CDDC 2019 Qualifiers

Challenge Writeup

Team: `S4F3 S0L1D5`

Category: Uni/Poly

June 2019

PART I

OSINT_RED

1 [R-0] Everyone <3 Fan Mail

This series of challenges were made slightly more tedious than necessary, because unfortunately there was a real company called ‘LightSpeed’ that diluted the search results.

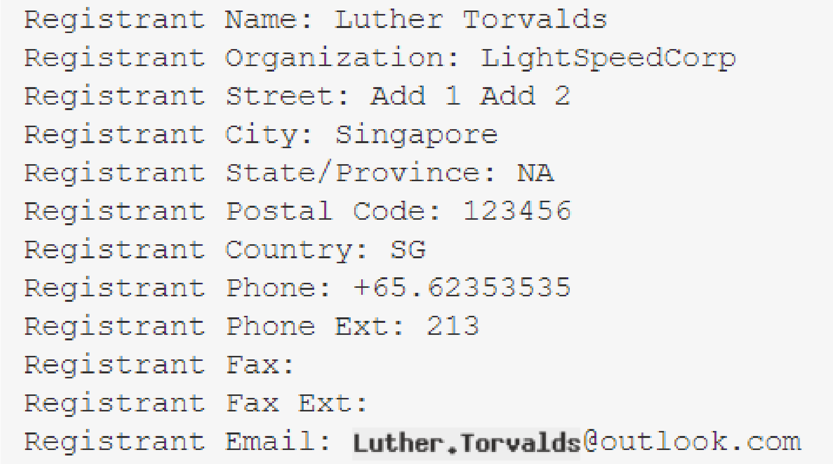
The main objective of this challenge was to find the email of the site administrator for `www.lightspeedcorp.global`, and send him an email.

1.1 Solution

In theory, all we would have needed to do was perform a WHOIS lookup on the domain to find the email address. Simple, right?

Nope.

Depending on which WHOIS search engine is used, certain bits of information about the registrant can be missing if the domain was registered with WHOISGuard, which hides certain identifiable information from WHOIS lookups.



```
Registrant Name: Luther Torvalds
Registrant Organization: LightSpeedCorp
Registrant Street: Add 1 Add 2
Registrant City: Singapore
Registrant State/Province: NA
Registrant Postal Code: 123456
Registrant Country: SG
Registrant Phone: +65.62353535
Registrant Phone Ext: 213
Registrant Fax:
Registrant Fax Ext:
Registrant Email: Luther.Torvalds@outlook.com
```

Figure 1.1: All the information is visible if WHOISGuard is ignored

However, it turns out that the guard isn't perfect, and all we had to do was find a WHOIS search engine that ignores the guard. After some digging, we found one, exposing all the juicy details of the registrant.

After getting the email address, we just had to send an email, and we were replied with the flag.

1.2 Flag

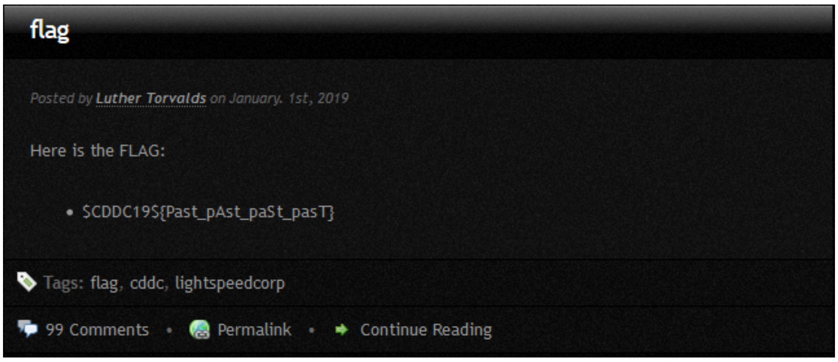
The flag for this challenge was `$CDDC19${IS_IT_I_AM_FAM0US_NAO}`.

2 [R-1] Travel to the Past

The challenge heavily hinted that we would need a way to look at an older version of the website to find the flag, and that's exactly what we did.

