

.data:
- global initialized variables

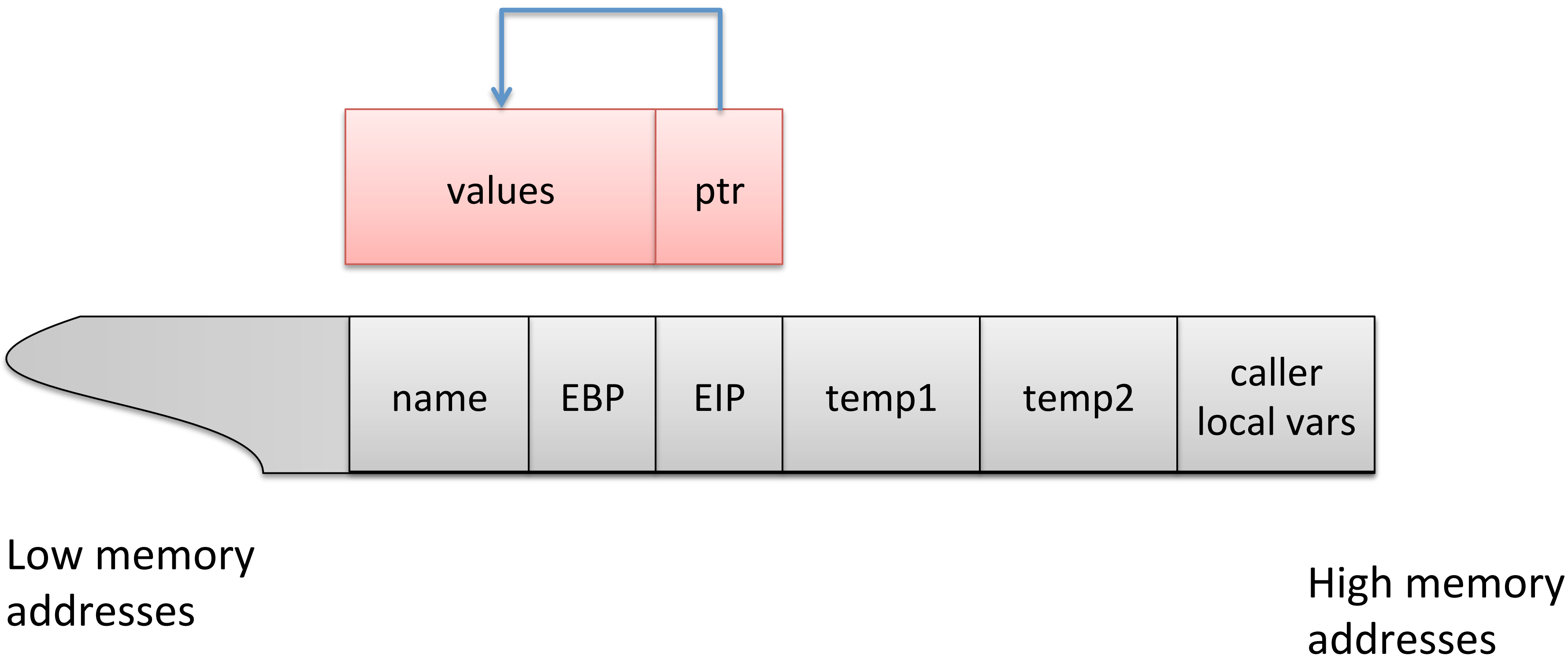
.bss:
- “below stack section”
- global uninitialized variables

heap:
- dynamic variables

stack:
- local variables, track func calls

Env:
- environment variables,
- arguments to program

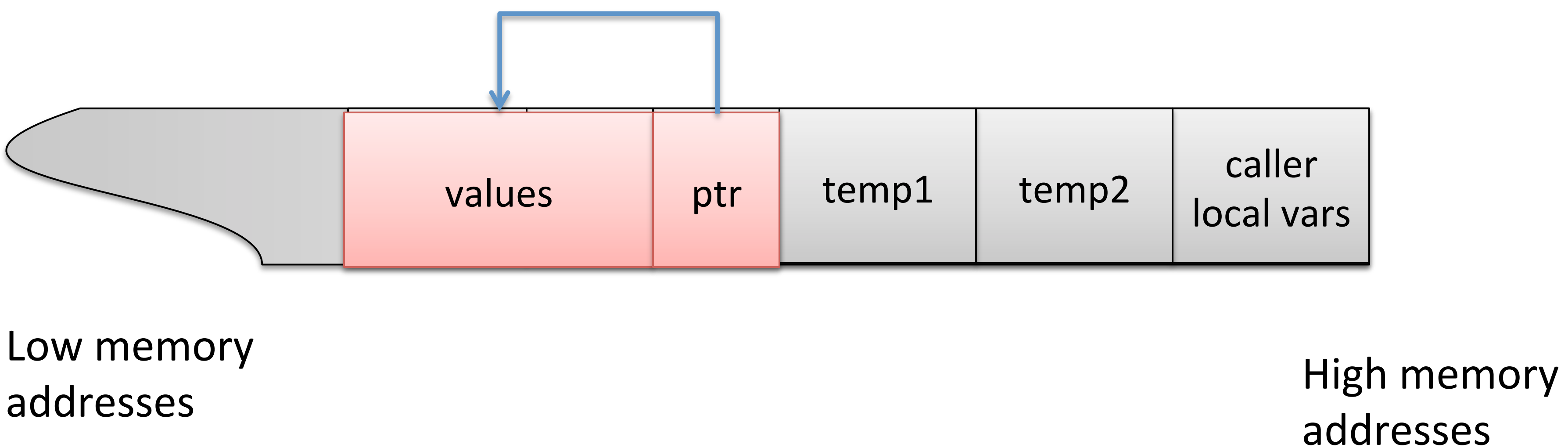
Typical return ptr overwrite exploit



Why should the machine interpret stack data as instructions?

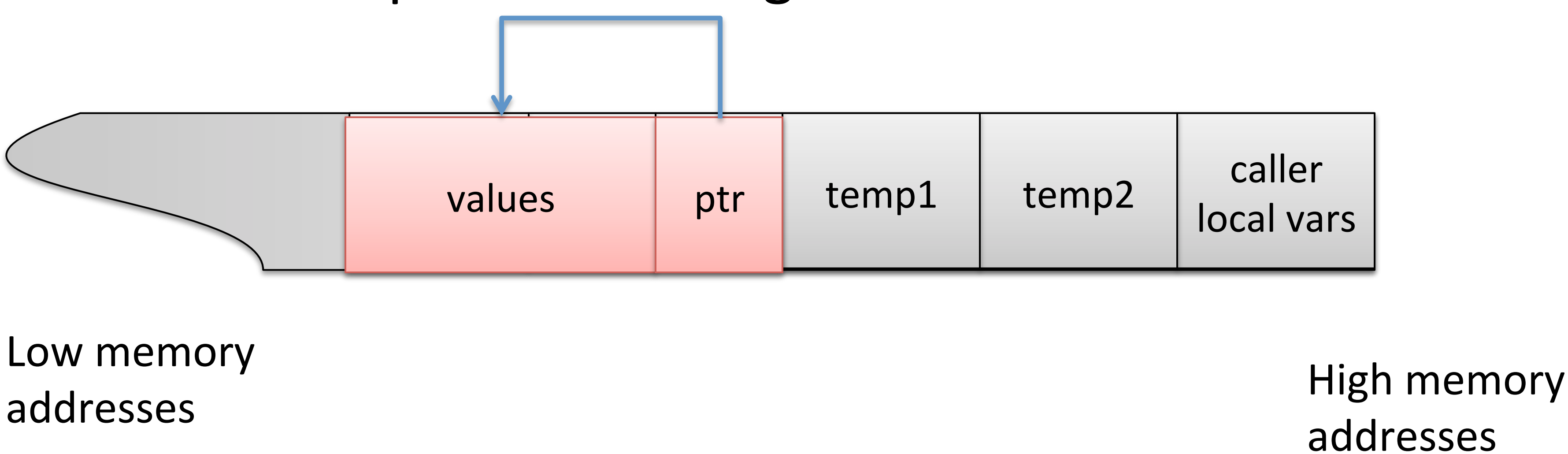
W^X (W xor X)

- The idea: mark memory page as either
  - Writable or Executable (not both)
- Specifically: make heap and stack non-executable



W^X (W xor X)

- AMD64: NX bit (Non-Executable)
  IA-64: XD bit (eXecute Disabled)
  ARMv6: XN bit (eXecute Never)
  - Extra bit in each page table entry
  - Processor refuses to execute code if bit = 1
  - Mark heap and stack segments as such



