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:

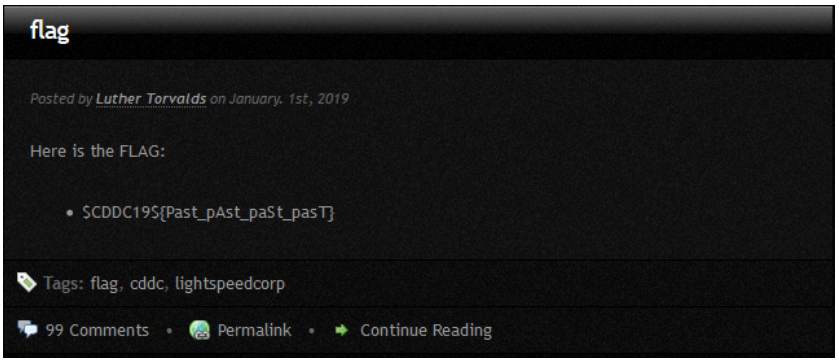

Figure 2.1: The flag is clearly visible

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:

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

4 [R-3-1] Have They Been Pwned?

The challenge description wasn't very helpful here...

4.1 Solution

This question was quite a challenge. The only hint from the question is that there has been a leak from LightSpeedCorp. With the Google search results being filled with the real life LightSpeed company, it was impossible to find relevant results.

The breakthrough came when we thought about where would leaks get posted — somewhere that would be anonymous but easy to access. After brainstorming and searching through multiple sites, we finally found the answer: Pastebin[3]. Searching for "LightSpeedCorp" in Pastebin yielded this page:

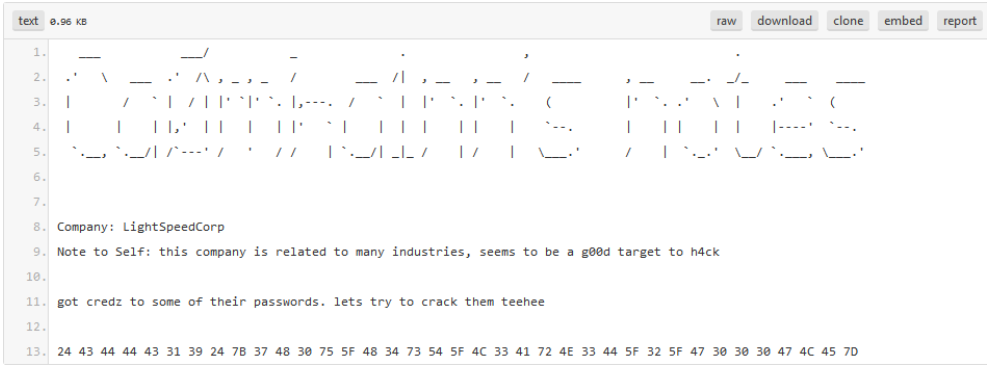

Figure 4.1: Oh, numbers again...

Interpreting the digits as hexdecimal and turning them into ASCII gives us our flag.

4.2 Part

The flag for this challenge was `$CDDC19${7H0u_H4sT_L3ArN3D_2_G000GLE}`

[3] https://pastebin.com

5 [R-4-1] Where I Get All My Memes From

Memes sound like fun!

5.1 Solution

With the name of our administrator obtained (Luther Torvalds), we just had to search him up on Twitter to reveal his profile, and the flag:

Figure 5.1: Oh, numbers again...

6 [R-4-1-2] Don't Be A Git

Git? Torvalds? Hmm...

6.1 Solution

Following the followers (hmm) of Luther, we found Bobby Hashlinger, who posted serveral GitHub links on his Twitter, one of which was `d4rkspeedcorp-framework`[4].

Upon further inspection, the flag (or something like it) was found in the branch `super-new-feature`, in the latest commit:

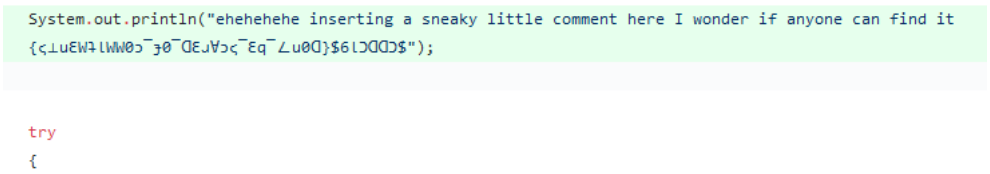

Figure 6.1: Oh, numbers again...