2.1 Solution

There was a single snapshot of the website on the Wayback Machine[1] on the 20th of May, and browsing that snapshot revealed the flag on the homepage of a blog:



flag

Posted by Luther Torvalds on January. 1st, 2019

Here is the FLAG:

- $CDDC19${Past_pAst_paSt_pasT}

Tags: flag, cddc, lightspeedcorp

99 Comments • Permalink • Continue Reading

Figure 2.1: The flag is clearly visible

2.2 Flag

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

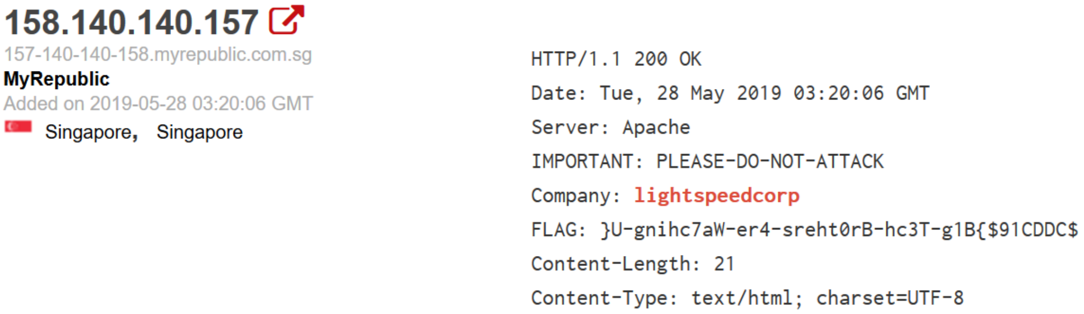
[1] https://archive.org/web/

3 [R-2] I'm Sho Done With This

3.1 Solution

The hint was in the question title itself — turns out that there is a search engine called 'ShoDan' [2], which claims to be the world's first search engine for internet connected devices.

Sure enough, searching for our favourite company 'lightspeedcorp' yielded the following results:



158.140.140.157
157-140-140-158.myrepublic.com.sg
MyRepublic
Added on 2019-05-28 03:20:06 GMT
Singapore, Singapore

```
HTTP/1.1 200 OK
Date: Tue, 28 May 2019 03:20:06 GMT
Server: Apache
IMPORTANT: PLEASE-DO-NOT-ATTACK
Company: lightspeedcorp
FLAG: }U-gnihc7aW-er4-sreht0rB-hc3T-g1B{$91CDDC$
Content-Length: 21
Content-Type: text/html; charset=UTF-8
```

Figure 3.1: The flag is reversed, for some reason

A trivial string-reverse later, the flag is obtained.

3.2 Flag

The flag for this challenge was `$CDDC19${B1g-T3ch-Br0thers-4re-Wa7ching-U}`.

[2] https://www.shodan.io

PART II

Programming

4 Count 1: Baby

This was a fairly simple code golfing challenge; inspection of the provided `count1-baby.py` file gave the character limit as 53 characters. The starting code is given as this:

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        int i;

        for( i = 1 ; i < 10000 ; i++ )
        {
                printf("%d,", i);
        }

        return 0;
}
```

4.1 Solution

Compiling and running the program shows that the expected output of the minified program are the numbers `1` to `9999`, separated by commas (with a trailing comma after `9999`).

Besides the obvious steps like removing whitespace and newlines, knowledge of the C standard and ignorance of compiler warnings can yield the following transformations:

- `printf` is an implicitly defined standard library function; `stdio.h` can be removed
- Functions do not require a type specifier; `int main` can be shortened to `main`
- `main` does not need to take arguments, leaving `main()`
- The verifier rejects spaces, so `int i` has to be changed to `int(i)` (which is valid)
- `main` does not need to explicitly `return 0`

This yields the final program, at 53 characters long:

