Exploit Defenses:
ASLR, W⊕X, TaintCheck

Brad Karp

UCL Computer Science

CS 0133

4th December 2019

Host-Based Exploit Defenses

- Firewalls: defenses against exploits in-network
  - Can see lots of traffic at one monitoring point
  - Can filter traffic for many vulnerable hosts
  - Limited information available: only packet fields, payload contents
- Today: identifying and defending against exploits on hosts
  - Much more information: see effect of network request on running process's execution!
  - Potentially more accurate
  - Requires changes to host software
  - Performance concern; don't want to slow busy server

Outline

- W$\oplus$X page protections
  - and limitations
- Address Space Layout Randomization
  - and limitations
- TaintCheck
  - and limitations

Goals for Host-Based Exploit Defenses

- Works on executables

  - ...and so for legacy code
  - Source code often not available

- Prevents broadest possible range of exploits
- Low/no false positives, false negatives
- Minimal performance reduction

  - Server operator won't want to sacrifice performance
  - Attacker may recognize server protected if performance slows—and not send malicious request!
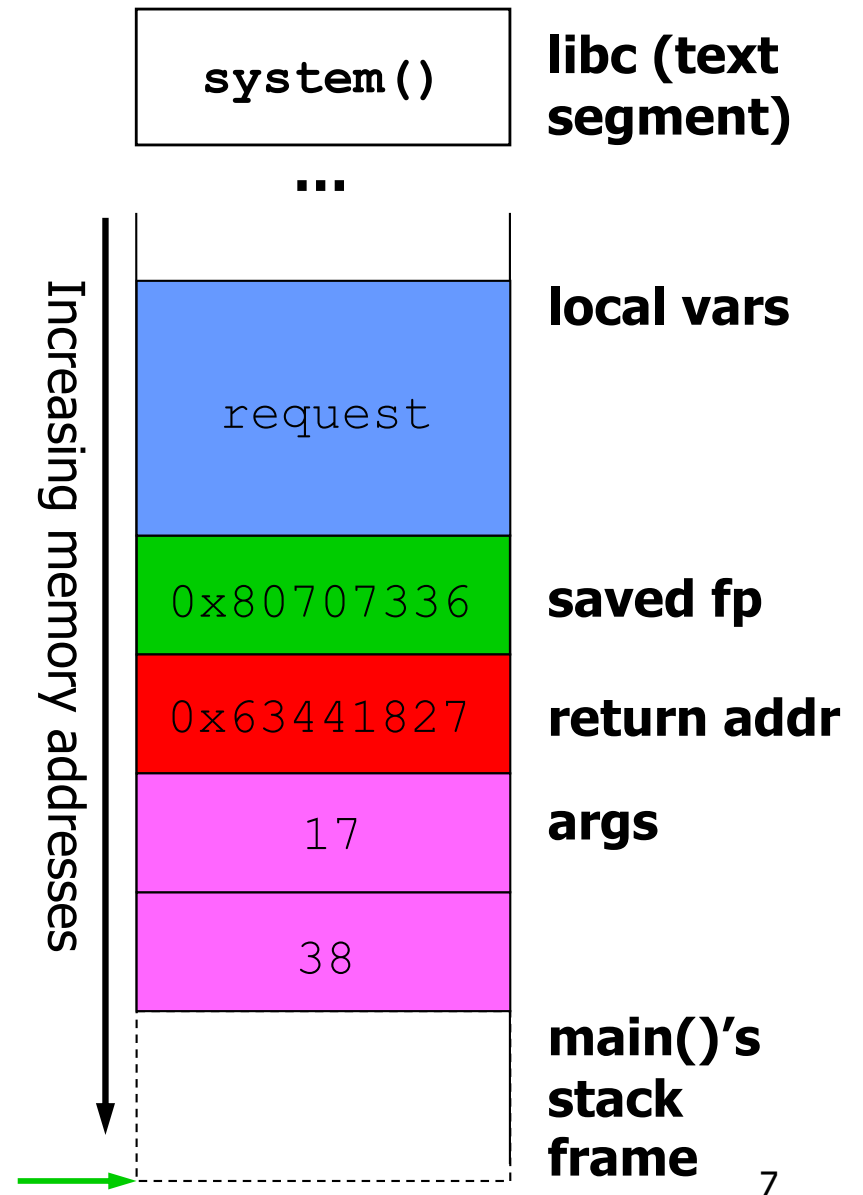
W⊕X Page Protections

- Recall from OS: CPU implements page protection in hardware
  - For each 4K memory page, permission bits specified in page table entry in kernel: read, write
- Central problem in many exploits:
  - Code supplied by user in input data
  - Execution transferred to user's input data
- **Idea: don't let CPU execute instructions stored in data pages**
  - i.e., each page should either be writable or executable, but not both: W⊕X
  - Text pages: X, not W
  - Data (stack, heap) pages: W, not X

W⊕X Details

- Originally no X bit in Intel CPUs; just R and W, all R pages implicitly X
- AMD and Intel introduced "NX" bit (no execute); available on today's processors (in PAE mode)
  - Not a new idea; present in, e.g., DEC Alpha
  - Used by Linux PaX and Windows XP SP2
- Linux PaX implements W⊕X for x86 processors without NX bit hardware
  - Based on segment limit registers
  - Halves address space available to each process
  - Minor performance reduction
- W⊕X breaks just-in-time (JIT) code generation in legacy applications!

W⊕X Hole: Return-to-libc Attacks

- Instead of putting shellcode on stack, can put args there, overwrite return address with pointer to well known library function

  - e.g., `system("/bin/sh");`

- Return-to-libc attack

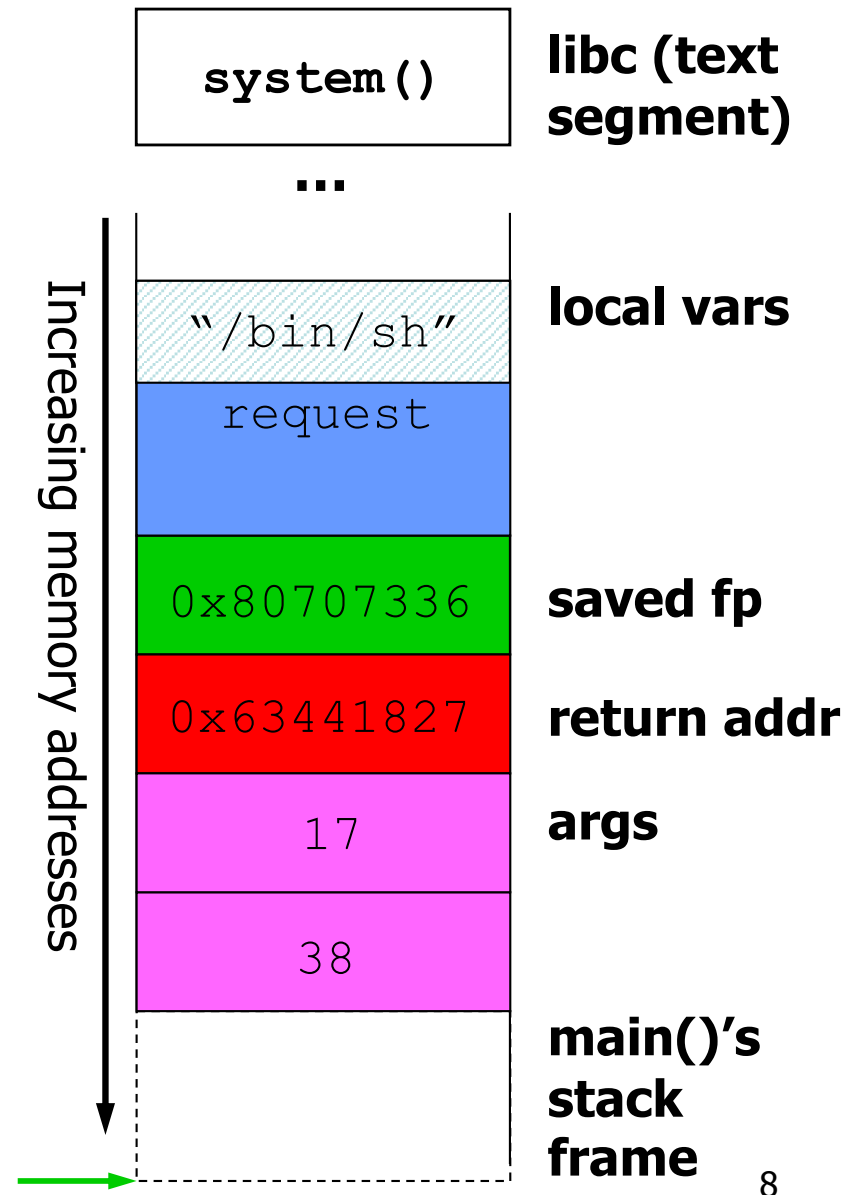|	
`system()`	libc (text segment)
...	
request	local vars
0x80707336	saved fp
0x63441827	return addr
17	args
38	
	main()'s stack frame