The more challenging part was to find the correct tool to make the text upright again... after some searching, we found it[5], and the flag.

6.2 Flag

The flag for this challenge was `$CDDC19${D0n7_b3_5cAr3D_0f_c0MM1tM3nT5}`

[4] https://github.com/sjang3141592653/d4rkspeedcorp-framework
[5] https://www.toolpond.com/tool/text-flipper

PART II

OSINT_BLUE

7 [B-0] What's in, Doc?

The challenge title is quite a big hint.

7.1 Solution

Word document files (`.docx`) are actually zip files that can be extracted to reveal a bunch of XML files that make up the actual content of the document.

Of course, we checked that the document didn't contain anything interesting before extracting it, which revealed a number of interesting files:

```
$ unzip test-docx.docx
Archive:  test-docx.docx
  inflating: [Content_Types].xml
  inflating: _rels/.rels
  inflating: word/document.xml
  inflating: word/_rels/document.xml.rels
  inflating: word/theme/theme1.xml
  inflating: word/settings.xml
  inflating: word/styles.xml
  inflating: word/webSettings.xml
  inflating: word/fontTable.xml
  inflating: docProps/core.xml
  inflating: docProps/app.xml
  inflating: Light_speed_corp_logo_pink_team.xml
  inflating: chatlog_6_may_2019.xml
```

The two files that look out of place are `Light_speed_corp_logo_pink_team.xml` and `chatlog_6_may_2019.xml`. Opening the former reveals the flag for this challenge:

Figure 7.1: The flag is revealed

Opening the chat log reveals the usernames of our two suspicious individuals, `Caomhainn` and `kondrat_ankudinov`.

7.2 Flag

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

8 [B-1] Fight the Binary Monster

An executable file, what could it be...

8.1 Solution

Disassembling Windows binaries gets quite tedious due to the Win32 functions; since the program asked for input, running strings on the executable gave us some good insights into the answer to its question:

```
https://pastebin.com/raw/EcrLPtRP
InternetOpenUrl failed
%.*s
InternetReadFile failed
https://pastebin.com/raw/v1cRRWEW
What domain is being accessed by this executable file?
pastebin.com
Correct! Now prove that you are a human to get my secret.
Go away Mr Robot.
```

Figure 8.1: Ah, pastebin my old friend

Dumping the output of the program when given the correct answer (`pastebin.com`) gives us a tree, and doing a post-order traversal gives us the flag.

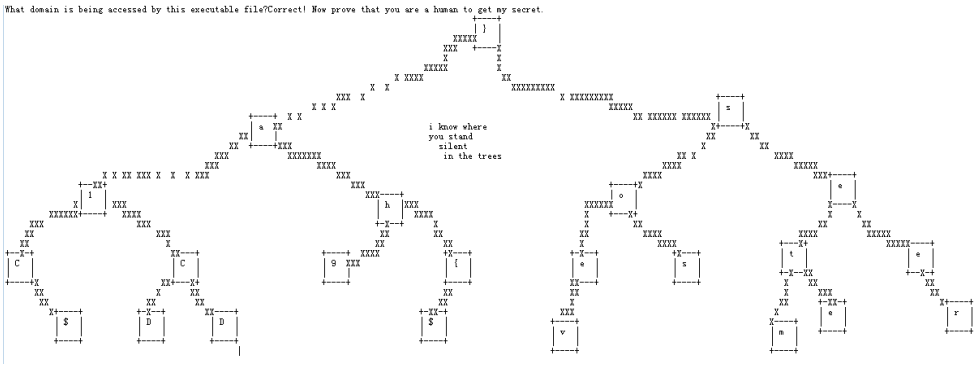

Figure 8.2: Silent in the trees?