W^X (W xor X)

Software emulation of NX bits

- ExecShield (RedHat Linux)
- PaX (Page-eXec) (uses NX bit if available)

mprotect()

- Process can set permissions on memory pages

Will W^X stop:

AlephOne's stack overflow exploit?	Yes
Stack smash that overwrites pointer to point at shell code in Heap or Env variable?	Yes
Heap overflow with same shell location?	Yes
Double free with same shell location?	Yes

Limitations of W^X

Software emulation …

- May not be perfect and is slow
- E.g., double-free or format-string vulnerability may allow turning off protections

Breaking compatibility

- GCC stack trampolines (calling conventions, nested functions)
- Just-in-time (JIT) compilation using heap

Exploits designed to only run existing code

Return-into-libc exploits

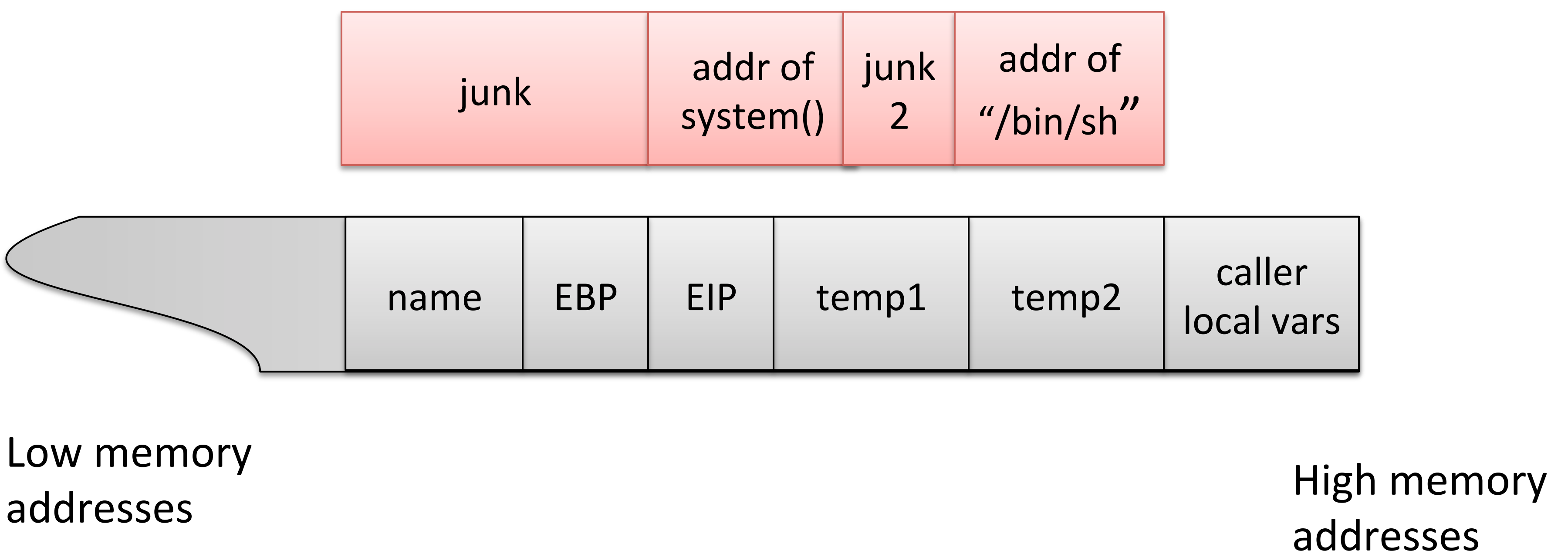
- libc is standard C library, included in all processes
- system() --- execute commands on system

```
(gdb) b main
Breakpoint 1 at 0x80484a0: file sploit1.c, line 15.
(gdb) r
Starting program: /home/user/pp1/sploits/sploit1

Breakpoint 1, main () at sploit1.c:15
15          args[0] = TARGET;
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7ecf180 <system>
(gdb) _
```

Return-into-libc exploits

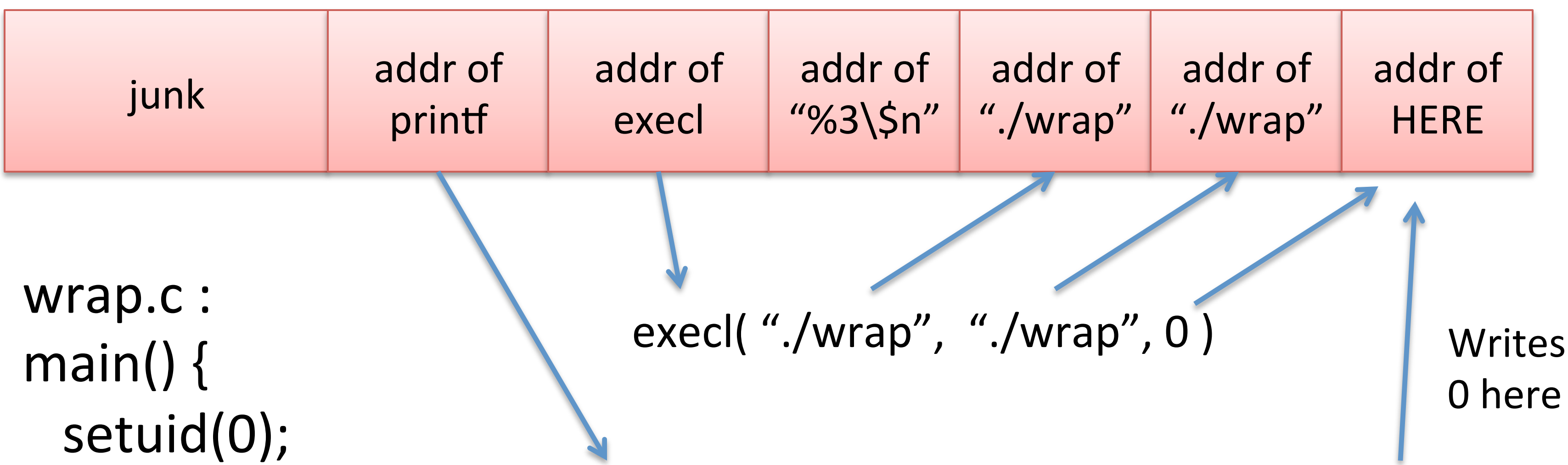
Overwrite EIP with address of system() function
junk2 just some filler: returned to after system call
first argument to system() is ptr to "/bin/sh"



Return-into-libc exploits

This simple exploit has a few deficiencies (from attacker's pov):

- Crashes after exiting called /bin/sh (easy to fix with exit())
- system() drops privileges by default



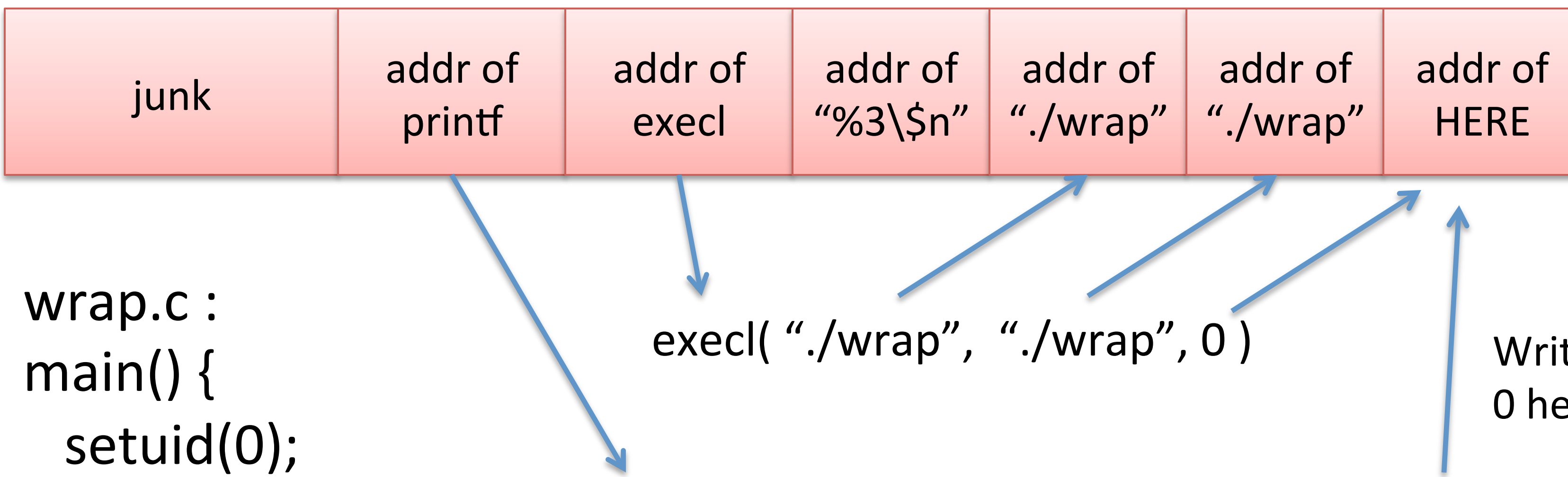

```
wrap.c :
main() {
  setuid(0);
  setgid(0);
  system("bin/sh");
}
```

printf("%3$n", …)
%3n means "write number of bytes in format string up to the format token into third parameter"

Return-into-libc exploits

These exploits only execute instructions marked executable

W^X cannot stop such an attack




```
wrap.c :
main() {
  setuid(0);
  setgid(0);
  system(“bin/sh”);
}
```

execl(“./wrap”, “./wrap”, 0)

Writes 0 here

printf(“%3$n”, …)

%3n means “write number of bytes in format string up to the format token into third parameter”

Return-into-libc exploits

Return-into-libc may seem limited:

- Only useful for calling libc functions
- Okay in last example, but not always sufficient
- Before W^X, exploit could run arbitrary code

Can we not inject any malicious code and yet have an exploit that runs arbitrary code?