Increasing memory addresses

W⊕X Hole: Return-to-libc Attacks

- Instead of putting shellcode on stack, can put args there, overwrite return address with pointer to well known library function
  - e.g., `system("/bin/sh");`

- Return-to-libc attack

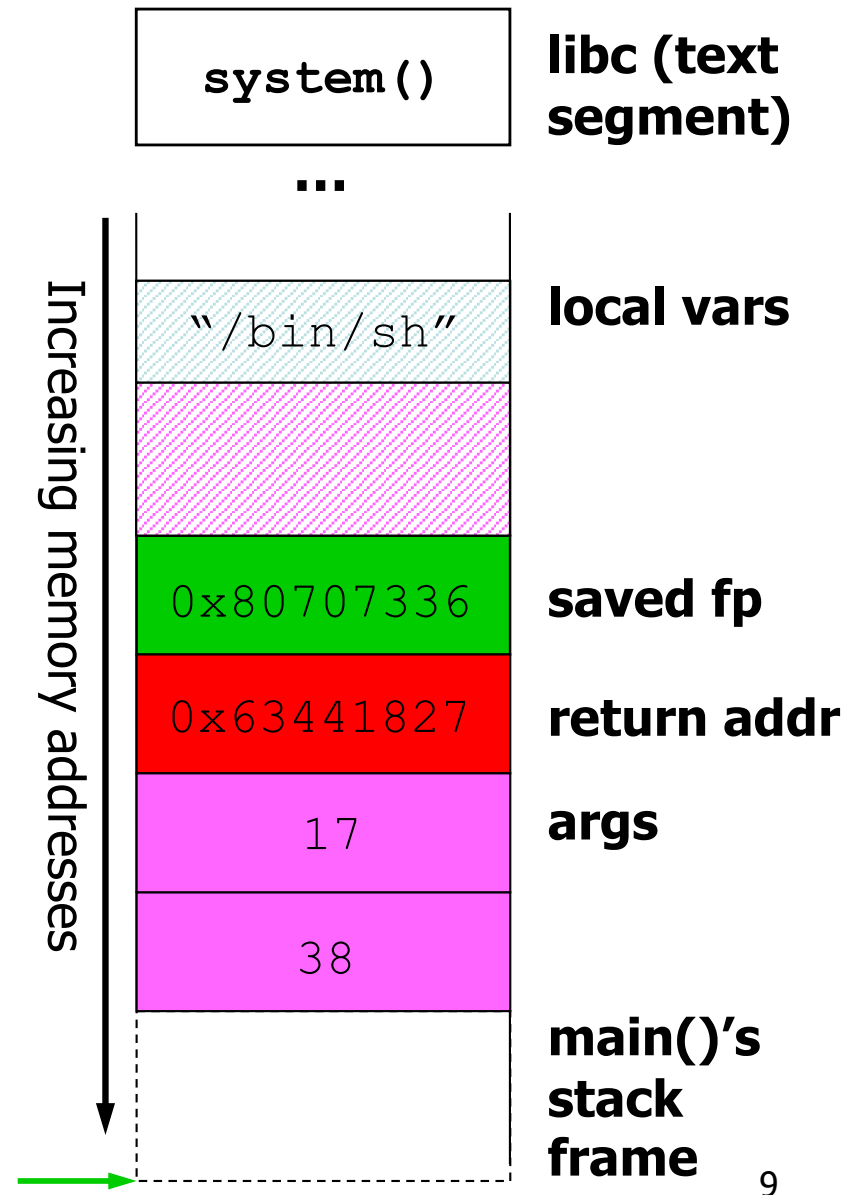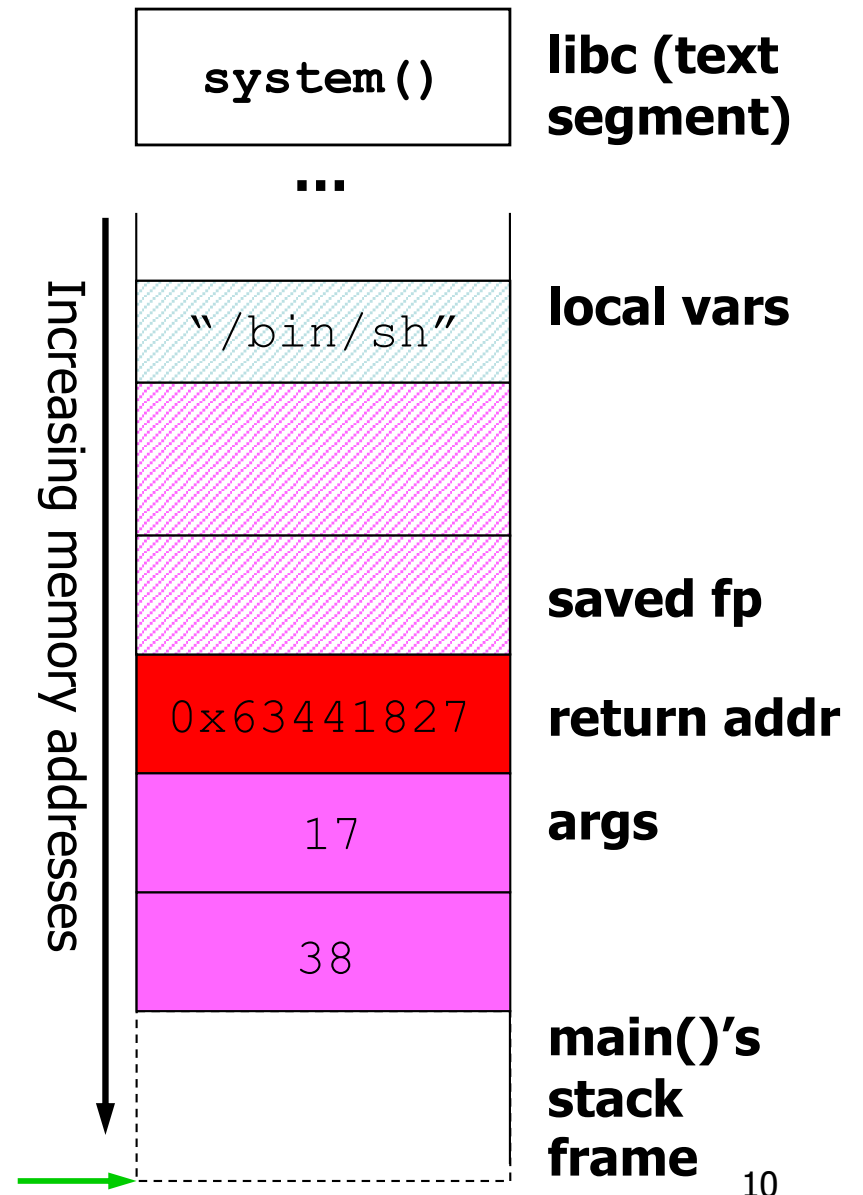|	
`system()`	libc (text segment)
...	
`"/bin/sh"`	local vars
`request`	
`0x80707336`	saved fp
`0x63441827`	return addr
`17`	args
`38`	
	main()'s stack frame