8.2 Flag

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

9 [B-2] I <3000 PHISH

A macro-enabled Word document? Only slightly better than an executable...

9.1 Solution

Opening the document (without enabling the macros, of course) reveals some code (Listing 1.2). The key lines are reproduced below:

```
filePath = Environ("temp") + "\" + Chr(108) + "i" + Chr(103) + "h" + Chr(116) + "spe
        + Chr(46) + Chr(116) + "x" + Chr(116)
Open filePath + ":woohoo.txt" For Output As #1
```

Since : is not a valid character in filenames in Windows, we replace it with _ and allow the macro to run, which creates a file at `lightspeed.txt_woohoo.txt`:

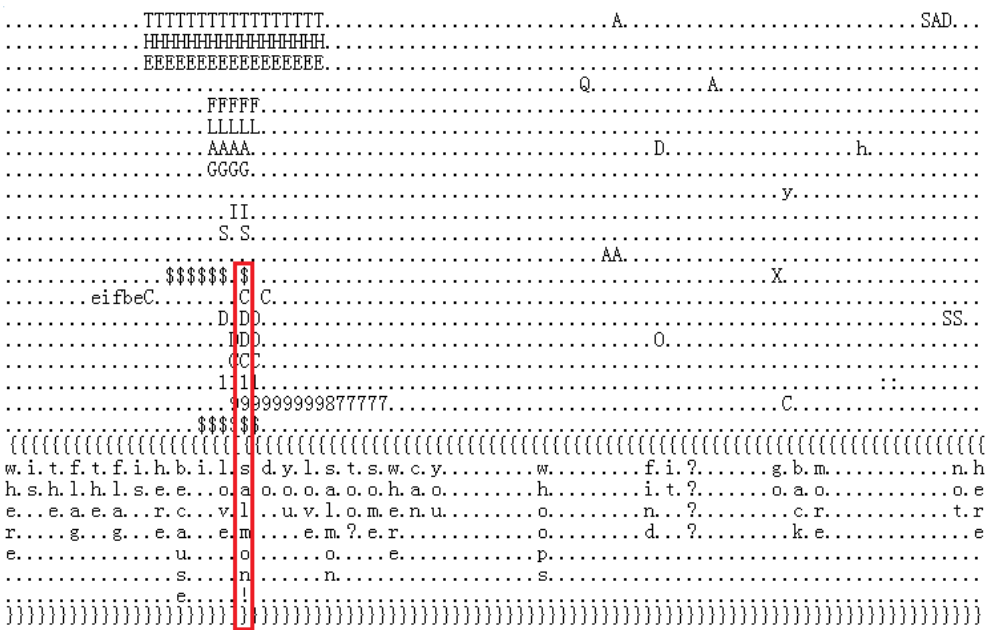

Figure 9.1: Vertical, sneaky

9.2 Flag

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

10 [B-3-1] Onion Sauce

The link given is clearly an onion site meant to be opened with the Tor browser.

10.1 Solution

Opening `ctfsg4bndpw6xurhitwa2dh66ycorghoa2ym3s3s4g3bgxqs3veaf4ad.onion` in Tor reveals an Ethereum address `0x7bd106a84773b43e2de9f68961b53cf8fb95a1f1`, and viewing the source code of the page reveals a long line of `<br>` tags.

When removed, the flag is revealed.

10.2 Flag

The flag for this challenge was `$CDDC19${n0W_Y0u_KnOw_Th3_S4uC3}`.

11 [B-3-2] When Your ZIL Turns to NIL

The ethereum address is found in the previous challenge, Onion Sauce.