Return-oriented programming (ROP)

Second return-into-libc exploit:
self-modifying exploit buffer to call a sequence of libc calls

Logical extreme:
chain together a long sequence of calls to code

But we want arbitrary code, not sequence of libc calls:
chain together a long sequence of calls to code snippets

Return-oriented programming (ROP)

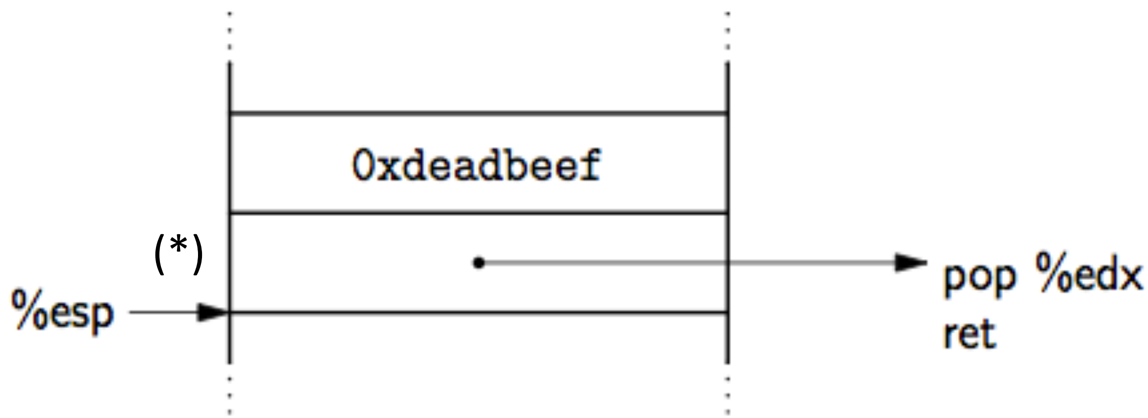


Figure 2: Load the constant 0xdeadbeef into %edx.

From Shacham “The Geometry of Innocent Flesh on the Bone…” 2007

If this is on stack and (*) is return pointer after buffer overflow, then the result will be loading 0xdeadbeef into edx register

Return-oriented programming (ROP)

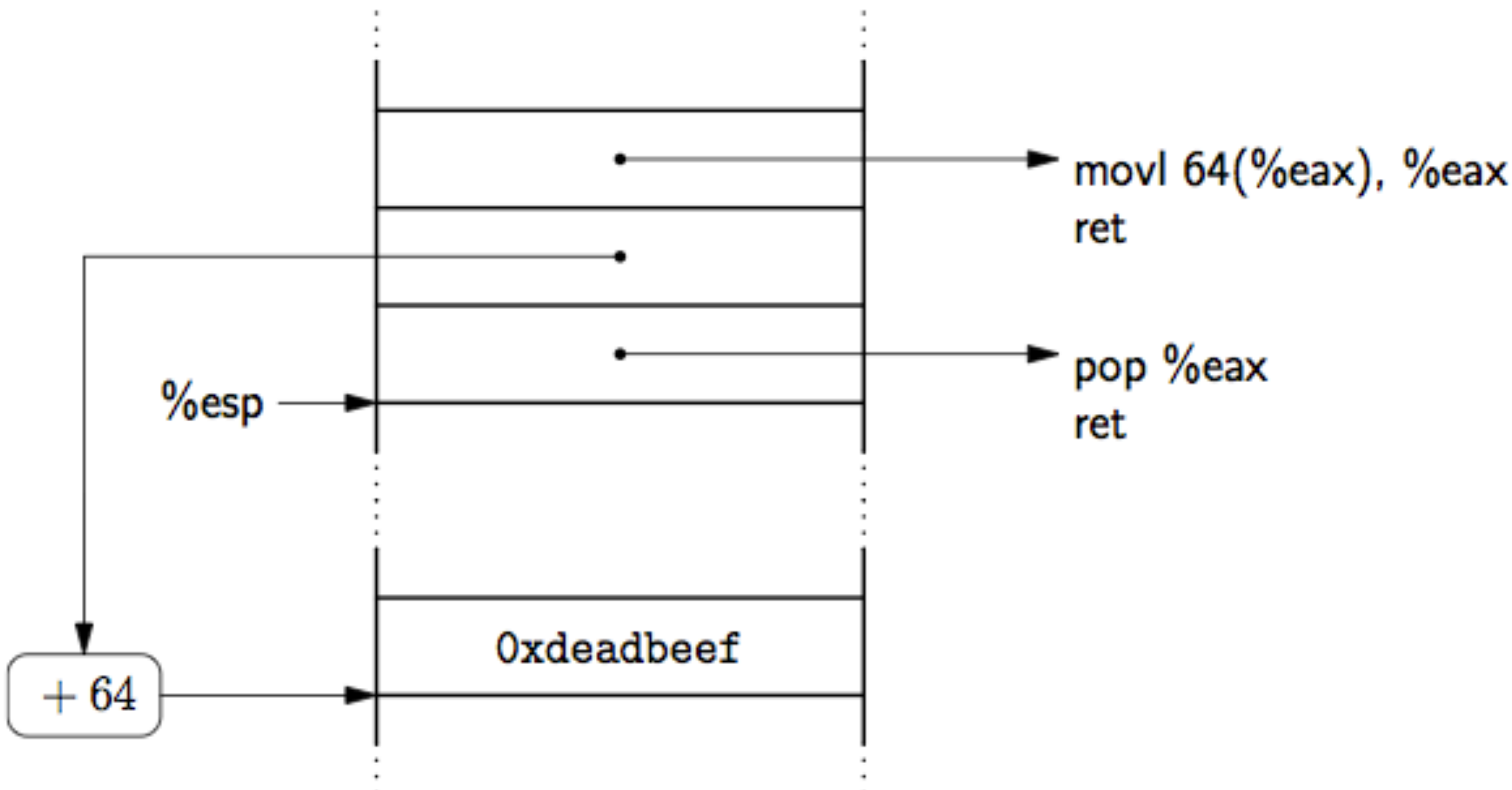


Figure 3: Load a word in memory into %eax.