Increasing memory addresses

8

W⊕X Hole: Return-to-libc Attacks

- Instead of putting shellcode on stack, can put args there, overwrite return address with pointer to well known library function
  - e.g., `system("/bin/sh");`

- Return-to-libc attack

`system()`	libc (text segment)

...

Increasing memory addresses →

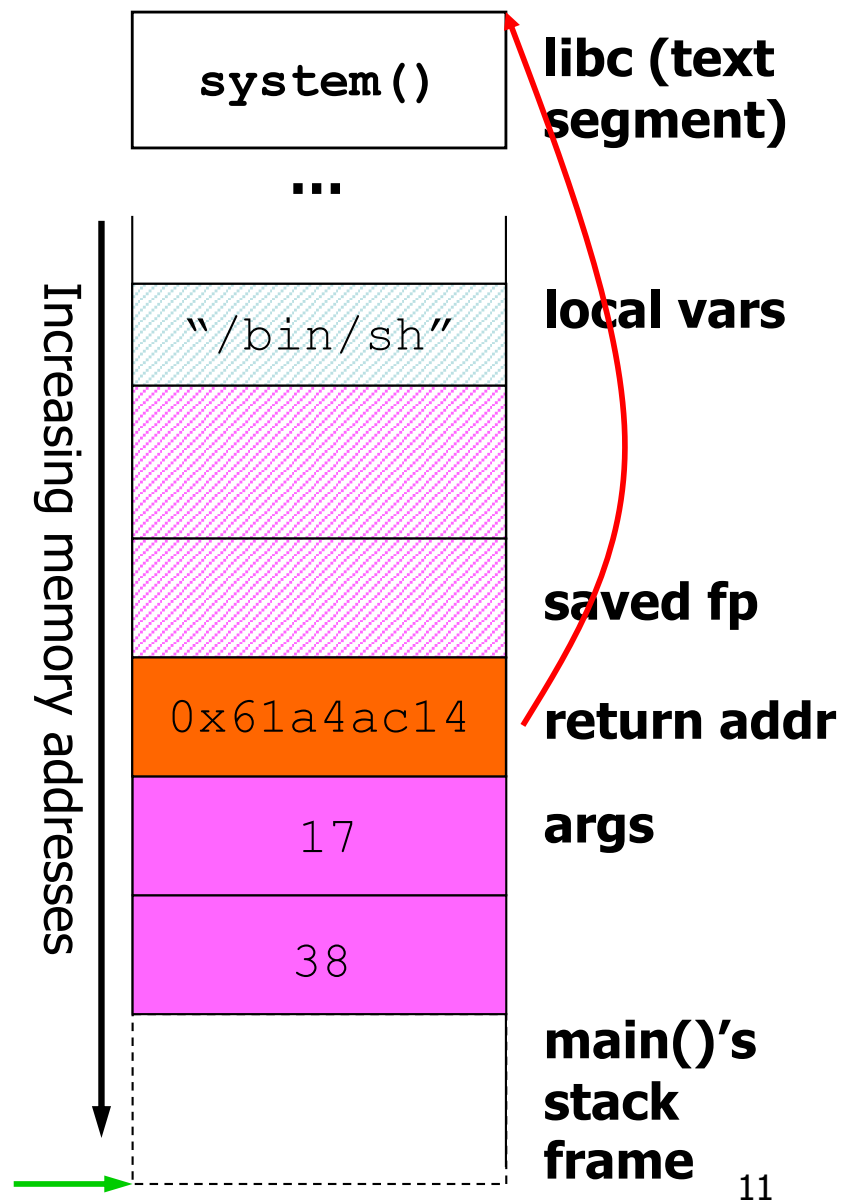|	
`"/bin/sh"`	local vars
`0x80707336`	saved fp
`0x63441827`	return addr
17	args
38	
	main()'s stack frame

W⊕X Hole: Return-to-libc Attacks

- Instead of putting shellcode on stack, can put args there, overwrite return address with pointer to well known library function
  - e.g., `system("/bin/sh");`

- Return-to-libc attack

```
system()
```
libc (text segment)

...

Increasing memory addresses

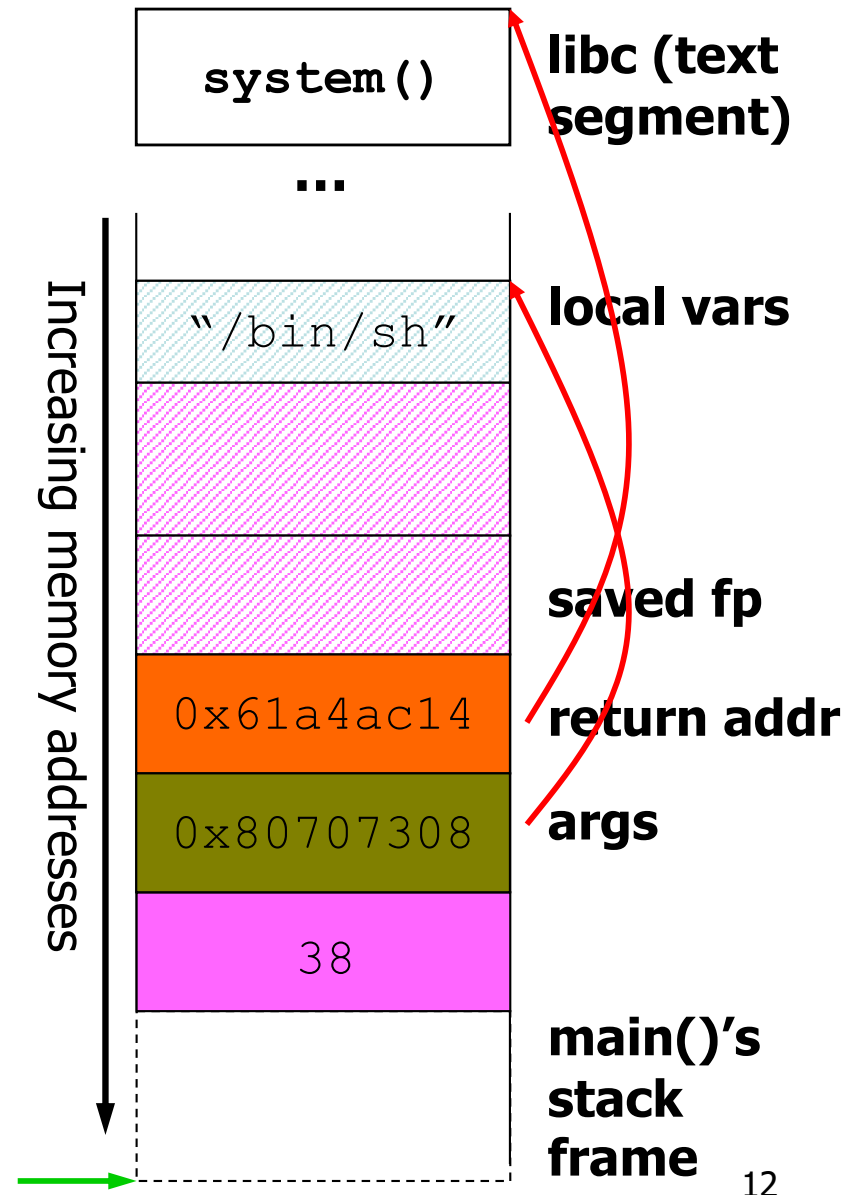|	
`"/bin/sh"`	**local vars**
	saved fp
`0x63441827`	**return addr**
`17`	**args**
`38`	
	main()'s stack frame