11.1 Solution

Viewing the wallet on public blockchain explorers[6] reveals that there are 3 transactions associated with the account:

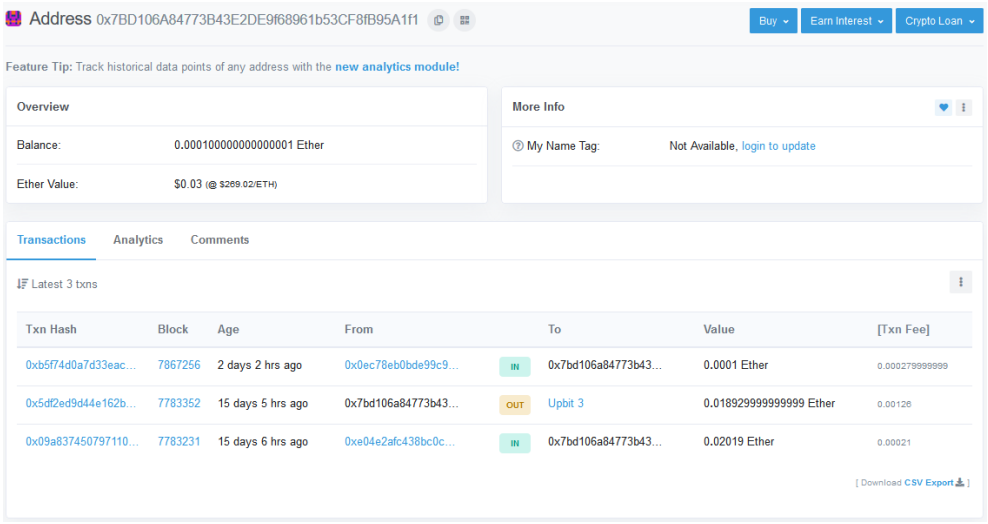

Figure 11.1: Small amounts...

Opening the first transaction gives us the address of the victim and the amount of ethereum:

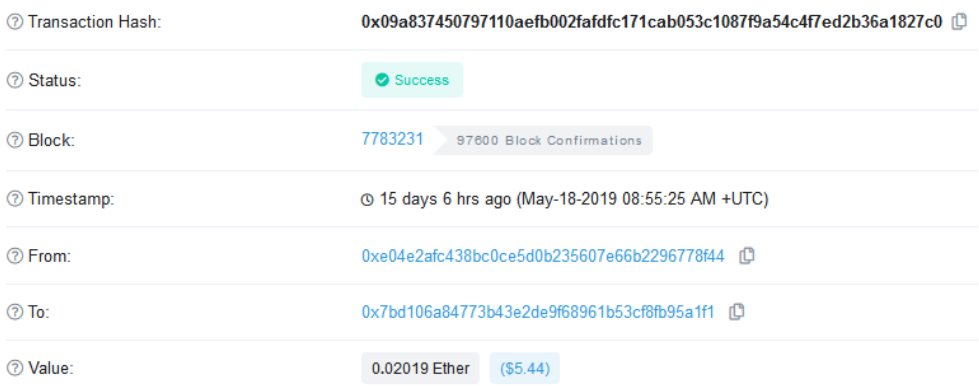

Figure 11.2: The amount of ether was 0.02019

[6] https://etherscan.io

11.2 Flag

The flag is `$CDDC19${0xE04E2AFC438BC0CE5D0B235607E66B2296778F44+0.02019}`.

12 [B-4-1] Where I Get All My GIFs From

The chat log from the first OSINT_BLUE challenge gives us the names of the people we need to search for in this and the next challenge.

12.1 Solution

We got lucky and tried Twitter first, and searching `Caomhainn` gave us the culprit immediately, along with the flag:

Figure 12.1: Never use your real name

12.2 Flag

The flag for this challenge was `$CDDC19${JEEVES_G0T_GIFS_1N_A_J1FFY}`.

13 [B-4-2] Hide N Seek

Following on from the previous challenge, 'The answer lies in this tweet'.

13.1 Solution

Following the trail of suspicious followers (made more difficult by CDDC participants following...), we arrive at this account:

Figure 13.1: Torvalds sounds familiar...

Again, the flag is clearly visible!

13.2 Flag

The flag for this challenge was `$CDDC19${00PS_U_F0UND_ME}`.