From Shacham "The Geometry of Innocent Flesh on the Bone…" 2007

From
Shacham
"The Geometry of
Innocent Flesh on
the Bone…" 2007

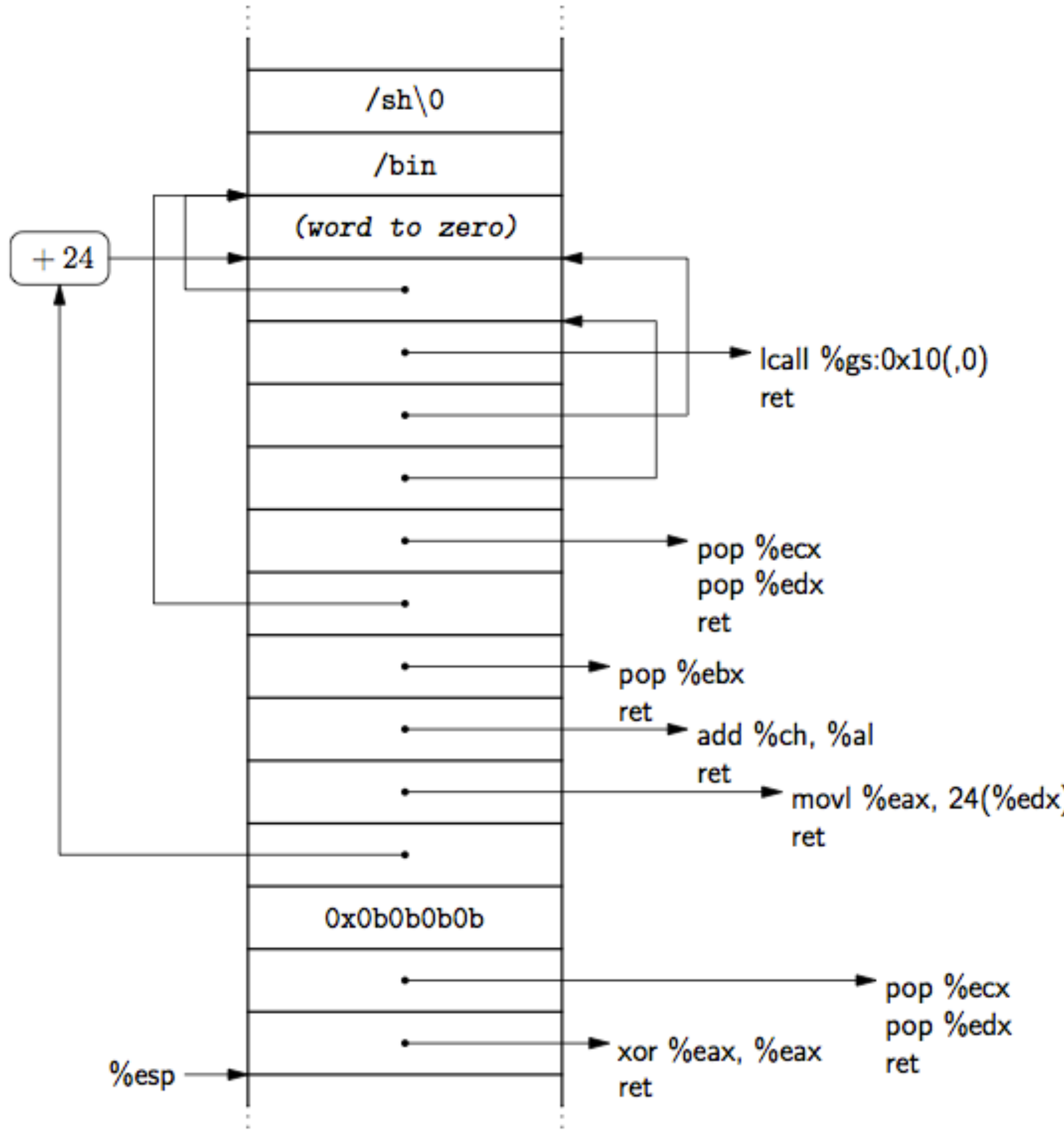
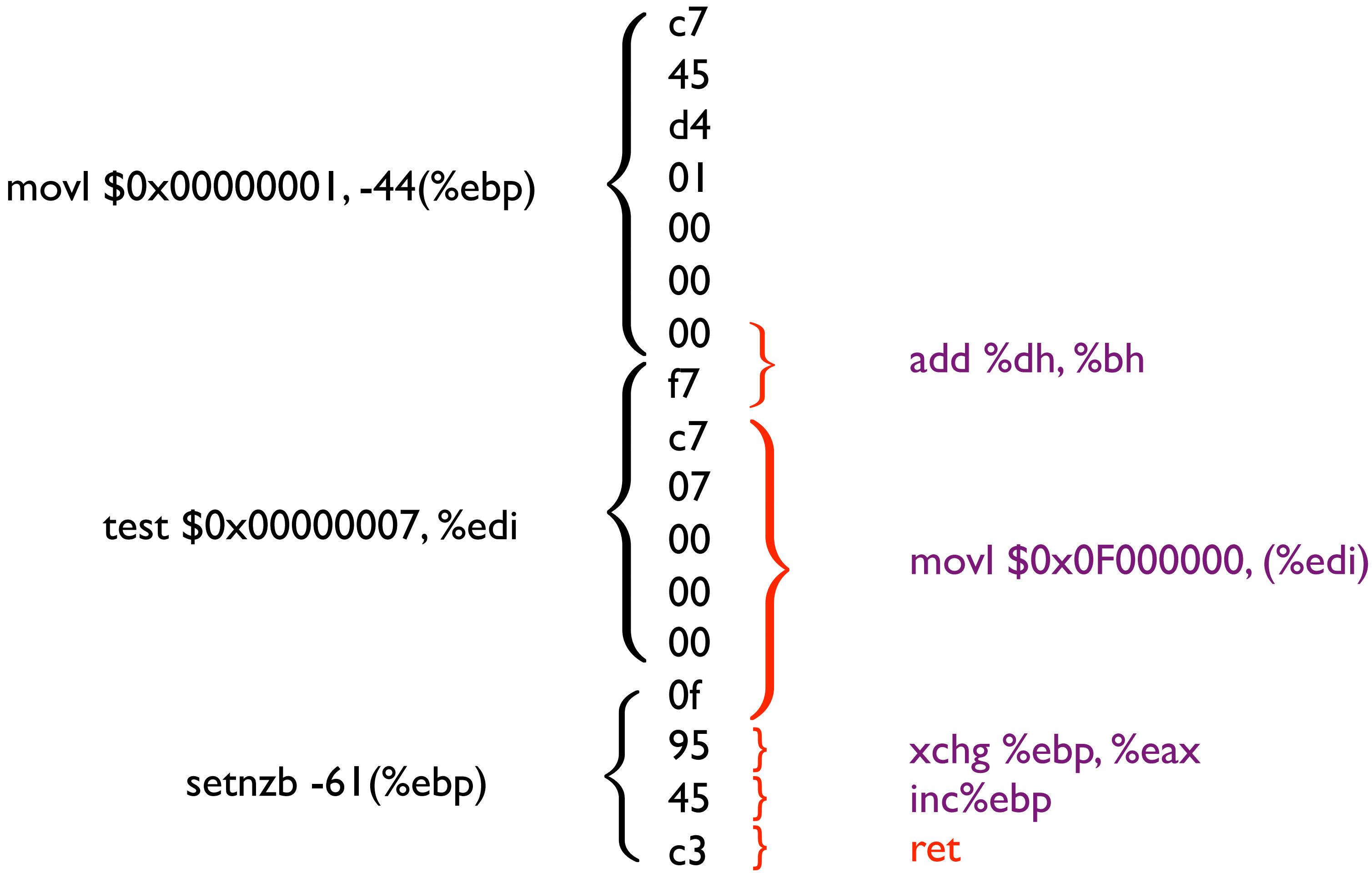


Figure 16: Shellcode.

ROP where do we get code snippets?