```
main(){int(i);for(i=1;i<10000;i++){printf("%d,",i);}}
```

Listing 4.1: The solution for Count 1: Baby

4.2 Flag

The flag for this challenge was `$CDDC19${Count2_is_waiting_Please_enjoy!}`.

5 Count 2: Wildness

The required output for this challenge is the same as the previous challenge (Count 1: Baby), but the restrictions have been tightened. This time, OP's friend 'doesn't like to go wild', or, in other words, the following characters cannot be used in the source code: `<== w1lD ==>`. Additionally, the character limit has been reduced to 41.

5.1 Solution

Starting from the code for Count 1 (Listing 4.1), we can apply some more non-obvious tricks:

- Variables don't need a type specifier, and will default to `int`
- Variables with static storage duration will be initialised to `0` (obviating the need for `=`)
- `!=` is equivalent to a bitwise XOR (`^`) for integers (removing `<`)
- Taking advantage of `i++` semantics can replace `1e4` with `9999` (since `1` can't be used)

This yields the final program, again at exactly 41 characters:

```
i;main(){for(;i++^9999;printf("%d,",i));}
```

Listing 5.1: The solution for Count 2: Wildness

5.2 Flag

The flag for this challenge was `$CDDC19${This_really_helps_m3_a_lot}`

6 Count 4: Madness - Filter

For this challenge, the character limit is slightly relaxed to 44 characters, but OP's friend has a faulty keyboard now, and all lowercase characters except those in '`mad printf`' ('a', 'd', 'f', 'i', 'm', 'n', 'p', 'r', and 't') cannot be used.

6.1 Solution

Given that all the looping constructs (`for`, `while`, and even `goto`) cannot be used due to having illegal characters, the only logical solution is recursion.

- It is not illegal to call `main` recursively
- When called with no arguments, `argc` is `1`; so `main(i)` will initialise `i` to `1`
- `1E4` is equally as valid as `1e4`, which is identical to `10000`
- The first arm of the ternary operator can be omitted: `x?:y` is valid.

We came up with two independent solutions to this problem:

```
main(i){printf("%d,",i++);if(i<1E4)main(i);}
```

Listing 6.1: The solution for Count 2: Wildness

```
i;main(){printf("%d,",++i);i>=9999?:main();}
```

Listing 6.2: An alternative solution for Count 2: Wildness

In both cases, the code was exactly 44 characters long.

6.2 Flag

The flag was `$CDDC19${Main_might_be_just_a_function_but_it_is_really_special!}`.

PART III

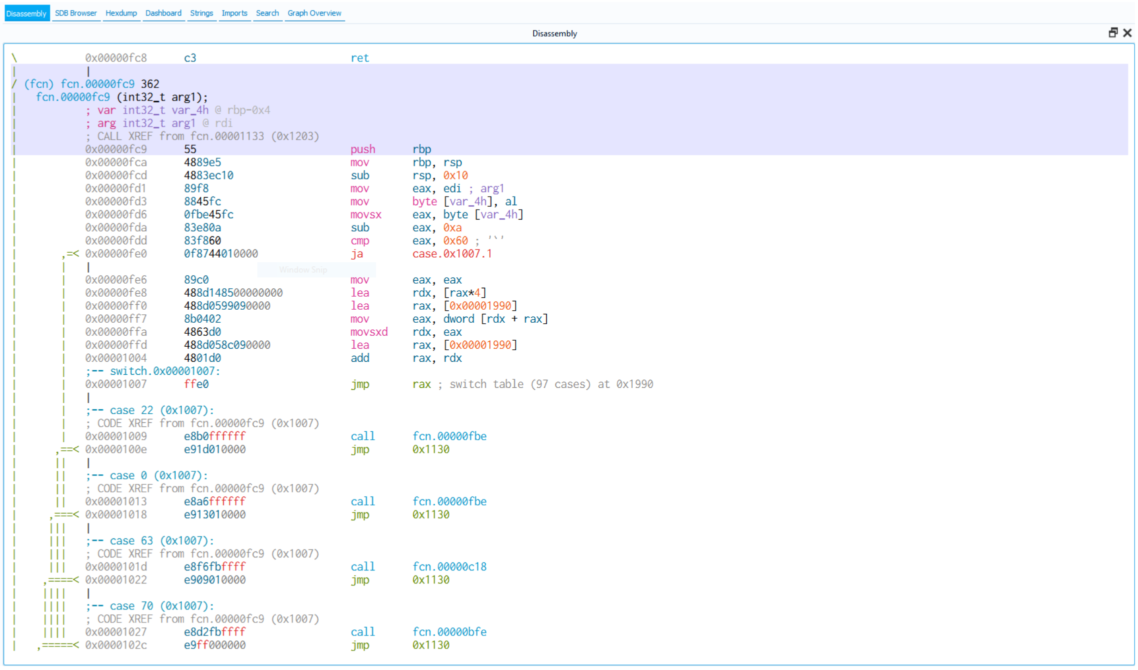
Reverse Engineering

7 LSCVM: Immaculate Invasion

Presented with a login prompt, the logical conclusion was that the flag would be accessible upon successfully logging into the server.

7.1 Solution

The tool of choice for this challenge was *Cutter*[3], an open-source reverse-engineering framework which performs disassembly, function analysis, and function/value renaming, among other things.