PART III

Programming

14 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;
}
```

14.1 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 14.1: The solution for Count 1: Baby

14.2 Flag

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

15 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.

15.1 Solution

Starting from the code for Count 1 (Listing 14.1), 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:

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

Listing 15.1: The solution for Count 2: Wildness

15.2 Flag

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

16 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.

16.1 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 16.1: The solution for Count 2: Wildness

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

Listing 16.2: An alternative solution for Count 2: Wildness

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

16.2 Flag

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

PART IV

Reverse Engineering

17 LSCVM: Immaculate Invasion

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

17.1 Solution

The tool of choice for this challenge was *Cutter*[7], an open-source reverse-engineering framework which performs disassembly, function analysis, and function/value renaming, among other things.

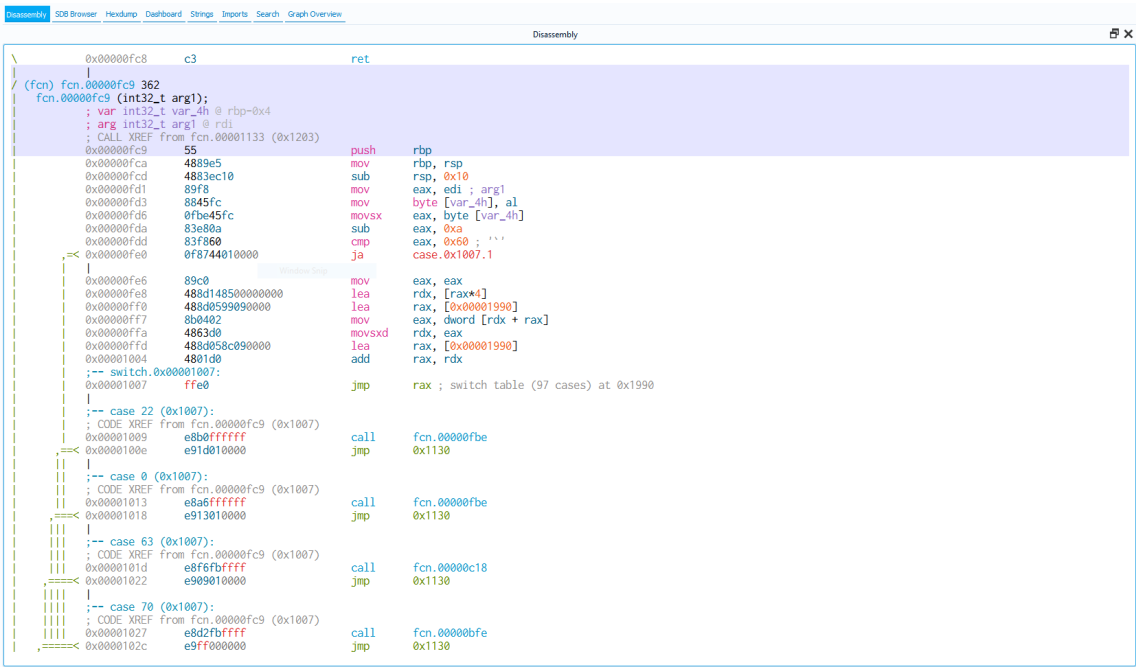

Figure 17.1: A screenshot of Cutter

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.

[7] 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 17.2: 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: `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 17.1. 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 17.3: `[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 17.4: 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!
```

17.2 Flag

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

17.3 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 V

Miscellaneous

18 Polyglot

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

18.1 Solution

The obvious solution was to put each sentence into Google Translate[8]; 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.

18.2 Flag

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

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

19 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.

19.1 Solution

Indeed, a quick trawl of Wikipedia led us to Suzhou numerals[9], 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).

19.2 Flag

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

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

20 Unzip

Oh, it isn't a zip bomb, is it?

20.1 Solution

Running `zip2john` on the file revealed a password protected file `flag.png`, and that the encryption format was `pkzip`.

```
$ zip2john Un.Zip
Created directory: /root/.john
ver 1.0 Un.Zip/flag.png PKZIP Encr: cmplen=579, decmplen=579, crc=C5C3E066
Un.Zip/flag.png:$pkzip...pkzip2$:flag.png:Un.Zip::Un.Zip
```

We could then use `pkcrack`[10] to crack the file:

```
$ ./extract -p Un.Zip flag.png
```

This reveals the flag:

Figure 20.1: Oh, numbers again...