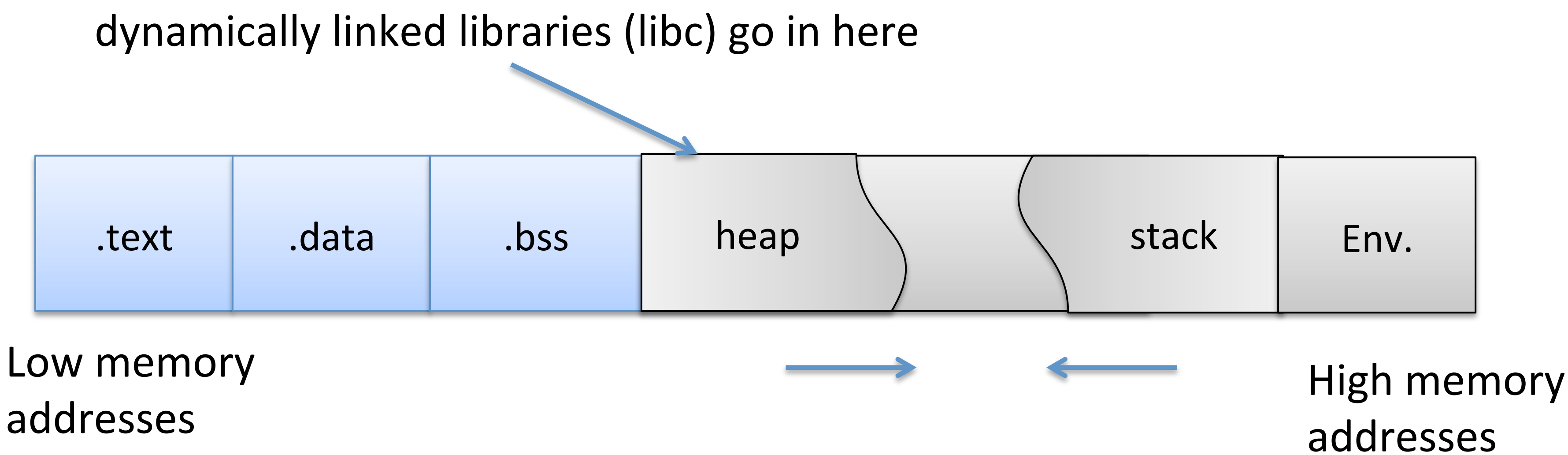
Buchanan et al., Blackhat 2008

W^X wrapup

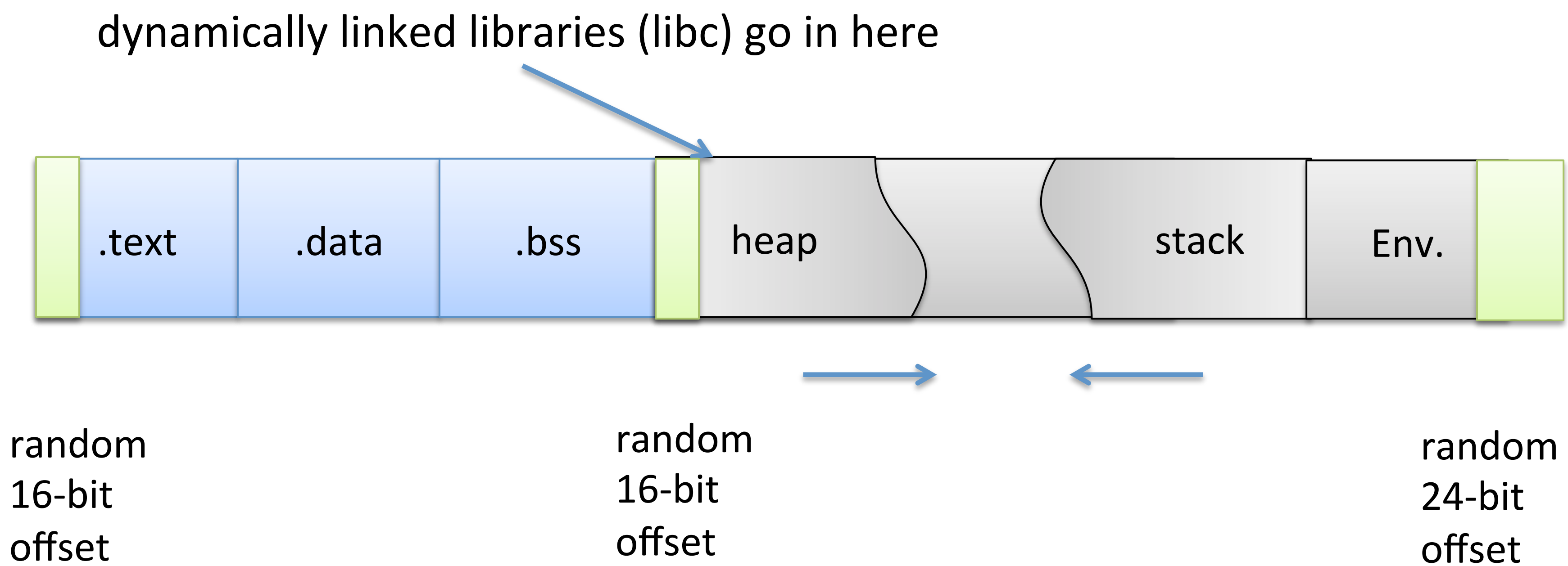
W^X does not prevent arbitrary code execution, but does make it harder!

What else can we do?

Address space layout randomization (ASLR)



Address space layout randomization (ASLR)



PaX implementation for example:

- Randomize offsets of three areas
- 16 bits, 16 bits, 24 bits of randomness
- Adds unpredictability… but how much?

Defeating ASLR

- W^X not on?

  Large nop sled with classic buffer overflow

- Use a vulnerability that can be used to leak address information (e.g., printf arbitrary read)

- Brute force the address

Defeating ASLR

Brute-forcing example from reading "On the effectiveness of Address Space Layout Randomization" by Shacham et al.

request

Apache forks off child process to handle request

response

Apache web server with Oracle 9 PL/SQL module

There is a buffer overflow in module that helps process request

Defeating ASLR

Brute-forcing example from reading “On the effectiveness of Address Space Layout Randomization” by Shacham et al.

Attacker makes a guess of where usleep() is located in memory

request

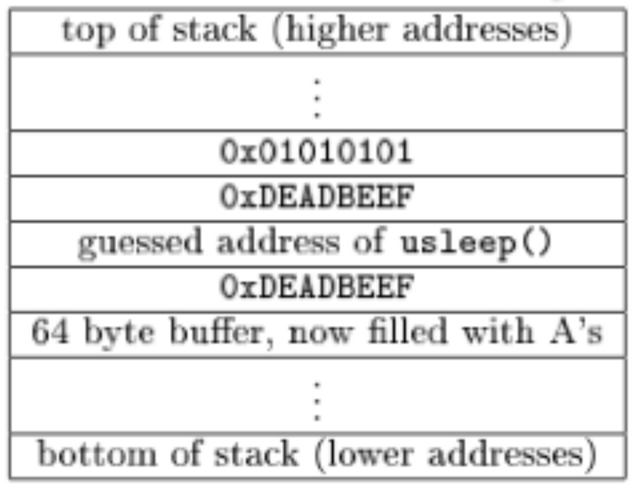


Figure 2: Stack after one probe

Apache web server with Oracle 9 PL/SQL module

Failure will crash the child process immediately and therefore kill connection

Success will crash the child process after sleeping for 0x01010101 microseconds and kill connection

Maximum time (s)	Average time (s)	minimum time (s)
810	216	29

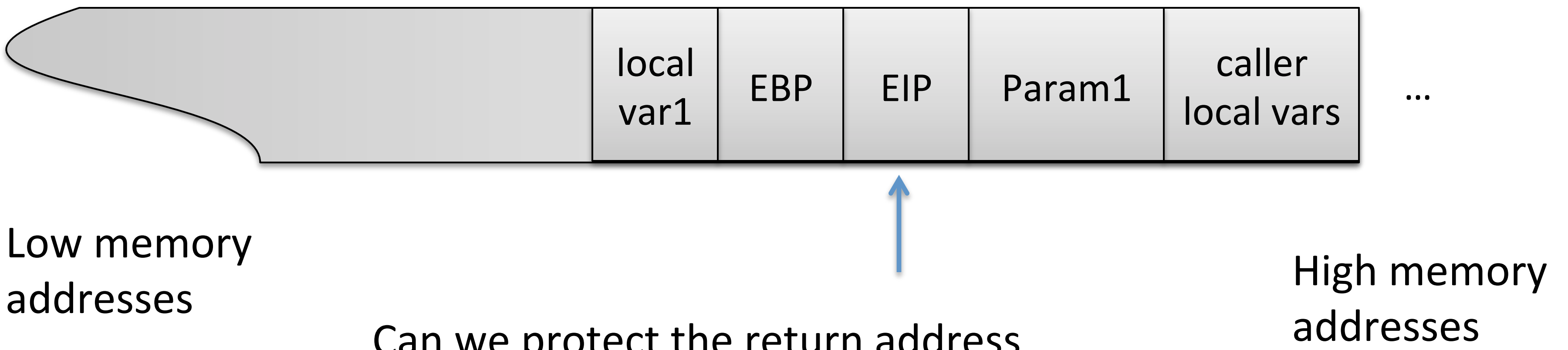
ASLR

If on 64-bit architecture, randomization significantly more effective

Can also randomize more stuff:

- Instruction set randomization
- per-memory-allocation randomization
- etc.

Protecting the stack

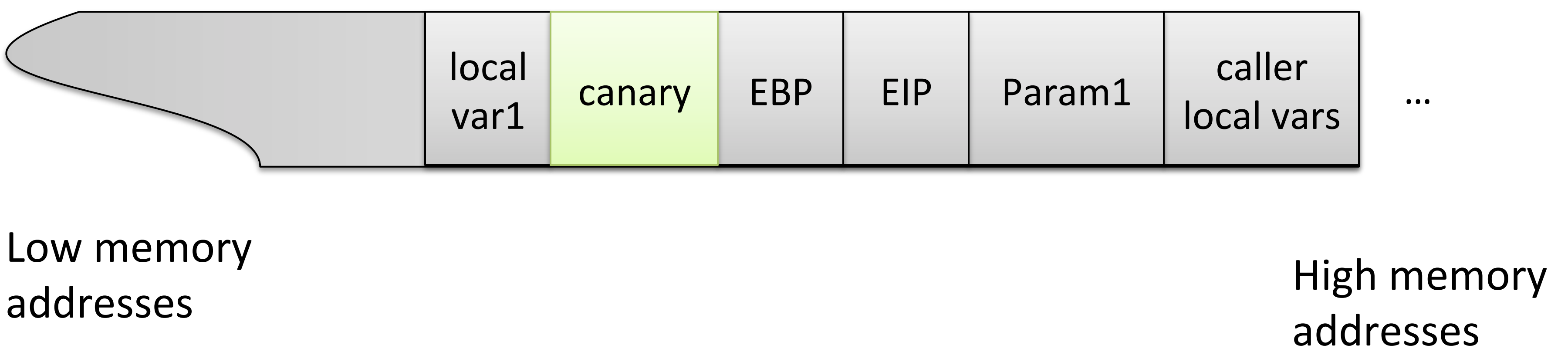


Can we protect the return address from being overwritten?