```
\          0x00000fc8      c3                      ret
|          |
/ (fcn) fcn.00000fc9 362
|   fcn.00000fc9 (int32_t arg1);
|          ; var int32_t var_4h @ rbp-0x4
|          ; arg int32_t arg1 @ rdi
|          ; CALL XREF from fcn.00001133 (0x1203)
|          0x00000fc9      55                      push    rbp
|          0x00000fca      4889e5                  mov     rbp, rsp
|          0x00000fcd      4883ec10                sub     rsp, 0x10
|          0x00000fd1      89f8                    mov     eax, edi ; arg1
|          0x00000fd3      8845fc                  mov     byte [var_4h], al
|          0x00000fd6      0fbe45fc                movsx   eax, byte [var_4h]
|          0x00000fda      83e80a                  sub     eax, 0xa
|          0x00000fdd      83f860                  cmp     eax, 0x60 ; '`'
|      ,=< 0x00000fe0      0f8744010000            ja      case.0x1007.1
|      |   |
|      |   0x00000fe6      89c0                    mov     eax, eax
|      |   0x00000fe8      488d148500000000        lea     rdx, [rax*4]
|      |   0x00000ff0      488d0599090000          lea     rax, [0x00001990]
|      |   0x00000ff7      8b0402                  mov     eax, dword [rdx + rax]
|      |   0x00000ffa      4863d0                  movsxd  rdx, eax
|      |   0x00000ffd      488d058c090000          lea     rax, [0x00001990]
|      |   0x00001004      4801d0                  add     rax, rdx
|      |   ;-- switch.0x00001007:
|      |   0x00001007      ffe0                    jmp     rax ; switch table (97 cases) at 0x1990
|      |   |
|      |   ;-- case 22 (0x1007):
|      |   ; CODE XREF from fcn.00000fc9 (0x1007)
|      |   0x00001009      e8b0ffffff              call    fcn.00000fbe
|     ,==< 0x0000100e      e91d010000              jmp     0x1130
|     ||   |
|     ||   ;-- case 0 (0x1007):