20.2 Flag

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

[10] https://www.unix-ag.uni-kl.de/~conrad/krypto/pkcrack.html

21 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[11]. It can also be used to hide easter[12] eggs[13]...

21.1 Solution

The solution was rather straightforward. Using *PulseAudio* as a link between *VLC* and *QSSTV*[14] (an open-source Linux SSTV application), we were able to decode the SSTV image rather quickly:

Figure 21.1: 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:

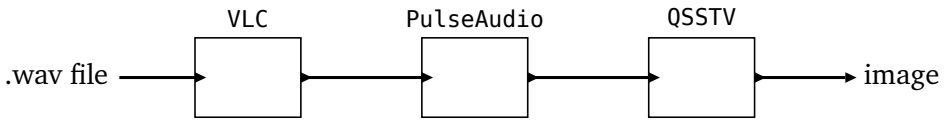

Figure 21.2: Data Pipeline Setup

[11] en.wikipedia.org/wiki/Slow-scan_television
[12] https://wiki.kerbalspaceprogram.com/wiki/List_of_easter_eggs
[13] https://half-life.fandom.com/wiki/Portal_ARG
[14] 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
```

Then by making QSSTV record from the `null sink`:

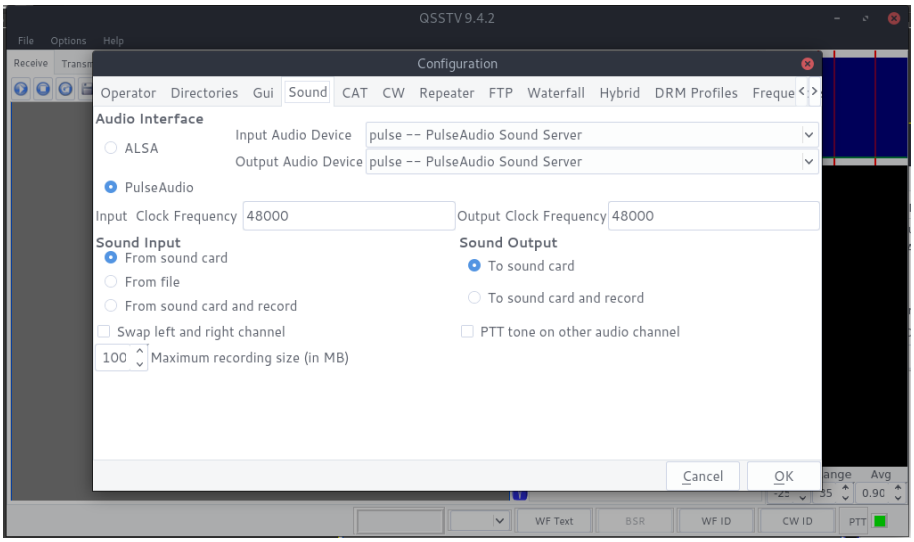

Figure 21.3: QSSTV – Set it to retrieve inputs from PulseAudio instead of the default *JACK*

And correspondingly in `pavucontrol`:

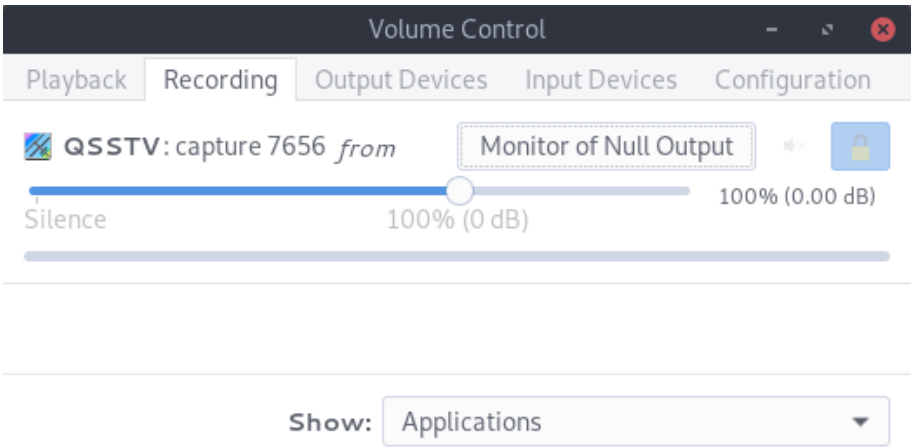

Figure 21.4: QSSTV – Record from the `null-sink`

Once everything is set up, we first start recording on QSSTV:

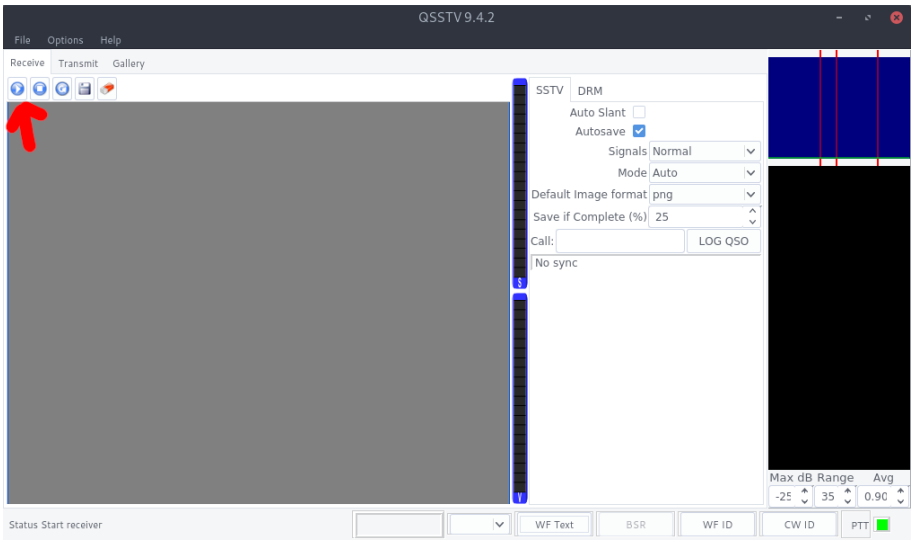

Figure 21.5: QSSTV – Record from the `null-sink`

Then use VLC to playback the `.wav` file to the `null-sink`:

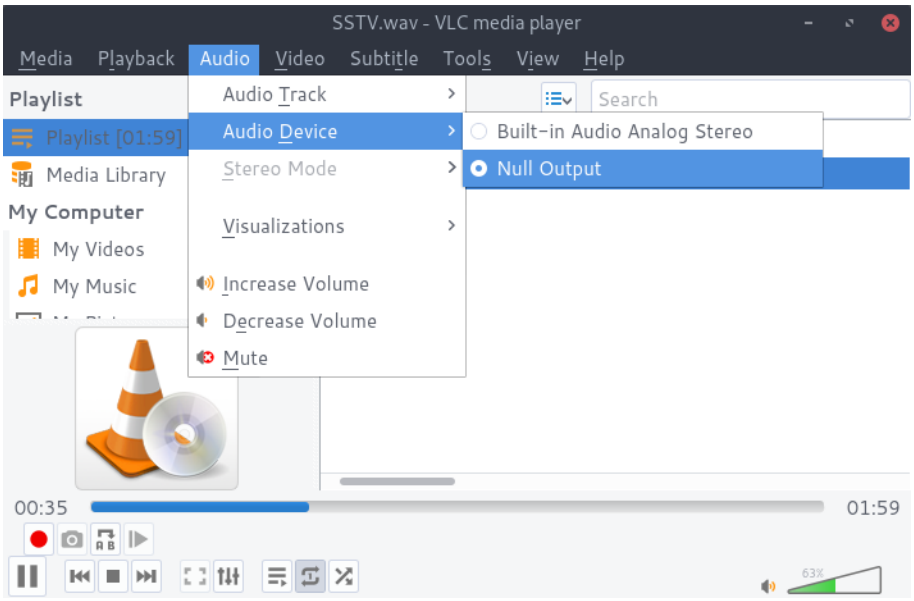

Figure 21.6: VLC – Play to `null-sink`

Then finally watch the magic unfold! Thankfully, most of the decoding is handled automatically by QSSTV and doesn't require much manual intervention.