Two approaches:

- Detect manipulation (and then fail safe)
- Prevent it completely

Detection: stack canaries

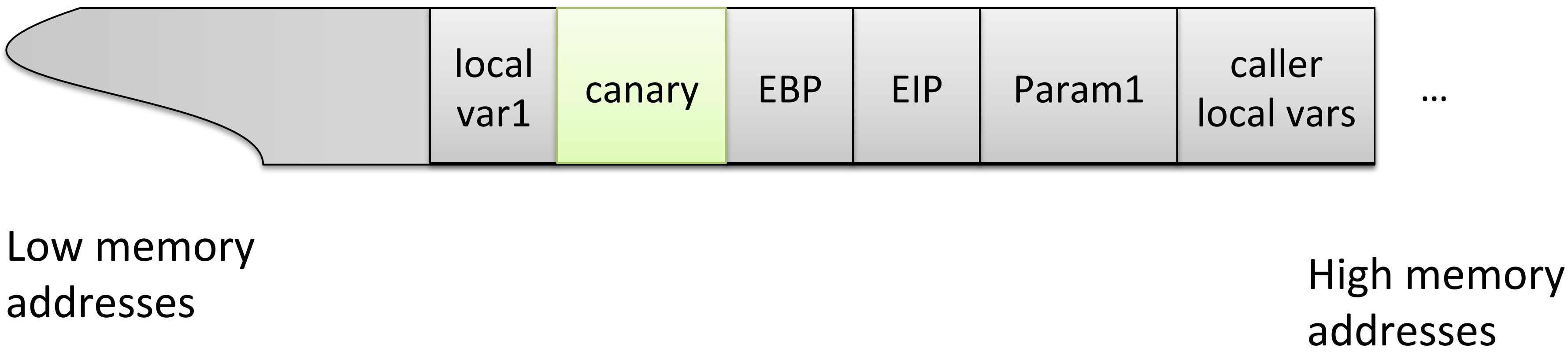


Canary value can be:

- Random value (choose once for whole process)
- NULL bytes / EOF / etc. (string functions won't copy past canary)

On end of function, check that canary is correct, if not fail safe

Detection: stack canaries



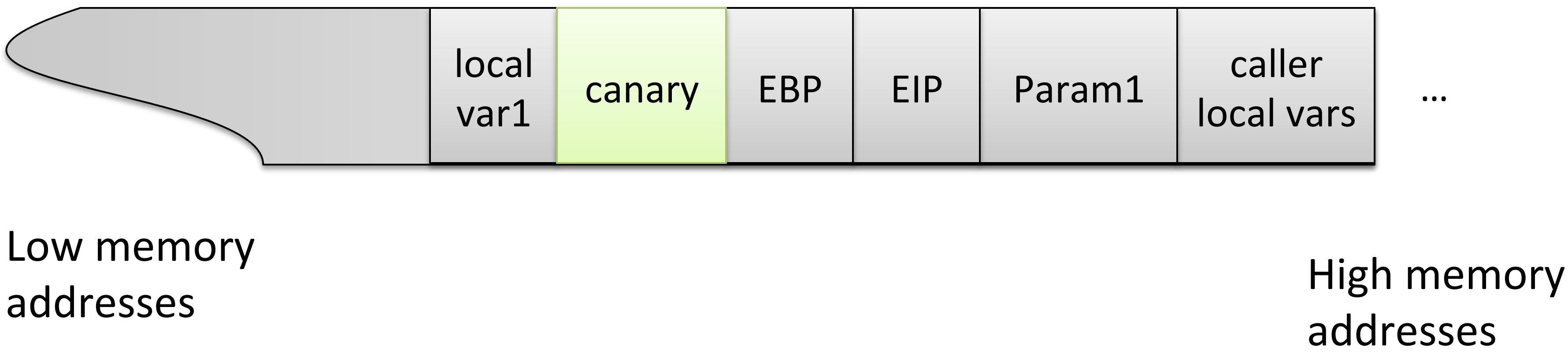
StackGuard:

- GCC extension that adds runtime canary checking
- 8% overhead on Apache

ProPolice:

- Modifies how canaries inserted
- Adds protection for registers
- Sorts variables so arrays are highest in stack

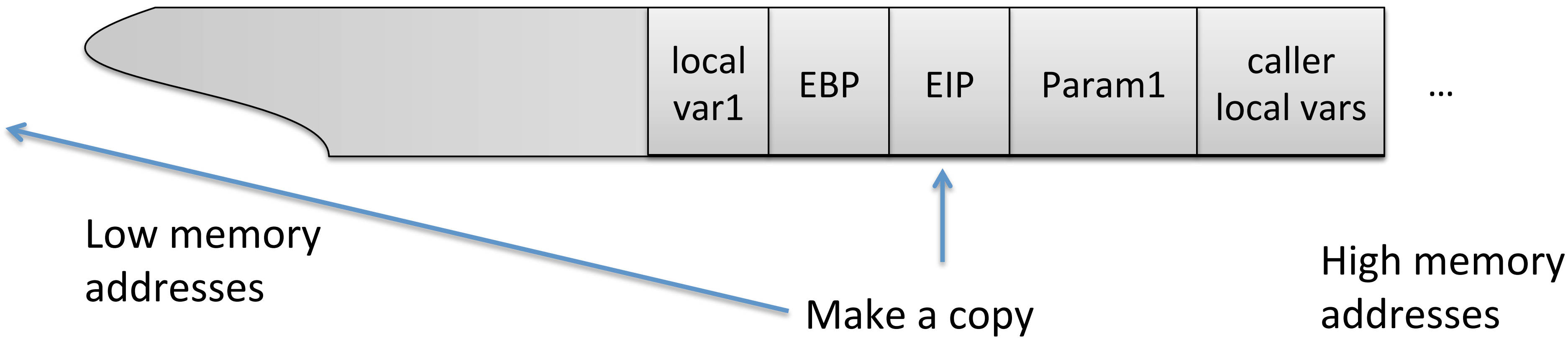
Detection: stack canaries



Discussion: How would you get around it?

http://www.phrack.org/issues.html?issue=56&id=5

Detection: copying values to safe location

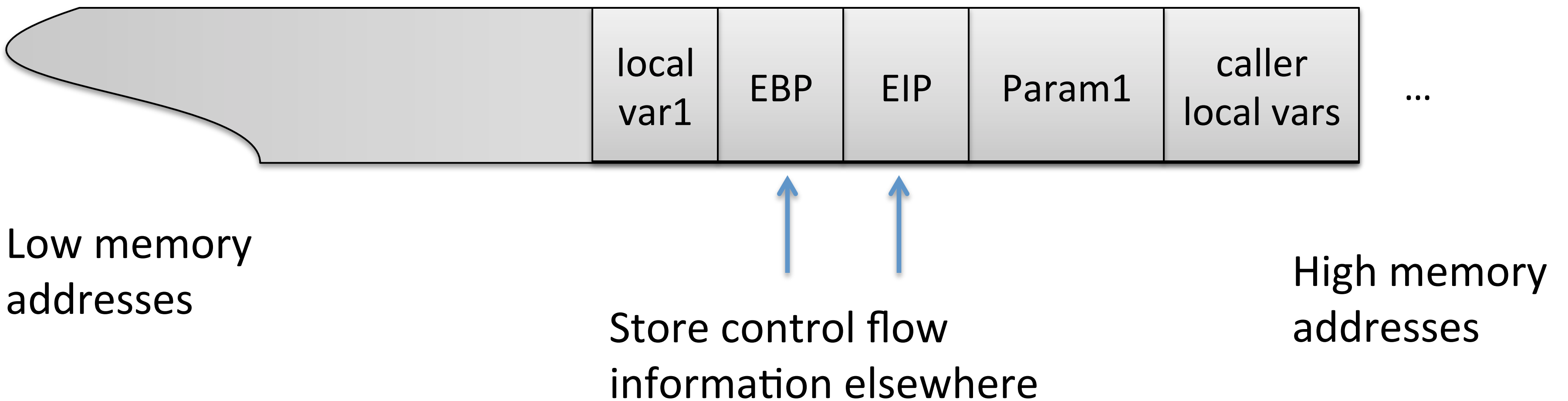


StackShield:

- Function call: copy return address to a safe location (beginning of .data)
- Check if stack value is different on function exit

Discussion: How would you get around this?