W⊕X Hole: Return-to-libc Attacks

- Instead of putting shellcode on stack, can put args there, overwrite return address with pointer to well known library function
  - e.g., `system("/bin/sh");`

- Return-to-libc attack

`system()`	**libc (text segment)**
...	
`"/bin/sh"`	**local vars**
	saved fp
`0x61a4ac14`	**return addr**
17	**args**
38	
	main()'s stack frame

Increasing memory addresses

W⊕X Hole: Return-to-libc Attacks

- Instead of putting shellcode on stack, can put args there, overwrite return address with pointer to well known library function

  – e.g., `system("/bin/sh");`

- Return-to-libc attack

```
system()
```
...

libc (text segment)

"/bin/sh"

local vars

saved fp

0x61a4ac14

return addr

0x80707308

args

38

main()'s stack frame

Increasing memory addresses

Address Space Layout Randomization (ASLR)

- Central observation: attacker must predict addresses
  - e.g., shellcode buffer address, libc function address, string argument address
- Idea: randomize addresses in process
  - With high probability, attacker will guess wrong
  - Jump to unmapped memory: crash
  - Jump to invalid instruction stream: crash
- Useful as efficient exploit detector
  - Memory faults or illegal instructions suggest exploit

ASLR Implementation:
x86-32 PaX for Linux

- Linux process contains three memory regions:
  - Executable: text, init data, uninit data
  - Mapped: heap, dynamic (shared) libraries, thread stacks, shared memory
  - Stack: user stack
- ASLR adds random offset to each area when process created
  - Efficient; easily supported by virtual memory hardware
  - 16, 16, 24 bits randomness, respectively
- Mapped offset limited to 16 bits
  - bits 28-31 cannot be changed; would interfere with big mmap()s
  - bits 0-11 cannot be randomized; would make mmap()ed pages not be page-aligned

Derandomization Attack on ASLR [Shacham, Boneh et al.]

- 16 bits not that big; try to guess random offset added to mapped area

- Once know random offset, can predict addresses of shared libraries
  - thus libc function addresses
  - …so can mount return-to-libc attack

- Two phases:
  - brute-force random offset to mapped area
  - compute "derandomized" address of syscall(), use in return-to-libc attack
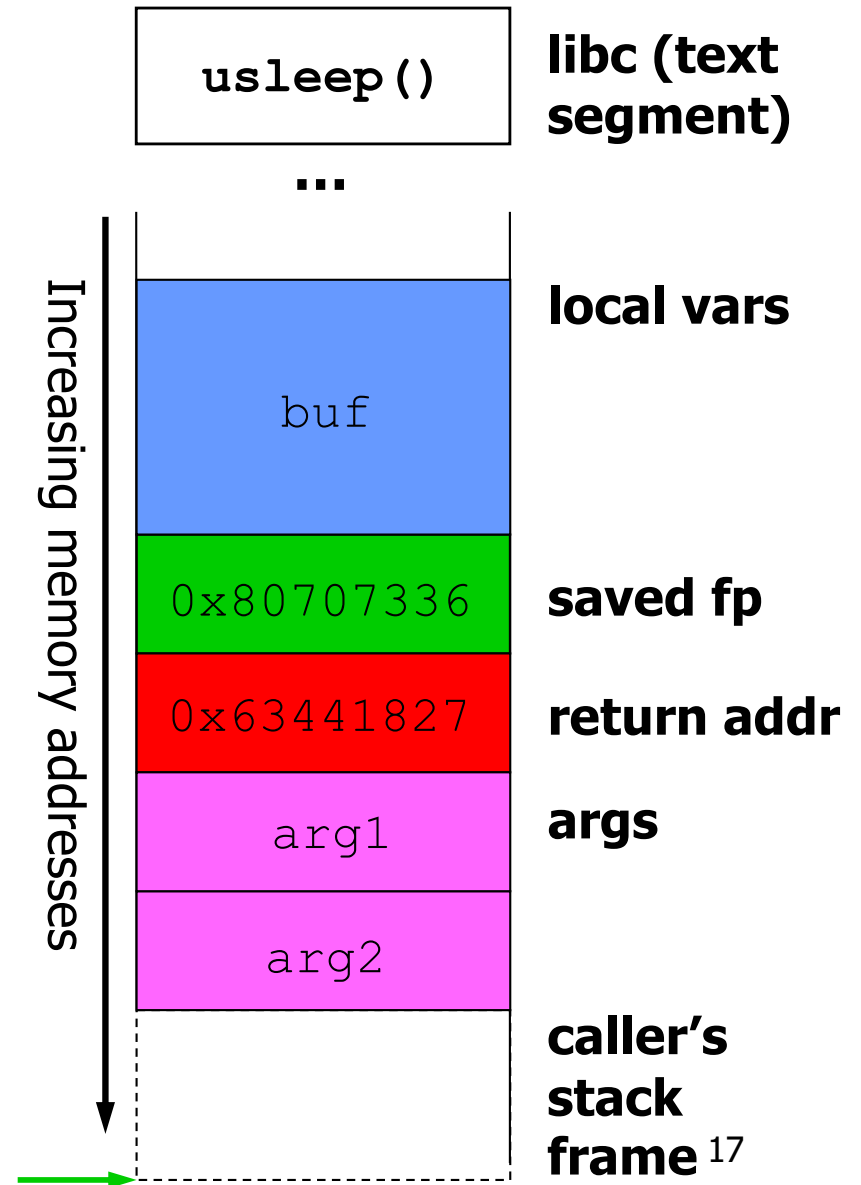
Derandomization Attack Details

- Target: "classic" stack buffer overflow placed in Apache web server

  ```
  char buf[64];
  …
  strcpy(buf, input);
  ```