21.2 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 21.7: Decoded SSTV Image

PART VI

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";
```

```
33
34                  int y = num % 10;
35                  ret += createNumber(y) + "A";
36
37                  return ret;
38          }
39  }
40
41
42  int main()
43  {
44          lookup['a'] = "jjMjAhA";
45          lookup['b'] = "jjMjAiA";
46          lookup['c'] = "jjMjAjA";
47          lookup['d'] = "cfMcfMM";
48          lookup['e'] = "cfMcfMMbA";
49          lookup['f'] = "jiAcdMM";
50          lookup['g'] = "jiAcdMMbA";
51          lookup['h'] = "jeAiM";
52          lookup['i'] = "hdfMM";
53          lookup['j'] = "hdfMMbA";
54          lookup['k'] = "hdfMMcA";
55          lookup['l'] = "ggdMM";
56          lookup['m'] = "ggdMMbA";
57          lookup['n'] = "fgAfMcM";
58          lookup['o'] = "fgAfMcMbA";
59          lookup['p'] = "fgAfMcMcA";
60          lookup['q'] = "fgAfMcMdA";
61          lookup['r'] = "fgAfMcMeA";
62          lookup['s'] = "fgAfMcMfA";
63          lookup['t'] = "fgAfMcMgA";
64          lookup['u'] = "fgAfMcMhA";
65          lookup['v'] = "fgAfMcMiA";
66          lookup['w'] = "fgAfMcMjA";
67          lookup['x'] = "gcfcMMM";
68          lookup['y'] = "fgAfgAM";
69          lookup['z'] = "fgAfgAMbA";
70          lookup['_'] = "gfdMMfA";
71          lookup['!'] = "fgAdM";
72          lookup['-'] = "fddMM";
73
74
75          std::string input;
76
77          while(true)
78          {
```

```
                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

1.2 phish-macro