Prevention



StackGhost:

- Encrypting the return address
  - XOR with random value on function entrance
  - XOR with same value on function exit
- Per-kernel XOR vs. Per-process XOR
- Return address stack

Confinement (sand boxing)

- All the mechanisms thus far are circumventable
- Can we at least confine code that is potentially vulnerable so it doesn’t cause harm?

Simple example is chroot

chroot /tmp/guest
su guest

Now all file access are prepended with /tmp/guest

open(“/etc/passwd”, “r”)

Attempts to open
/tmp/guest/etc/passwd

Limitation is that all needed files must be inside chroot jail

jailkit

Limitation: network access not inhibited

Escaping jails

open("../../etc/passwd", "r")

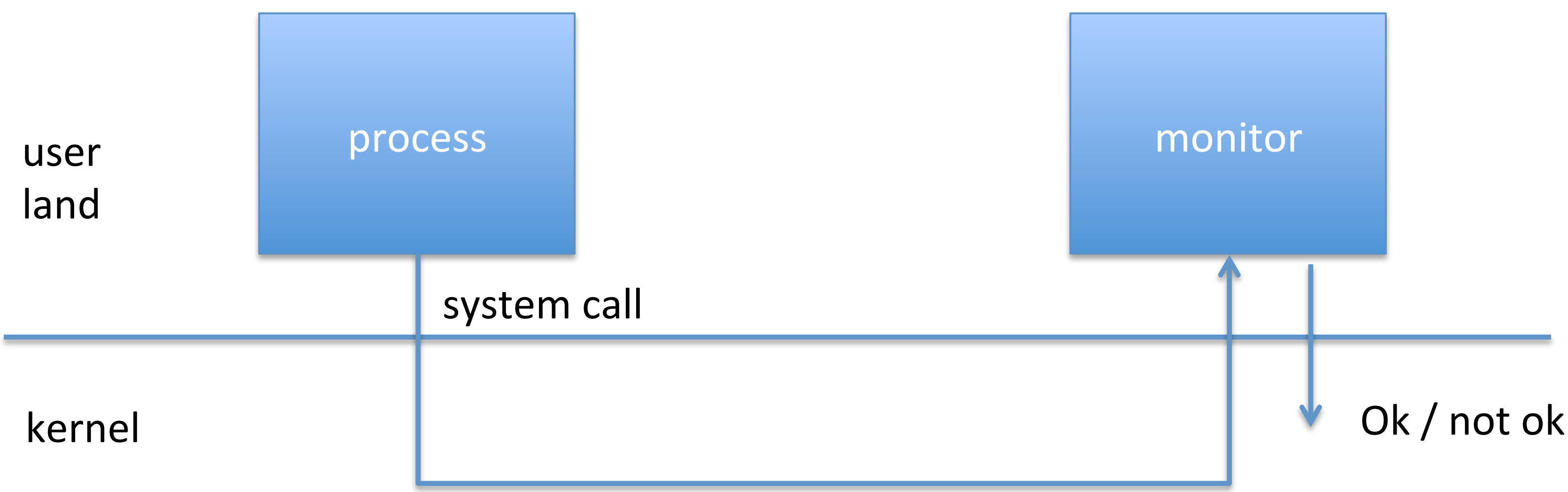
Attempts to open
/tmp/guest/../../etc/passwd

chroot should only be executable by root

create /aaa/etc/passwd
chroot /aaa
su root

System call interposition

- Malicious code must make system calls in order to do bad things
- So monitor system calls!



Janus

Wagner et al.