- Plan:
  - Try to return to usleep(), guessing random offset for mapped area each time
  - If guess wrong, target process crashes, closes connection immediately; parent forks new child (with same random offset)
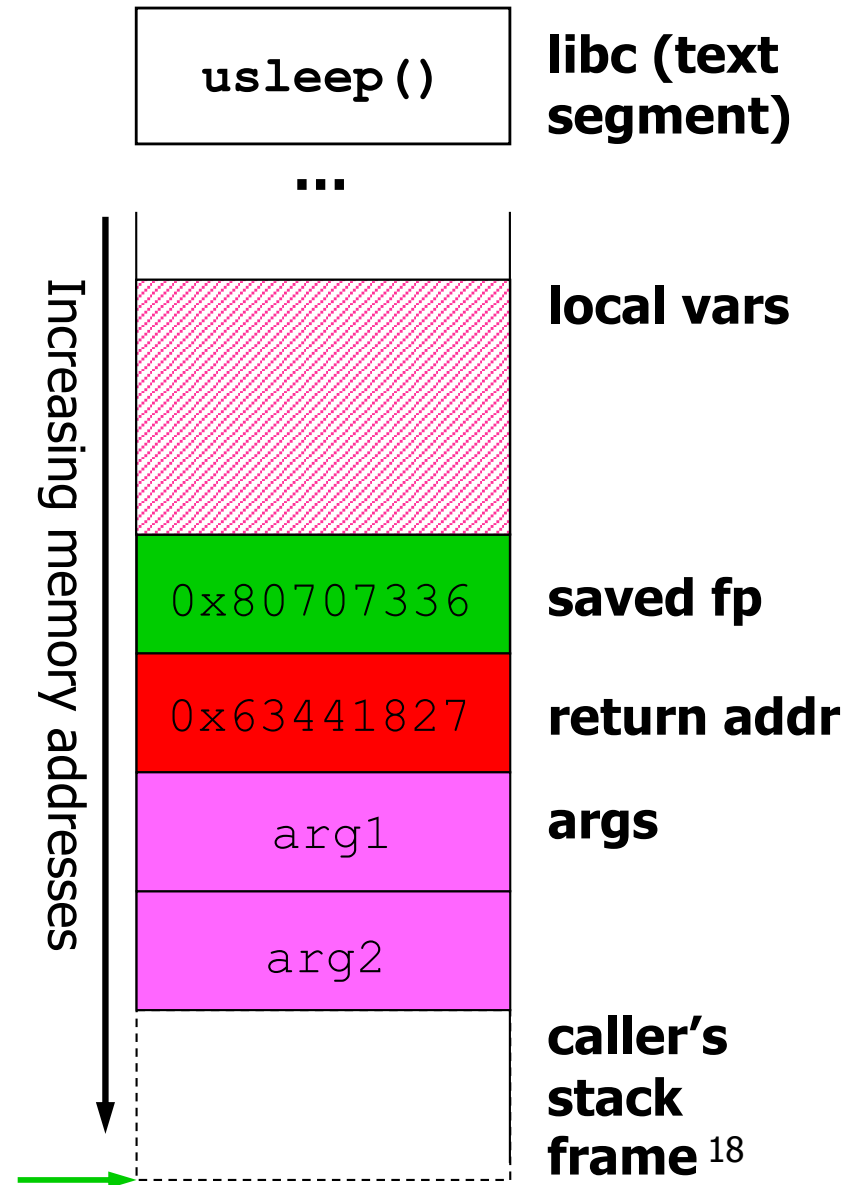 - If guess right, target process delays in usleep(), then crashes and closes connection immediately

Derandomization Attack: Phase 1

- Know offset of `usleep()` within libc, know base of mapped area (w/o randomization)
- Each return address guess:
  base + `usleep()` offset + guess in [0, 64K]
- If guess wrong, crash
- If guess right, `usleep()` sees return address `0xdeadbeef`, arg 16,843,009 usec (16 sec); sleep, crash

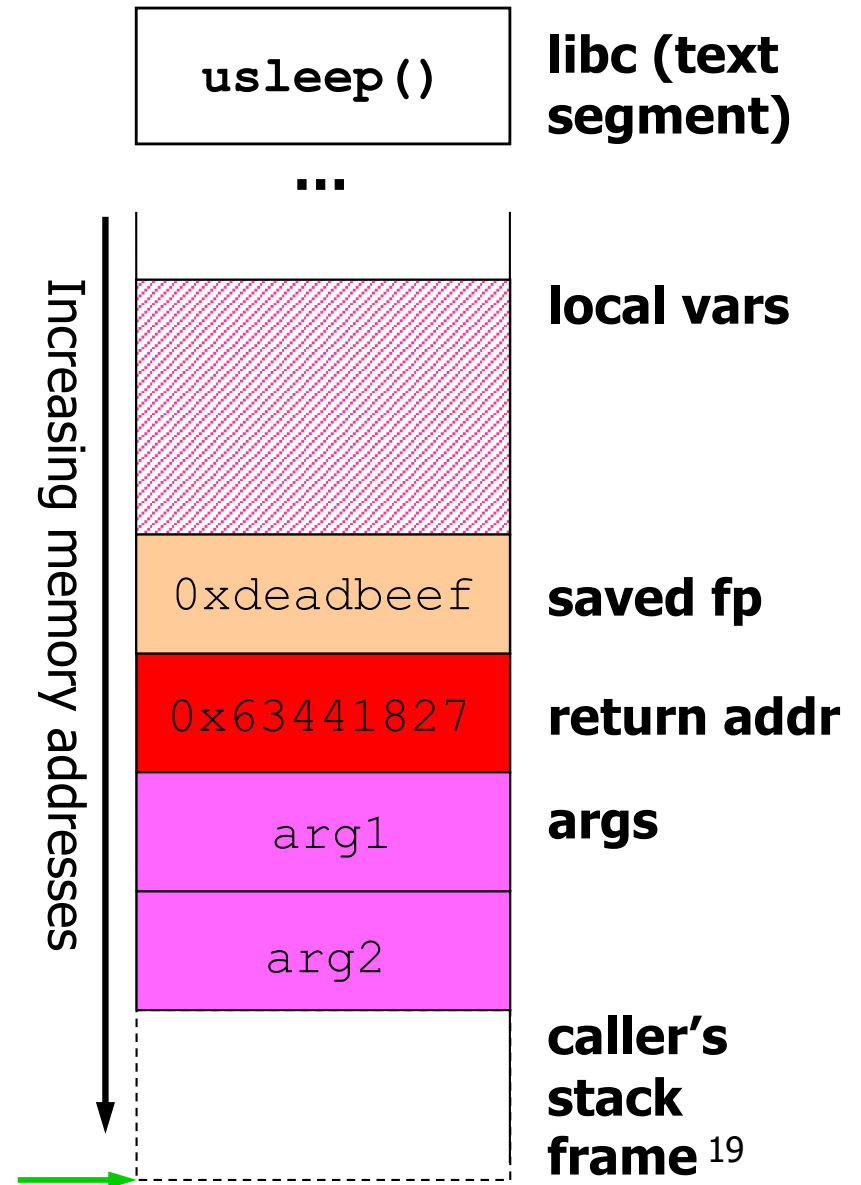|	
`usleep()`	libc (text segment)
...	
buf	local vars
0x80707336	saved fp
0x63441827	return addr
arg1	args
arg2	
	caller's stack frame [17]

Increasing memory addresses

Derandomization Attack: Phase 1

- Know offset of `usleep()` within libc, know base of mapped area (w/o randomization)
- Each return address guess:
  base + `usleep()` offset + guess in [0, 64K]
- If guess wrong, crash
- If guess right, `usleep()` sees return address `0xdeadbeef`, arg 16,843,009 usec (16 sec); sleep, crash

`usleep()`	**libc (text segment)**
...	

