CDDC 2019 Qualifiers

Challenge Writeup

Team: S4F3 S0L1D5

Category: Uni/Poly

June 2019

PART I

Reverse Engineering

1 LSCVM: Immaculate Invasion

1.1 Solution

The tool of choice for this challenge was *Cutter*[1], an open-source reverse-engineering framework which performs disassembly, function analysis, and function/value renaming, among other things.

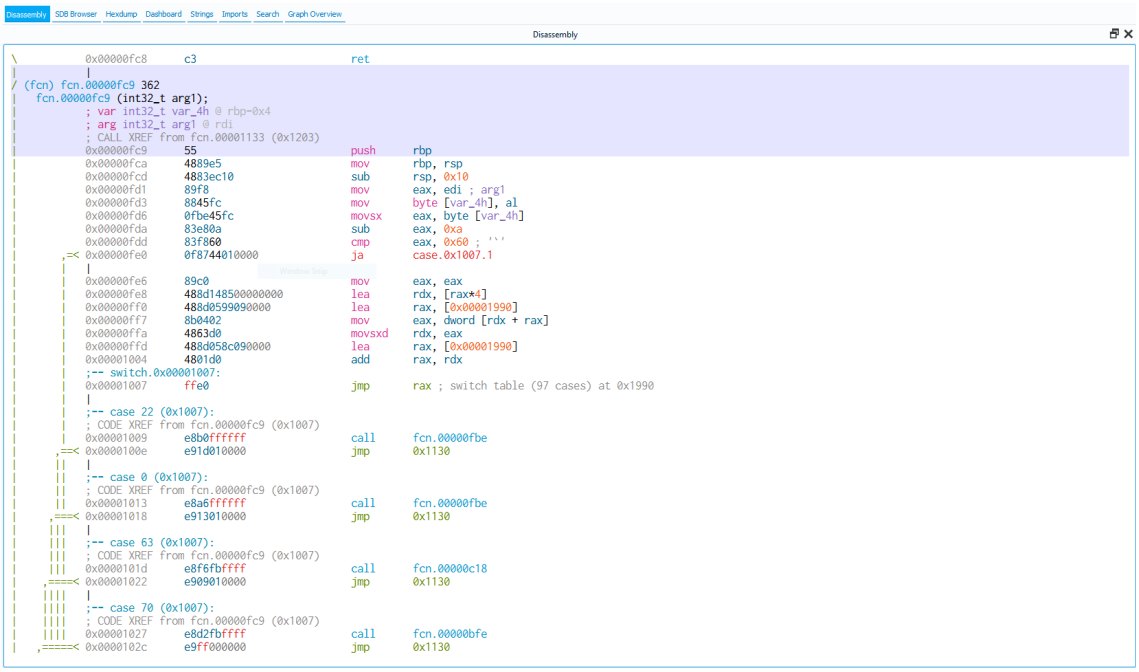

Figure 1.1: A screenshot of Cutter

The first hurdle was getting the program to run beyond its error message of `[-] Flag file open error`; upon inspection of the disassembly, the program tries to open a file named 'flag', which presumably contains the flag on the server-side — `$ touch flag` convinced the program to start.

[1] https://github.com/radareorg/cutter

Given that the program asks for a login ID, the next step was to check for strings and `strcmp`s in the code, and one was indeed found:

```
call      fcn.00001133
lea       rsi, str.lsc_user ; 0x2065 ; "lsc_user" ; const char *s2
lea       rdi, [0x00203140] ; 0x203140 ; const char *s1
call      sym.imp.strcmp ; int strcmp(const char *s1, const char *s2)
test      eax, eax
je        0x14b3
```

Figure 1.2: A suspicious `strcmp`

Of course, no challenge is so easy, especially one that sat at above 950 points more than 24 hours after the qualifiers started. On further inspection, the operand of `strcmp` appeared to be the output of a call to `fcn.00001133`, which itself took some very long and suspicious strings as input: `eiMaAghMcAgjcMMdAgjcMMaAgjcMMaAhhcMMdAijMaAcfMPPPPPPPPP`.

Another discovery was this instruction: `nasm`cmp dword [var_1174h], 2, where `var_1174h` contained the value of `argc`. Passing a second argument in the invocation (eg. `$ ./lscvm-ii x`) revealed a *very* useful debugging mode that printed what appeared to be a stack.