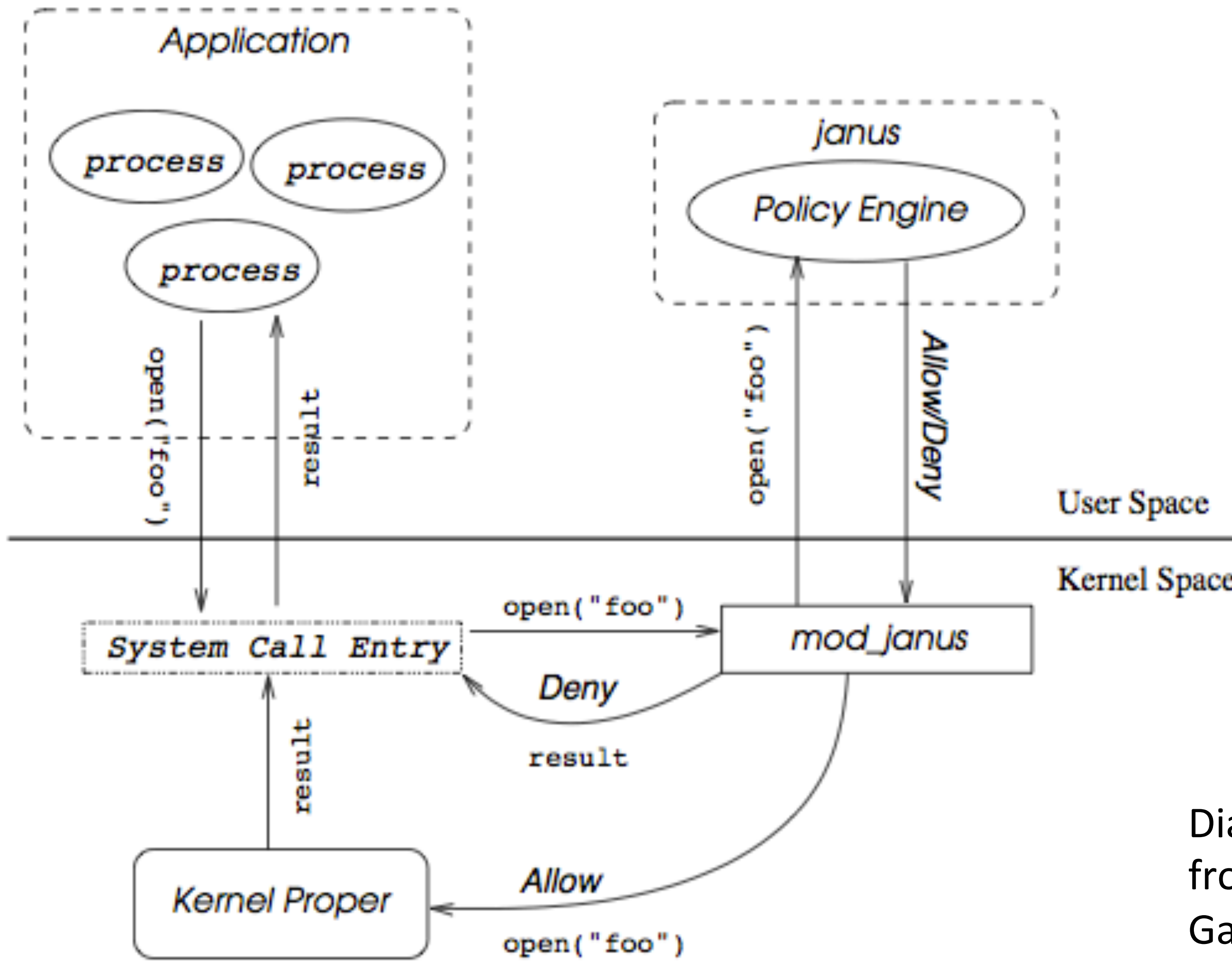


Figure 1. System Call Interposition in Janus

Diagram from Garfinkel 2003

Software-fault isolation example: Google Native Client

Goal: run native code from a web browser safely

Examples are Quake and XaoS ported over

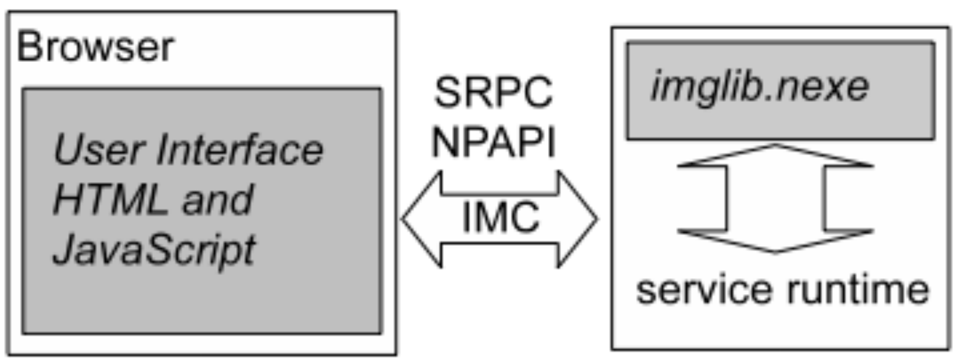


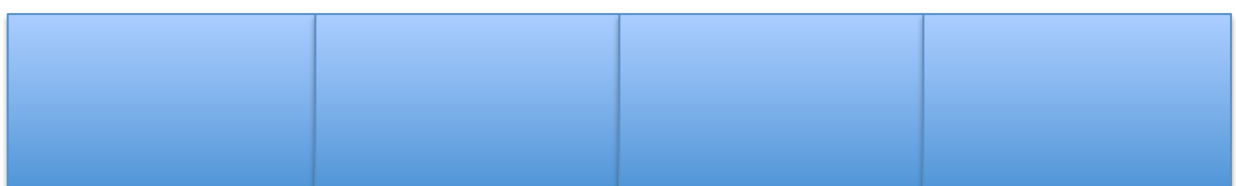
Figure 1: Hypothetical NaCl-based application for editing and sharing photos. Untrusted modules have a grey background.

From Yee et al. 2009

Software-fault isolation example: Google Native Client

Inner sandbox

- static analysis to detect flaws
- require code to abide by alignment and structure rules, allowing disassembly.
- Find any disallowed machine instructions
- x86 segmented memory to confine data and instruction references



Validator quickly checks that a binary abides by these rules

Software-fault isolation example: Google Native Client

Outer sandbox
- system call interposition to monitor
- similar to Janus / ptrace

Native client spec perf

	static	aligned	NaCl	increase
ammp	200	203	203	1.5%
art	46.3	48.7	47.2	1.9%
bzip2	103	104	104	1.9%
crafty	113	124	127	12%
eon	79.2	76.9	82.6	4.3%
equake	62.3	62.9	62.5	0.3%
gap	63.9	64.0	65.4	2.4%
gcc	52.3	54.7	57.0	9.0%
gzip	149	149	148	-0.7%
mcf	65.7	65.7	66.2	0.8%
mesa	87.4	89.8	92.5	5.8%
parser	126	128	128	1.6%
perlbmk	94.0	99.3	106	13%
twolf	154	163	165	7.1%
vortex	112	116	124	11%
vpr	90.7	88.4	89.6	-1.2%

Table 4: SPEC2000 performance. Execution time is in seconds. All binaries are statically linked.

Native client Quake perf

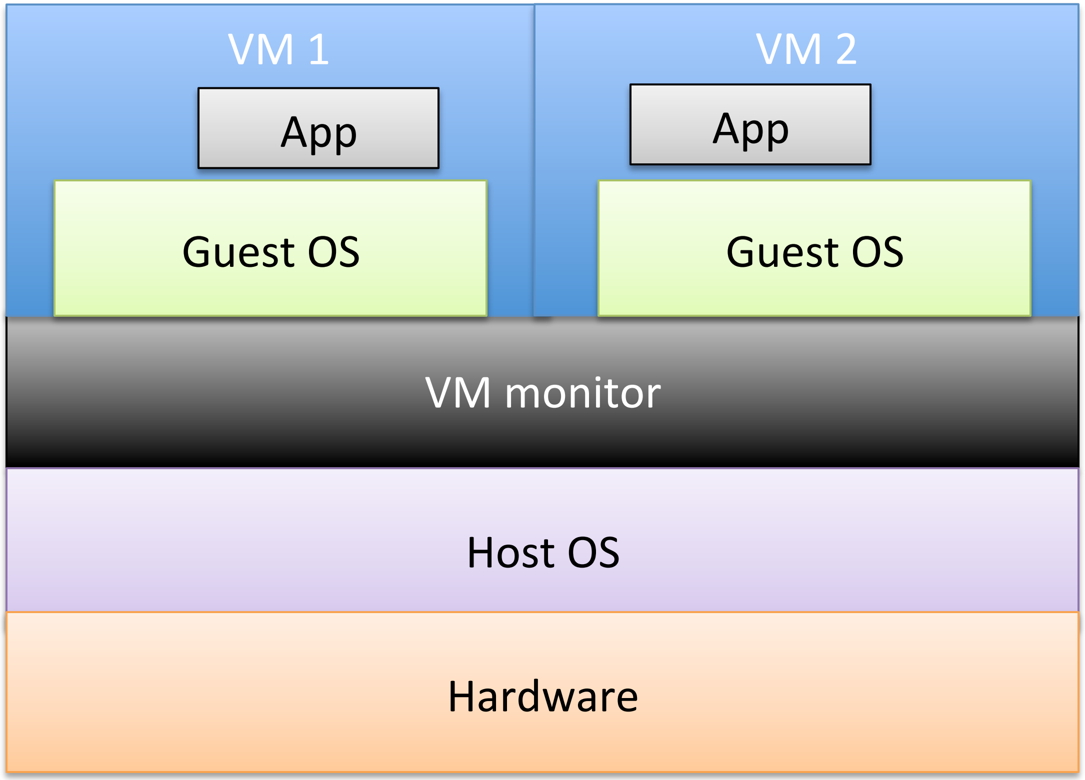
Run #	Native Client	Linux Executable
1	143.2	142.9
2	143.6	143.4
3	144.2	143.5
Average	143.7	143.3

Table 8: Quake performance comparison. Numbers are in frames per second.

More sandboxing: virtualization

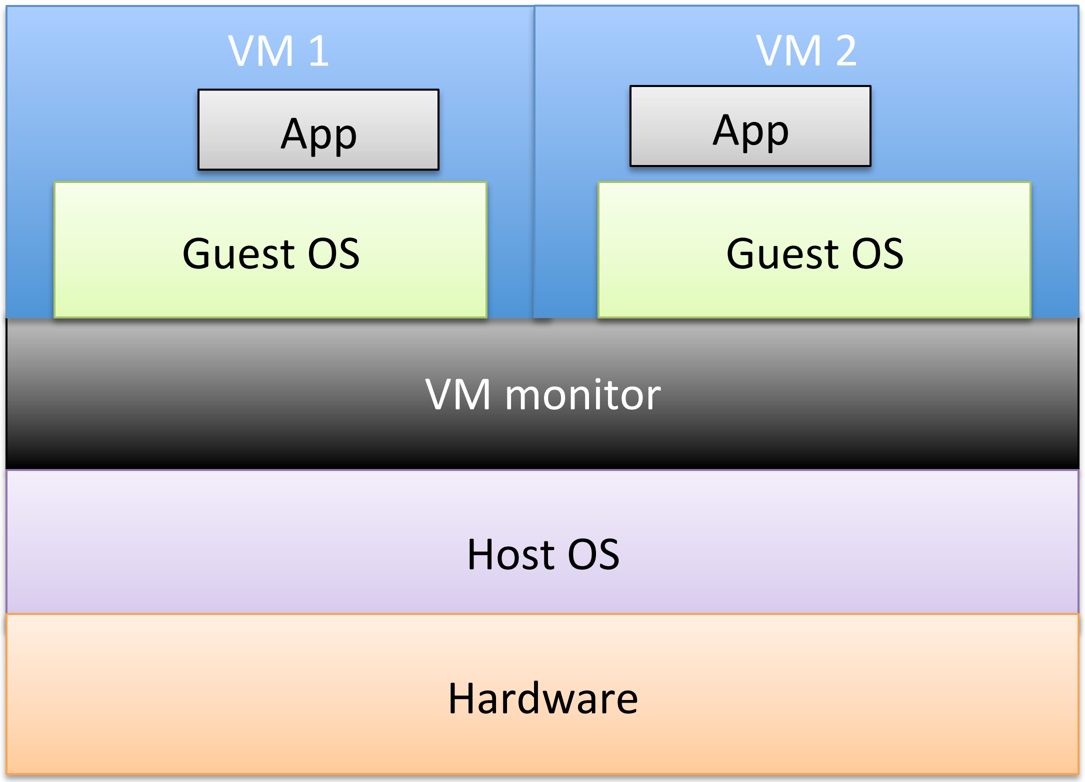
- Modern virtual machines (VMs) often used for sandboxing

NSA NetTop



More sandboxing: virtualization

- Malicious use of virtualization: blue pill virus



Discussion:
state of low level software security

- Do you think Native Client is fool proof?
- What about VM-based sandboxing?

- How does all this make you feel?