|     ||   ; CODE XREF from fcn.00000fc9 (0x1007)
|     ||   0x00001013      e8a6ffffff              call    fcn.00000fbe
|    ,===< 0x00001018      e913010000              jmp     0x1130
|    |||   |
|    |||   ;-- case 63 (0x1007):
|    |||   ; CODE XREF from fcn.00000fc9 (0x1007)
|    |||   0x0000101d      e8f6fbffff              call    fcn.00000c18
|   ,====< 0x00001022      e909010000              jmp     0x1130
|   ||||   |
|   ||||   ;-- case 70 (0x1007):
|   ||||   ; CODE XREF from fcn.00000fc9 (0x1007)
|   ||||   0x00001027      e8d2fbffff              call    fcn.00000bfe
|  ,=====< 0x0000102c      e9ff000000              jmp     0x1130
```

Figure 7.1: 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.

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

Given that the program asks for a login ID, the next step was to check for strings and strcmps 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 7.2: 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: eiMaAghMcAgjcMMdAgjcMMaAgjcMMaAhhcMMdAijMaAcfMPPPPPPPP.

Another discovery was this instruction: `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.

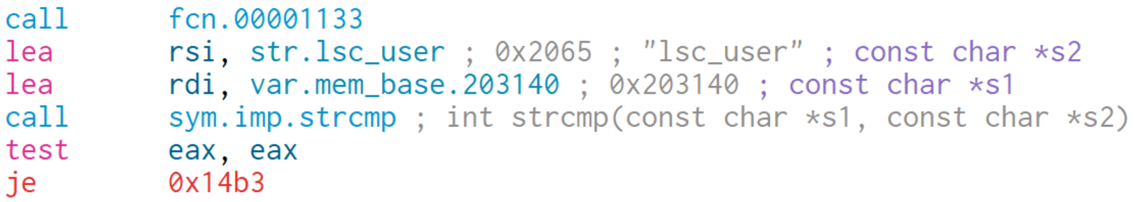
Following the call chain eventually led to the jump table for opcode dispatch, pictured in Figure 7.1. Some quick analysis revealed that this was a indeed stack-based virtual machine (*thankfully*). Combined with the debugging output, and the fact that the text output (the banner, login prompt, etc.) was printed using the VM instead of say printf, some basic opcodes were deciphered:

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.

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

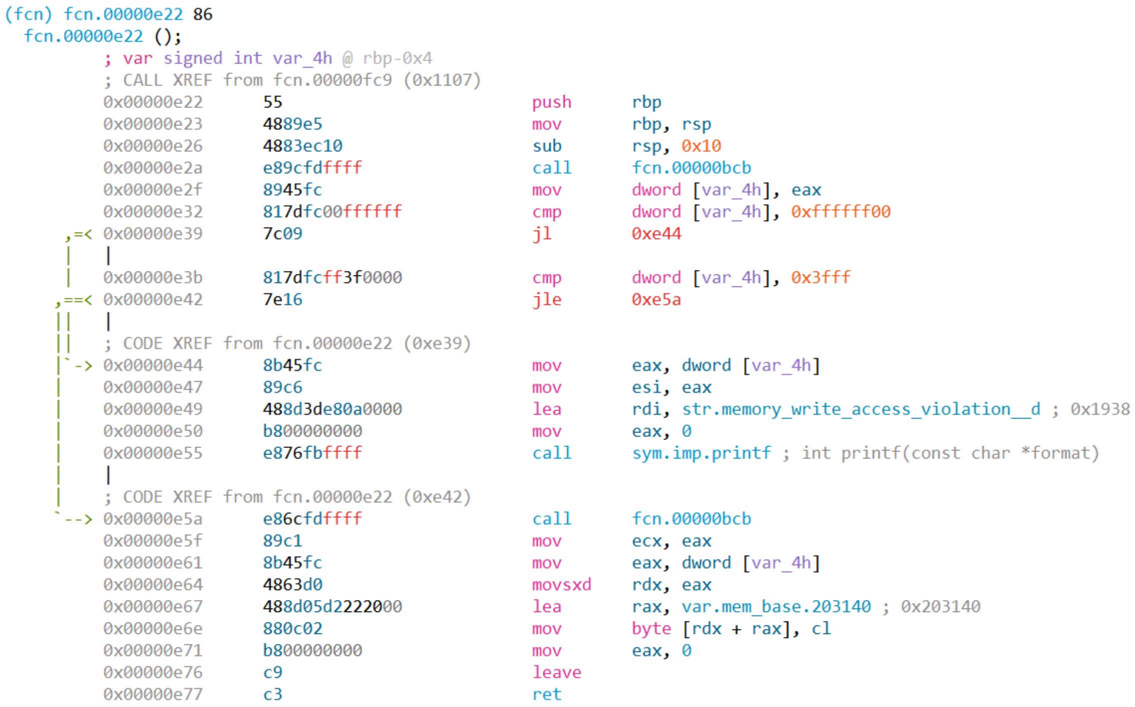
After more digging, the user id was expected to be at the memory address `0x203140`; using the renaming feature of Cutter, we could make it easier to spot:



```
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 7.3: `[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 7.4: 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` — keeping track of an offset and replacing 'P' with 'K' — which would generate opcodes to write a string into a given offset in memory, yielding `lscvm-memoryinator` (Listing 1.1).

The password was also in plaintext alongside the username: `hi_darkspeed-corp!`. Finally, then, the challenge could be solved:

```
$ ./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!
```

7.2 Flag

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

7.3 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 identically constructed 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 IV

Miscellaneous

8 Polyglot

What to do, when presented with languages that we can't understand?

8.1 Solution

The obvious solution was to put each sentence into Google Translate[4]; all 10 sentences were some variation of 'the first letter of this language is the flag'. Coupled with the input format, given as `[01][02][03][04][05][06]&[07][08][09][10]!`, we were able to decode the flag.

Number	Language	Sentence
01	Hindi	इस भाषा का पहला चरित्र झंडा बनाता है।
02	Indonesian	Karakter pertama bahasa ini yang mengibarkan bendera.
03	Chinese	这种语言的第一个字符构成了旗帜。
04	Dutch	Het eerste teken van deze taal vormt de vlag.
05	Danish	Det første tegn på dette sprog udgør flag.
06	Catalan	El primer caràcter d'aquest idioma constitueix la bandera.
07	Norwegian	Det første tegnet av dette språket utgjør flagget.
08	Spanish	El primer carácter de este lenguaje lo constituye la bandera.
09	Hmong	Thawj qhov cim ntawm hom lus no ua rau tus chij.
10	Croatian	Prvi znak ovog jezika čini zastavu.

Assembled, the message was `HI~CDDC&NSHC`; it helped that there was a coherent message to verify that Google Translate didn't misdetect any of the langauges.

8.2 Flag

The flag for this challenge was `$CDDC19${HI~CDDC&NSHC}`.

[4] https://translate.google.com

9 Do You Fancy Numbers?

We are presented with a picture of what are apparently numbers. On first guess they appear to be Chinese in nature.

9.1 Solution

Indeed, a quick trawl of Wikipedia led us to Suzhou numerals[5], which contained everything we needed to decode the message.

After substituting them for arabic numerals, we get the following string of numbers: `36 67 68 68 67 49 57 36 123 53 48 95 121 48 117 95 102 52 78 99 89 95 102 108 48 87 51 114 95 78 117 77 98 51 82 53 125`.

Putting them through an ASCII decoder yields the following Base64 encoded string: `JCBDIEQgRCBDIDEgOSAkIHsgNSAwIF8geSAwIHUgXyBmIDQgTiBjIFkgXyBmIGwgMCBXIDMgciBfIE4gdSBNIGIgMyBSIDUgfQ==`

Finally, putting that through a Base64 decoder yields the flag (with some spaces).

9.2 Flag

The flag for this challenge was `$CDDC19${50_y0u_f4NcY_fl0W3r_NuMb3R5}`.

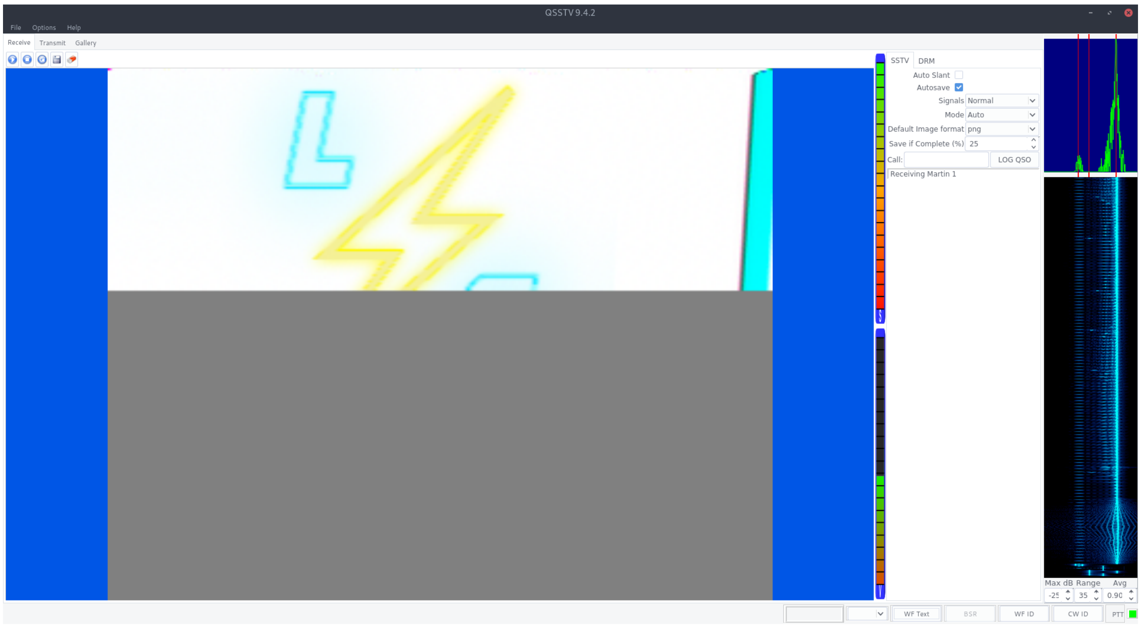
[5] https://en.wikipedia.org/wiki/Suzhou_numerals