Following the call chain eventually led to the jump table for opcode dispatch, pictured in Figure 1.1. Some quick analysis revealed that this was a indeed stack-based virtual machine (*thankfully*). Combined with the debugging output, and the fact that the text output (the banner, login prompt, etc.) was printed using the VM instead of say `printf`, some basic opcodes were decoded:

Opcode	Pops	Pushes	Description
u to **u**	–	0 to 9	constant
A	a, b	a + b	add
M	a, b	a * b	multiply
P	a	–	`printf("%c",a)`

Piecing things together, it was inferred that we were most likely supposed to input a *program* when asked for the user ID, which would then print `lsc_user` as required. Given that ASCII values for the lowercase alphabets start at 97, a small program was quickly written that would take in a string and output opcodes that would print it (Listing 1.1).

After trying it out, however, we were quickly met with disappointment, as we were faced with our old friend `[-] Wrong id`.

After more digging, the user id was expected to be at the memory address `0x203140`; using the renaming feature of Cutter, we could make it easier to spot:

```
call      fcn.00001133
lea       rsi, str.lsc_user ; 0x2065 ; "lsc_user" ; const char *s2
lea       rdi, var.mem_base.203140 ; 0x203140 ; const char *s1
call      sym.imp.strcmp ; int strcmp(const char *s1, const char *s2)
test      eax, eax
je        0x14b3
```

Figure 1.3: `[0x203140]` after being renamed into `var.mem_base.203140`

Scrolling through the opcode list while looking for the address (now renamed) yielded this very interesting function, mapped to opcode K:

```
(fcn) fcn.00000e22 86
  fcn.00000e22 ();
          ; var signed int var_4h @ rbp-0x4
          ; CALL XREF from fcn.00000fc9 (0x1107)
          0x00000e22      55                      push      rbp
          0x00000e23      4889e5                  mov       rbp, rsp
          0x00000e26      4883ec10                sub       rsp, 0x10
          0x00000e2a      e89cfdffff              call      fcn.00000bcb
          0x00000e2f      8945fc                  mov       dword [var_4h], eax
          0x00000e32      817dfc00ffffff          cmp       dword [var_4h], 0xffffff00
      ,=< 0x00000e39      7c09                    jl        0xe44
      |   |
      |   0x00000e3b      817dfcff3f0000          cmp       dword [var_4h], 0x3fff
     ,==< 0x00000e42      7e16                    jle       0xe5a
     ||   |
     ||   ; CODE XREF from fcn.00000e22 (0xe39)
     |`-> 0x00000e44      8b45fc                  mov       eax, dword [var_4h]
     |    0x00000e47      89c6                    mov       esi, eax
     |    0x00000e49      488d3de80a0000          lea       rdi, str.memory_write_access_violation__d ; 0x1938
     |    0x00000e50      b800000000              mov       eax, 0
     |    0x00000e55      e876fbffff              call      sym.imp.printf ; int printf(const char *format)
     |    |
     |    ; CODE XREF from fcn.00000e22 (0xe42)
     `--> 0x00000e5a      e86cfdffff              call      fcn.00000bcb
          0x00000e5f      89c1                    mov       ecx, eax
          0x00000e61      8b45fc                  mov       eax, dword [var_4h]
          0x00000e64      4863d0                  movsxd    rdx, eax
          0x00000e67      488d05d2222000          lea       rax, var.mem_base.203140 ; 0x203140
          0x00000e6e      880c02                  mov       byte [rdx + rax], cl
          0x00000e71      b800000000              mov       eax, 0
          0x00000e76      c9                      leave
          0x00000e77      c3                      ret
```

Figure 1.4: A function to ~~surpass metal gear~~ write to memory?

There was also an accompanying opcode **E**, which read a value from memory. The memory buffer appeared to be an array of 32-bit values, and opcodes **K** and **E** took an index into this array. Slight modifications were made to `lscvm-deasciiinator` to yield `lscvm-memoryinator` (basically keeping track of an offset and replacing 'P' with 'K'), which would generate opcodes to write a string into a given offset in memory.