Increasing memory addresses →

local vars	
`0x80707336`	**saved fp**
`0x63441827`	**return addr**
`arg1`	**args**
`arg2`	
	caller's stack frame 18

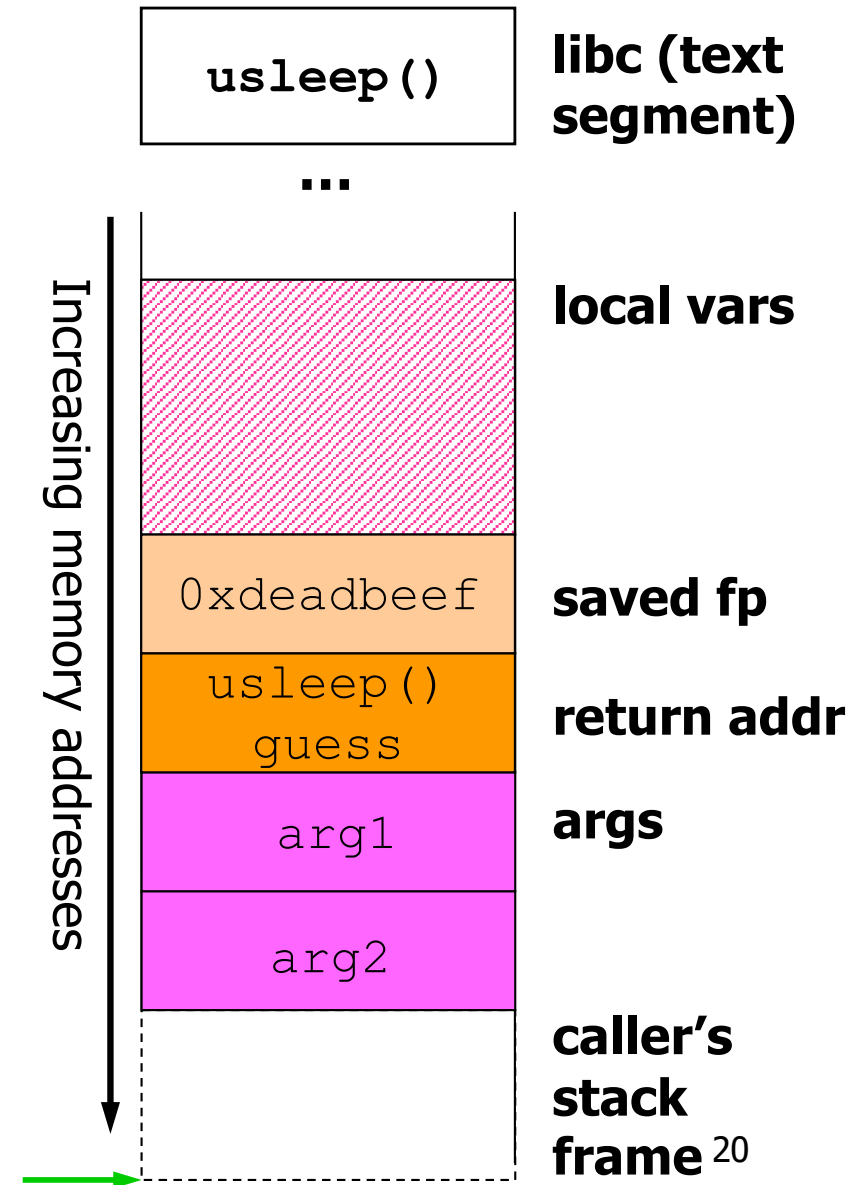# Derandomization Attack: Phase 1

- Know offset of `usleep()` within libc, know base of mapped area (w/o randomization)
- Each return address guess:
  base + `usleep()` offset + guess in [0, 64K]
- If guess wrong, crash
- If guess right, `usleep()` sees return address `0xdeadbeef`, arg 16,843,009 usec (16 sec); sleep, crash

`usleep()`	**libc (text segment)**
...	
(local vars)	**local vars**
`0xdeadbeef`	**saved fp**
`0x63441827`	**return addr**
`arg1`	**args**
`arg2`	
	caller's stack frame