10 Super Strong TeleVision

With the obvious hints in the challenge statement, we got to work decoding. 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[6]. It can also be used to hide easter[7] eggs[8]...

10.1 Solution

The solution was rather straightforward. Using *PulseAudio* as a link between *VLC* and *QSSTV*[9] (an open-source Linux SSTV application), we were able to decode the SSTV image rather quickly:



Auto Slant □
Autosave ☑
Signals: Normal
Mode: Auto
Default Image format: png
Save if Complete (%): 25
Call:
Receiving Martin 1

Figure 10.1: QSSTV decoding the image; its frequency spectrum can be seen on the right

In this case, PulseAudio acted as the pipe that redirects the .wav file as an input into the QSSTV decoder:

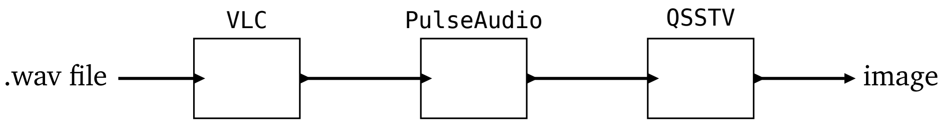


Figure 10.2: Data Pipeline Setup

[6] en.wikipedia.org/wiki/Slow-scan_television

[7] https://wiki.kerbalspaceprogram.com/wiki/List_of_easter_eggs

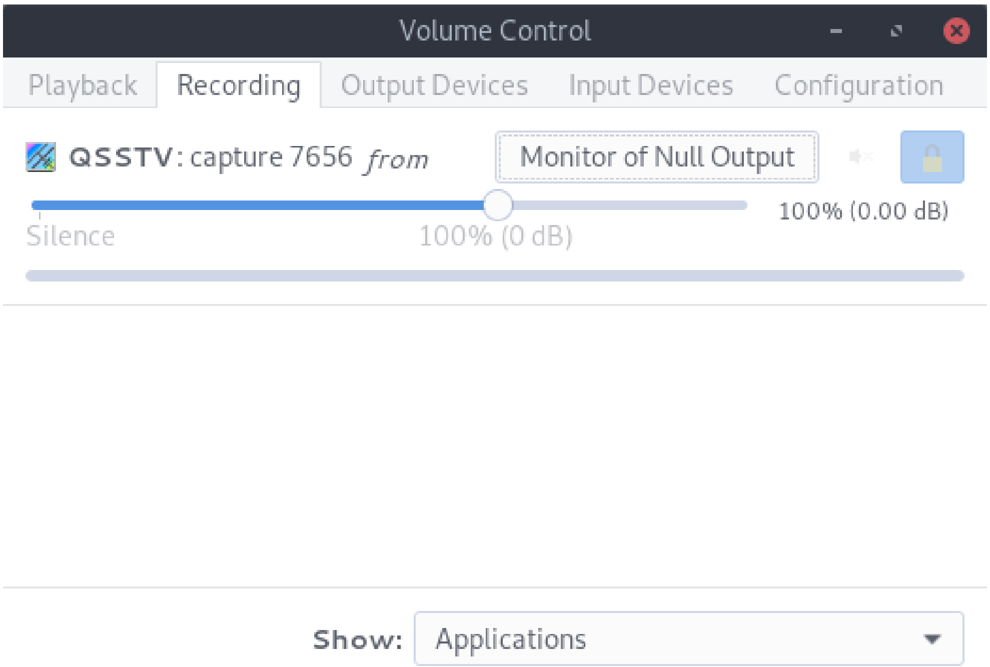
[8] https://half-life.fandom.com/wiki/Portal_ARG