Finally, then, the challenge could be solved (the password was also in plain sight, `hi_darkspeed-corp!`):

```
$ ./lscvm-memoryinator
string: lsc_user
address: 0

ggdMMaKfgAfMcMfAbKjjMjAjAcKgfdMMfAdKfgAfMcMhAeKfgAfMcMfAfKcf
McfMMbAgKfgAfMcMeAhK

string: hi_darkspeed-corp!
address: 0

jeAiMaKhdfMMbKgfdMMfAcKcfMcfMMdKjjMjAhAeKfgAfMcMeAfKhdfMMcAg
KfgAfMcMfAhKfgAfMcMcAiKcfMcfMMbAjKcfMcfMMbAcfMKcfMcfMMbcfMMb
AKfddMMbcfMMcAKjjMjAjAbcfMMdAKfgAfMcMbAbcfMMeAKfgAfMcMeAbcfM
MfAKfgAfMcMcAbcfMMgAKfgAdMbcfMMhAK
^C

$ nc lscvm-ii.cddc19q.ctf.sg 9001

=== Welcome to LSCVM(LightSpeed Corp Virtual Machine) ===
ID : ggd...AhK
Password : jeA...hAK

Login Successful! $CDDC19${IcY_GrE37ings_Fr0M_LigHT5pEeDC0Rp}

lsc_user, Good Bye!
```

1.2 Flag

The flag for this challenge was `$CDDC19${IcY_GrE37ings_Fr0M_LigHT5pEeDC0Rp}`.

1.3 Further Analysis

While we were unable to actually come up with a quine to submit for the next LSCVM challenge (Quintessential Harlequin), we continued to take apart the VM (on the advice that it would be used again in the Finals!), since solving `lscvm-ii` did not require all of the opcodes (far from it).

Again, the renaming feature of Cutter was extremely helpful, and we were able to discover the address of the program counter, the base address of the stack, and (in the case of `lscvm-qh`), the mirror of stdout. Each buffer appears to be a identical with a fixed size of `0x4e200` bytes (`0x13880` 32-bit words), and the number of items stored after the last element (ie. at offset `0x4e200` from the base address).

We (eventually) managed to decipher all of the opcodes and their purpose:

Opcode	Pops	Pushes	Description
a to **j**	–	`0` to `9`	constant
A	`a`, `b`	`a + b`	add
B	–		stop execution immediately
C	`x`	–	call (jump to absolute instruction `x`)
D	`x`	–	pop (drop)
E	`addr`	`value`	read memory from `addr`
F	`ofs`	`value`	fetch from stack (`ofs` elms below top)
G	`ofs`	–	relative jump forward
H	`ofs`	`value`	same as **F**, but removes the element
I	`x`	–	`printf("%d",x)`
J	`a`, `b`	`cmp`	-1 if $a < b$, 0 if $a = b$, 1 if $a > b$
K	`val`, `addr`	–	writes `val` to memory at `addr`
M	`a`, `b`	`a * b`	multiply
P	`x`	–	`printf("%c",x)`
R	–	–	return
S	`a`, `b`	`a - b`	subtract
V	`a`, `b`	`a / b`	integer divide
Z	`cond`, `ofs`	–	jump (relative) if `cond` is 0

Of note are the **C** and **R** opcodes, which *call* and *return* respectively. There is another array which is only accessed by these instructions that functions as a *callstack*. **C** pushes the current program counter (*PC*) to this callstack, and **R** pops the return address from the callstack, and sets *PC* to it, moving execution back to the callsite.

PART II

Miscellaneous

2 Super Strong TeleVision

2.1 Solution

SSTV or Slow Scan television is a picture transmission method used mainly by amateur radio operators to transmit and receive static pictures via radio in monochrome or colour[2].

The `.wav` file provided in the challenge is an SSTV-encoded image. Using *PulseAudio* was used as a link between *VLC* and *QSSTV*[3] (an open-source Linux SSTV application), we were able to decode the SSTV image.