Increasing memory addresses

19

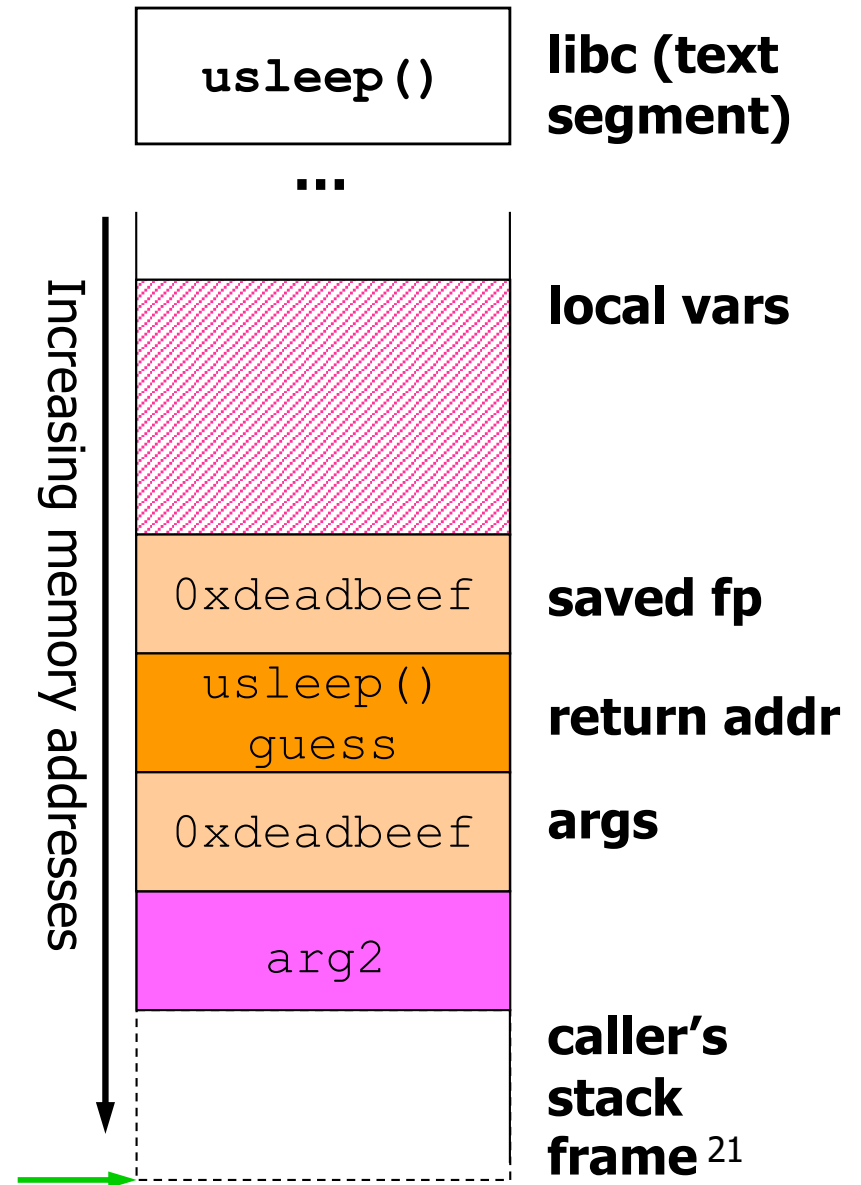# Derandomization Attack: Phase 1

- Know offset of `usleep()` within libc, know base of mapped area (w/o randomization)
- Each return address guess:
    base + `usleep()` offset + guess in [0, 64K]
- If guess wrong, crash
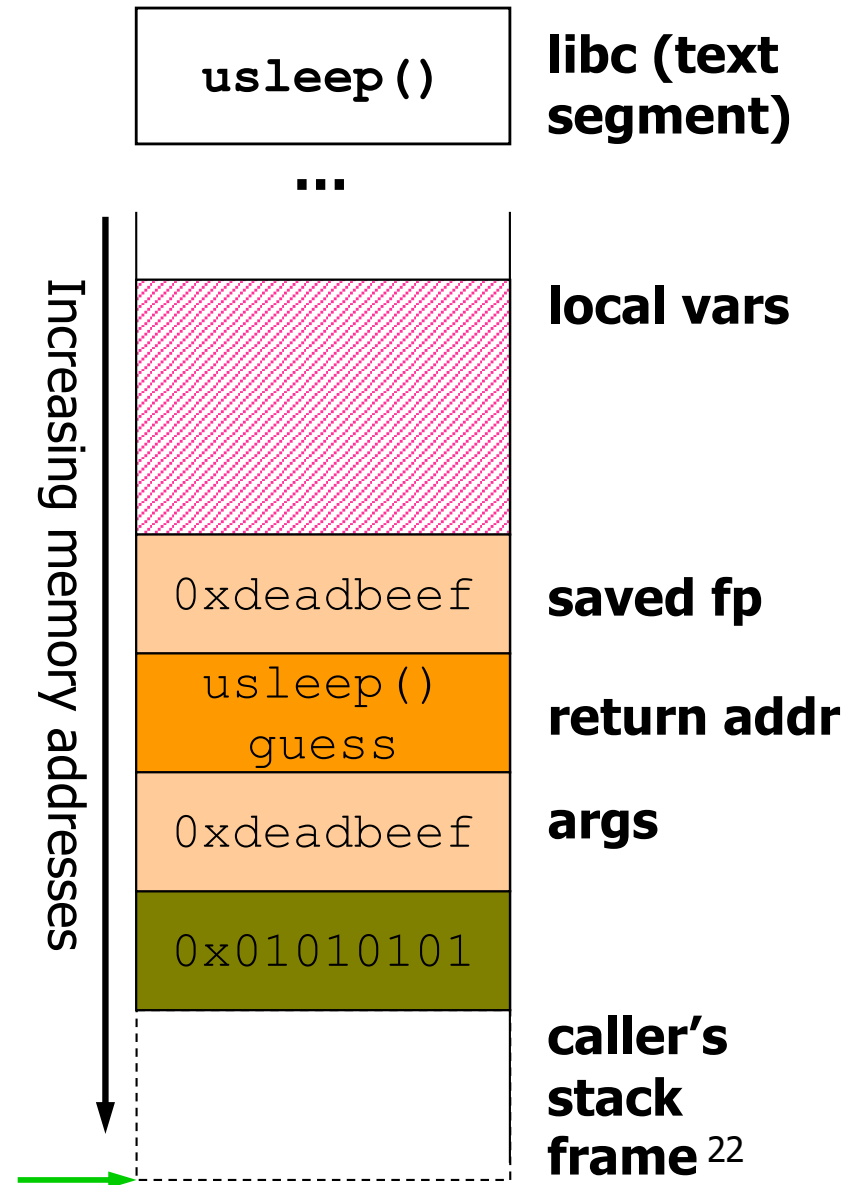- If guess right, `usleep()` sees return address `0xdeadbeef`, arg 16,843,009 usec (16 sec); sleep, crash

`usleep()`	libc (text segment)
...	
	local vars
`0xdeadbeef`	saved fp
`usleep()` guess	return addr
arg1	args
arg2	
	caller's stack frame

Increasing memory addresses

20

Derandomization Attack: Phase 1

- Know offset of `usleep()` within libc, know base of mapped area (w/o randomization)
- Each return address guess:
  base + `usleep()` offset + guess in [0, 64K]
- If guess wrong, crash
- If guess right, `usleep()` sees return address `0xdeadbeef`, arg 16,843,009 usec (16 sec); sleep, crash

`usleep()`	**libc (text segment)**
...	
(local vars)	**local vars**
`0xdeadbeef`	**saved fp**
`usleep()` guess	**return addr**
`0xdeadbeef`	**args**
`arg2`	
(caller's stack frame)	**caller's stack frame**

Increasing memory addresses

Derandomization Attack: Phase 1

- Know offset of `usleep()` within libc, know base of mapped area (w/o randomization)
- Each return address guess:

    base + `usleep()` offset + guess in [0, 64K]

- If guess wrong, crash
- If guess right, `usleep()` sees return address `0xdeadbeef`, arg 16,843,009 usec (16 sec); sleep, crash

`usleep()`	libc (text segment)
...	
(local vars)	local vars
`0xdeadbeef`	saved fp
`usleep() guess`	return addr
`0xdeadbeef`	args
`0x01010101`	
	caller's stack frame

Increasing memory addresses

Derandomization Attack: Phase 2

- Now know random offset of mapped area
- Compute exact address of system() libc function:
  address = base + system() offset in libc + guessed random offset
- Perform return-to-libc attack using system(), as in earlier example; "/bin/sh" in buf[] on stack
- Turns out caller's frame contains pointer to buf[]!
- So overwrite stack past buf[] with several copies of address of any ret instruction found in libc, followed by address of system()
  – Repeatedly pops stack until returns to system(), with pointer to buf[] on top of stack (argument position)
  – Details in paper, top of p. 8

Derandomization Attack: Performance

- Many trials of phase 1 necessary to learn random offset of mapped area on server

- For 1.8 GHz AMD Athlon server, attacked by 2.4 GHz Pentium 4 client:
  - 216 seconds on average to complete both phases
  - 200 bytes of traffic per probe; 12.8 MB data from client worst-case, 6.4 MB data in expectation

Can ASLR Be Made More Robust?

- 64-bit CPU architectures
  - Probably 40 bits of random offset; much harder to brute-force without attracting attention; so some help with more recent hardware
- Re-randomize address space after every crash (probe)
  - For single randomization at startup, expected number of probes: 2^{n-1}
  - For re-randomized n-bit random offset, expected number of probes: 2^n
  - Only twice as many probes needed as in attack when randomizing once at start!
  - Not promising…

TaintCheck: Detecting Exploits by Analyzing Server Execution

- Approach: instrument program to monitor its own execution, detect when exploit occurs

- Goals:
  - Work on binaries (no source code required)
  - Low false positives/false negatives
  - Detect wide range of exploits (new varieties all the time; point solutions unconvincing)
  - Help humans understand how exploit worked, after the fact; how did data flow from malicious input to point of exploit?

TaintCheck:
Basic Execution Monitoring Idea

- Many exploits use data supplied by user (or derived from data supplied by user) to subvert control flow of program
  - Need to modify jump, call instruction target addresses, or function return addresses
- During execution, before any control transfer instruction, validate target address not derived from user-supplied data
  - If it is, exploit detected; raise alarm
  - If it isn't, continue execution normally

Tainting User Input and Data Derived from It

- User is the source of exploits; don't trust data from him

- Mark all data from user (received from network, or from input files) as tainted

- Propagate taint during execution
  - Results of operations on tainted data should be tainted
  - Copies of tainted data should be tainted

- Clear taint when tainted data overwritten with untainted data

- How do we get a precompiled program executable to behave this way?