```
Sub Document_Open()

Dim filePath As String, myURL As String, myPath As String
filePath = Environ("temp") + "\" + Chr(108) + "i" + Chr(103) + "h" + Chr(116) _
    + "speed" + Chr(46) + Chr(116) + "x" + Chr(116)
myURL = "https://pastebin.com/"
myPath = "raw/J6YCXPCM"

Dim WinHttpReq As Object
Set WinHttpReq = CreateObject("Microsoft.XMLHTTP")
WinHttpReq.Open "GET", myURL + myPath, False
WinHttpReq.Send

myURL = WinHttpReq.ResponseBody
If WinHttpReq.Status = 200 Then
    Set oStream = CreateObject("ADODB.Stream")
```

```
    oStream.Open
    oStream.Type = 1
    oStream.Write WinHttpReq.ResponseBody
    oStream.SaveToFile filePath, 2
    oStream.Close
End If

Dim text As String, textline As String, posLat As Integer, posLong As Integer
Open filePath For Input As #1
Do Until EOF(1)
    Line Input #1, textline
    text = text & textline & Chr$(13) & Chr$(10)
Loop
Close #1

Open filePath + ":woohoo.txt" For Output As #1
Print #1, text + "w.i.t.f.t.f.i.h.b.i.l.s.d.y.l.s.t.s.w.c. _
    y.........w.........f.i.?.......g.b.m............n.h"
Print #1, "h.s.h.l.h.l.s.e.e...o.a.o.o.o.a.o.o.h.a.o...... _
    ...h.........i.t.?.......o.a.o............o.e"
Print #1, "e...e.a.e.a...r.c...v.l...u.v.l.o.m.e.n.u...... _
    ...o.........n...?.........c.r............t.r"
Print #1, "r.....g...g...e.a...e.m.....e.m.?.e.r.......... _
    ...o.........d...?.........k.e..............e"
Print #1, "e...............u.....o.......o.....e.......... _
    ...p.........................................."
Print #1, "................s.....n.......n................ _
    ...s.........................................."
Print #1, "................e.....!........................ _
    ............................................."
Print #1, "}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}} _
    }}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}"
Close #1

End Sub
```

Listing 1.2: job-requirements.docm