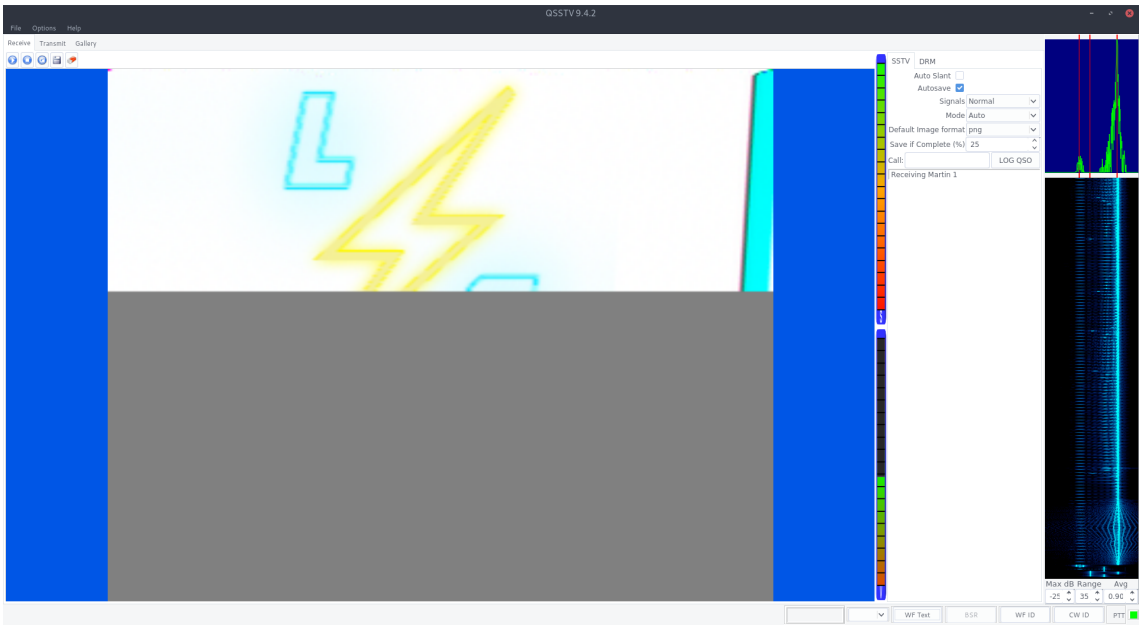

Figure 2.1: QSSTV decoding the image; its frequency spectrum can be seen on the right

[2] en.wikipedia.org/wiki/Slow-scan_television
[3] http://users.telenet.be/on4qz/qsstv/index.html

In this case, *PulseAudio* acts as the pipe that redirects the `.wav` file as an input into the decoder. To achieve that, we first set up a virtual *null sink* and configure the system in the following manner:

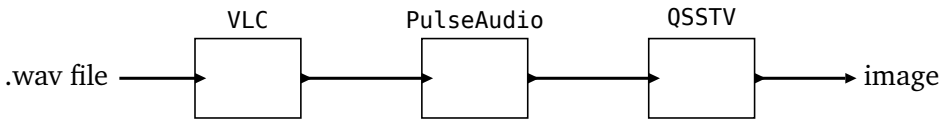

Figure 2.2: Data Pipeline Setup

2.2 Flag

After some suspense as the SSTV audio played back in real time, the image was fully decoded, revealing the flag, which was `$CDDC19${Light$peedCorp-$$TV}`.

Figure 2.3: Decoded SSTV Image

PART III

Appendix

1 Code Listings

1.1 `lscvm-deasciiinator`

```cpp
// lscvm-deasciiinator.cpp

#include <stdio.h>

#include <map>
#include <vector>
#include <string>
#include <iostream>

int main()
{
	std::map<char, std::string> lookup;

	lookup['a'] = "jjMjAhA";
	lookup['b'] = "jjMjAiA";
	lookup['c'] = "jjMjAjA";
	lookup['d'] = "cfMcfMM";
	lookup['e'] = "cfMcfMMbA";
	lookup['f'] = "jiAcdMM";
	lookup['g'] = "jiAcdMMbA";
	lookup['h'] = "jeAiM";
	lookup['i'] = "hdfMM";
	lookup['j'] = "hdfMMbA";
	lookup['k'] = "hdfMMcA";
	lookup['l'] = "ggdMM";
	lookup['m'] = "ggdMMbA";
	lookup['n'] = "fgAfMcM";
	lookup['o'] = "fgAfMcMbA";
	lookup['p'] = "fgAfMcMcA";
	lookup['q'] = "fgAfMcMdA";
	lookup['r'] = "fgAfMcMeA";
	lookup['s'] = "fgAfMcMfA";
```

```
33          lookup['t'] = "fgAfMcMgA";
34          lookup['u'] = "fgAfMcMhA";
35          lookup['v'] = "fgAfMcMiA";
36          lookup['w'] = "fgAfMcMjA";
37          lookup['x'] = "gcfcMMM";
38          lookup['y'] = "fgAfgAM";
39          lookup['z'] = "fgAfgAMbA";
40          lookup['_'] = "gfdMMfA";
41          lookup['!'] = "fgAdM";
42          lookup['-'] = "fddMM";
43
44
45          std::string input;
46
47          while(true)
48          {
49                  std::getline(std::cin, input);
50
51                  std::string output;
52                  for(size_t i = input.size(); i-- > 0; )
53                          output += lookup[input[i]];
54
55                  for(size_t i = 0; i < input.size(); i++)
56                          output += "P";
57
58                  printf("\n%s\n", output.c_str());
59          }
60  }
```

Listing 1.1: lscvm-deasciiinator.cpp