[9] http://users.telenet.be/on4qz/qsstv/index.html

This works by enabling and making use of the default `null-sink` provided with every PulseAudio installation:

```
$ pactl load-module module-null-sink sink_name=virtual-cable
$ pavucontrol
```

and making QSSTV record from the `null sink`:



QSSTV: capture 7656 *from* Monitor of Null Output

100% (0.00 dB)

Show: Applications

Figure 10.3: QSSTV – Record from the `null-sink`

Once everything is set up, we simply start recording on QSSTV, use VLC to playback the `.wav` file to the `null-sink`, and watch the magic unfold.

10.2 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}`.

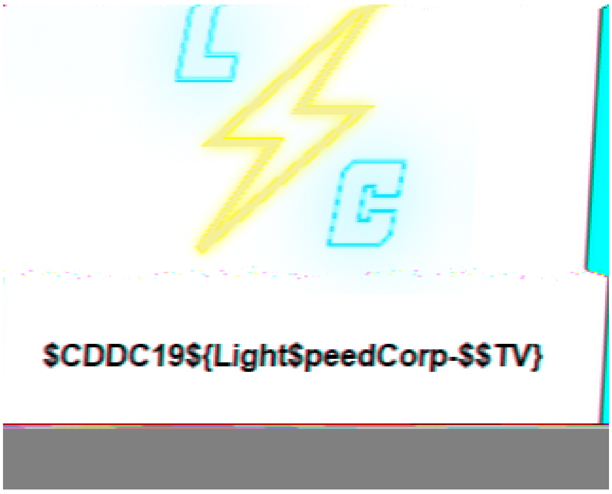


Figure 10.4: Decoded SSTV Image

PART V

Appendix

1 Code Listings

1.1 lscvm-memoryinator

```cpp
// lscvm-memoryinator.cpp

#include <stdio.h>

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

std::map<char, std::string> lookup;

std::string createNumber(int num)
{
        if(num < 10)
        {
                return std::string(1, (char) (num + 'a'));
        }
        else if(num == 10)
        {
                return "cfM";
        }
        else if((num >= 'a' && num <= 'z') || num == '-'
                || num == '_' || num == '!')
        {
                return lookup[num];
        }
        else
        {
                std::string ret;

                int x = num / 10;
                ret = createNumber(x) + createNumber(10) + "M";

                int y = num % 10;
                ret += createNumber(y) + "A";

                return ret;
        }
}


int main()
{
        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";
        lookup['t'] = "fgAfMcMgA";
        lookup['u'] = "fgAfMcMhA";
        lookup['v'] = "fgAfMcMiA";
        lookup['w'] = "fgAfMcMjA";
        lookup['x'] = "gcfcMMM";
        lookup['y'] = "fgAfgAM";
        lookup['z'] = "fgAfgAMbA";
        lookup['_'] = "gfdMMfA";
        lookup['!'] = "fgAdM";
        lookup['-'] = "fddMM";


        std::string input;

        while(true)
        {
```

```cpp
                printf("string: ");
                std::getline(std::cin, input);

                printf("address: ");
                std::string ofs;
                std::getline(std::cin, ofs);

                int offset = 0;
                if(!ofs.empty()) offset = std::stol(ofs);

                std::string output;
                for(size_t i = 0; i < input.size(); i++)
                {
                        // K writes to memory. format: K [value] [address]
                        output += createNumber(input[i]); // this is the value
                        output += createNumber(offset);   // this is the offset
                        output += "K";

                        offset++;
                }

                printf("\n%s\n", output.c_str());
        }
}
```

Listing 1.1: lscvm-memoryinator.cpp