Valgrind: Modifying Executables at Runtime

- Run executable under Valgrind system
- Give Valgrind instructions on how to instrument executable
  - literally, what instructions or function calls to search for, and what instructions to add to them
- Valgrind's processing loop:
  - Fetch next basic block of program (dictated by IP/PC)
  - Translate code into UCode, Valgrind's instruction set
  - Add instrumentation code to Valgrind UCode
  - Translate code back to x86; cache for reuse
  - Execute instrumented x86 basic block
  - Repeat…

Adding Instrumentation: Tracking Tainted Data

- After I/O system calls:
  - If reading from socket, mark target buffer contents as tainted

- After all memory load instructions:
  - If source memory tainted, mark register tainted
  - If source memory untainted, mark register untainted

- After all memory store instructions:
  - If source register tainted, mark memory tainted
  - If source register untainted, mark memory untainted

- After all arithmetic instructions:
  - If any operand tainted, mark result tainted
  - If no operands tainted, mark result untainted

Adding Instrumentation: Detecting Invalid Uses of Tainted Data

- Before all control transfer instructions, add code:

  - If register or memory location holding target function pointer is tainted, raise alarm

  - Means derived from user input; should never happen!

- Needed before each jump, call, ret

Tracking Taint: Shadow Memory

- For every byte of memory, keep shadow memory that tracks taint status
- Simple interface:
  - Is-Tainted(addr) -> {T | F}
  - Taint(addr, len), Untaint(addr, len)
- Two modes of operation
  - Fast: single bit for each byte of memory
  - Detailed: 4-byte pointer to Taint data structure, containing details of system call, stack, value; written at time of tainting
  - Detailed mode useful for analysis of exploits
- Implementation greatly affects performance
  - Space vs. time tradeoff: packed vs. unpacked

Corner Case: Implicit Flows

- Suppose x tainted, then execute:
```
if (x == 0)
    y = 0;
else
    y = 1;
```
- TaintCheck <span style="color:red">doesn't taint processor condition flags</span>
  - Would often result in <span style="color:red">inappropriate propagation of taint; false positives</span>
- But x clearly influences value of y, and y could later influence other values
- Result: <span style="color:red">false negatives are possible</span>
  - e.g., image compression bit-twiddling code?

# Exploit Detection Coverage



- **Default Policy** (green)
- **Optional Policy** (yellow)

| | Format String | Buffer Overflow | Double Free | Heap Smash |
|---|---|---|---|---|
| Return Address | Default | Default | Default | Default |
| Jump Address | Default | Default | Default | Default |
| Function Pointer | Default | Default | Default | Default |
| Fn Ptr Offset | Default | Default | Default | Default |
| System Call Args | Default | Optional | Optional | Optional |
| Function Call Args | Default | Optional | Optional | Optional |

- TaintCheck can also instrument function and system calls
- e.g., check printf()-like library calls for tainted format string args
- Built system successfully detects many overwrite exploits (return address, function pointer, format string, GOT entry)

34

# TaintCheck's Performance: Monitoring Apache
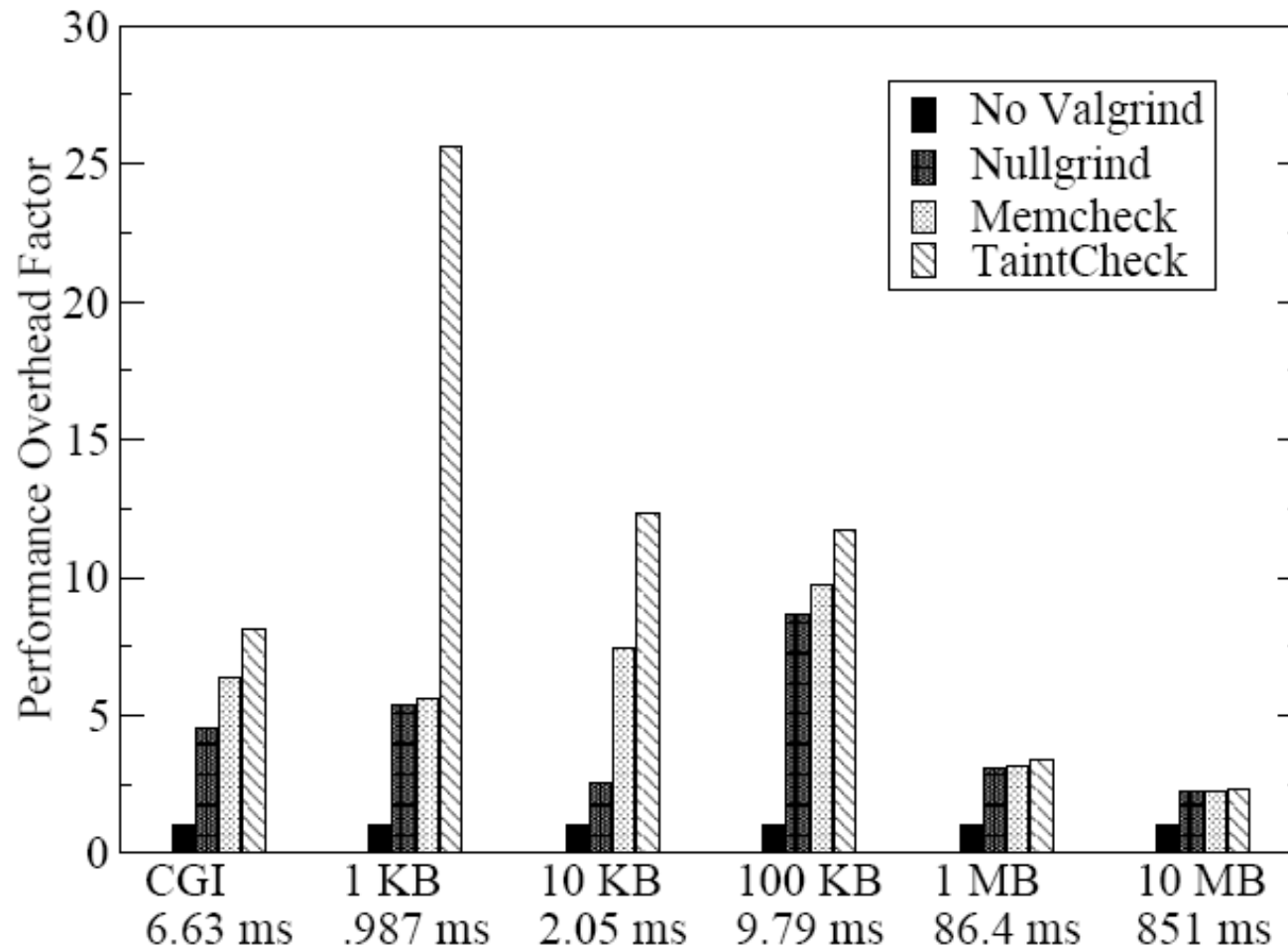


- Lots of extra instructions…
- Exec time not really right metric; throughput better metric

# TaintCheck: Modes of Use (1)

- Identify worm payloads
  - Can be configured to store trace of tainted data flow from all inputs
  - When exploit detected, can walk back to identify input that led to exploit
  - Could pass worm payloads to signature generation system, like Autograph
    - Much more accurate than port-scanner heuristic!
- Prevent exploit of server
  - Halt execution upon exploit detection

# TaintCheck: Modes of Use (2)

- Probably too slow for production servers
  - 25X server farm size increase for Amazon?
- Could possibly deploy on a few servers: sample traffic
  - Would slow detection of new worm, though; only sampling some inputs
  - Adversary may possibly be able to detect monitored servers by their slow response time; avoid sending them exploit payload