

FLARE On 5: Challenge 3 Solution – FLEGGO

Challenge Author: Moritz Raabe (@m_r_tz)

Introduction

FLEGGO.zip is a ZIP archive containing 48 32-bit PE binaries. This challenge requires reverse engineers to identify how the binaries are related, how they work, and what information needs to be extracted from them to obtain the final flag. Malware analysts encounter this scenario when dealing with many similar malware samples. After figuring out a malware family's functioning, writing a program to automatically extract its interesting data saves a lot of manual analysis work. This LEGO themed challenge demonstrates that automation of repetitive tasks is an important skill for reverse engineers.

This write-up describes a mixture of static and dynamic analysis methods. We will go beyond the basics that are necessary to solve the challenge. Hopefully, this provides some helpful tips for other automation tasks.

This write-up relies mainly on the following tools:

- IDA Pro, <https://www.hex-rays.com/products/ida/>
- FLARE VM, <https://github.com/fireeye/flare-vm>
- Python, <https://www.python.org>

Basic Analysis

Basic Static Analysis

The ZIP archive contains 48 executable files with seemingly random names. Each binary has a file size of 45,056 bytes. We analyze the basic static features of one of the binaries, namely 1BpnGjH0T7h5vvZsV4vISSb60Xj3pX5G.exe. The executable's imports indicate that the file may access its resource section and create files. The file's resource section contains a resource of type BRICK. The resource data has a size of 0x8150 bytes. It starts with a wide character string, followed by likely encoded data and many zero bytes.

We use FLOSS [1] to extract strings from the binary. Figure 1 shows the interesting strings that provide an idea of the binary’s functionality. FLOSS does not identify any obfuscated strings for this sample.

```
BRICK
%s\s
IronManSucks
Oh, hello Batman...
I super hate you right now.
What is the password?
%15ls
Go step on a brick!
Oh look a rainbow.
Everything is awesome!
%s => %s
BRICK
ZImIT7DyCM0eF6
```

Figure 1: Interesting strings from 1BpnGjHOT7h5vvZsV4vISSb60Xj3pX5G.exe

Basic Dynamic Analysis

To get a basic understanding of the program we run 1BpnGjHOT7h5vvZsV4vISSb60Xj3pX5G.exe. Figure 2 shows that the application prompts for a password. After entering an arbitrary string, the program terminates with the message “Go step on a brick!”. From the extracted strings we deduce that there is a correct password that results in a different behavior.



Figure 2: Running a binary from FLEGG0.zip

File Comparison

We observe identical static and dynamic features between multiple binaries from the archive. A binary diffing tool such as VBinDiff [2] identifies the differences between files. Figure 3 shows the differences between two binaries. The differences start at file offset 0x2AB0 (10,928). After several bytes the files are identical again. All differences lie in the files’ resource sections.

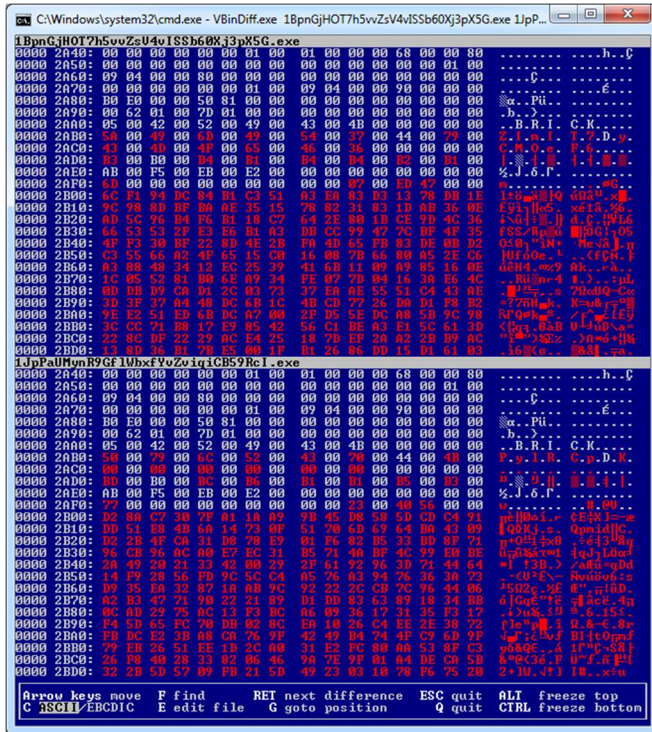


Figure 3: Binary difference between two binaries

We suspect that the files only differ in their resource data. We confirm this using the Linux `cmp` command and the Bash script shown in Figure 4. The script shows the first byte-wise difference between the file `1BpnGjHOT7h5vvZsV4vISSb60Xj3pX5G.exe` and the other binaries. For almost all files the first difference is encountered within the resource section at byte 10,929. We can use similar commands to confirm that all files are identical after the resource data.

```
$ for f in $(ls .); do cmp 1BpnGjHOT7h5vvZsV4vISSb60Xj3pX5G.exe $f; done
1BpnGjHOT7h5vvZsV4vISSb60Xj3pX5G.exe 1JpPauMynR9Gf1WbxfYvZvqiCB59RcI.exe differ: byte 10929, line 23
1BpnGjHOT7h5vvZsV4vISSb60Xj3pX5G.exe 2AljFfLleprkThThuVvg63I70gjG2LQT.exe differ: byte 10929, line 23
1BpnGjHOT7h5vvZsV4vISSb60Xj3pX5G.exe 3Jh0ELkck1MuRvzr8PLIpBNUGlspmGnu.exe differ: byte 10929, line 23
1BpnGjHOT7h5vvZsV4vISSb60Xj3pX5G.exe 4ihY3RWk4WYqI4XOXLtAH6XV51koIdgv.exe differ: byte 10929, line 23
<snip>
```

Figure 4: Bash script to identify the first binary difference between files

Although we can solely rely on byte-wise comparisons here, real-world malware may require more advanced techniques. This includes function comparisons on assembly level using tools like BinDiff [3] or Diaphora [4].

Advanced Analysis

We disassemble the executable file 1BpnGjHOT7h5vvZsV4vISSb60Xj3pX5G.exe using IDA Pro. The tool automatically recognizes the main function where we recognize the password prompt, the failure output “Go step on a brick!” and the suspected success case with output "Everything is awesome!". Based on our knowledge from executing the binary, we identify the function at offset 0x401510 as wprintf and the function at offset 0x4014C0 as wscanf. Figure 5 shows the disassembled main function in IDA Pro.

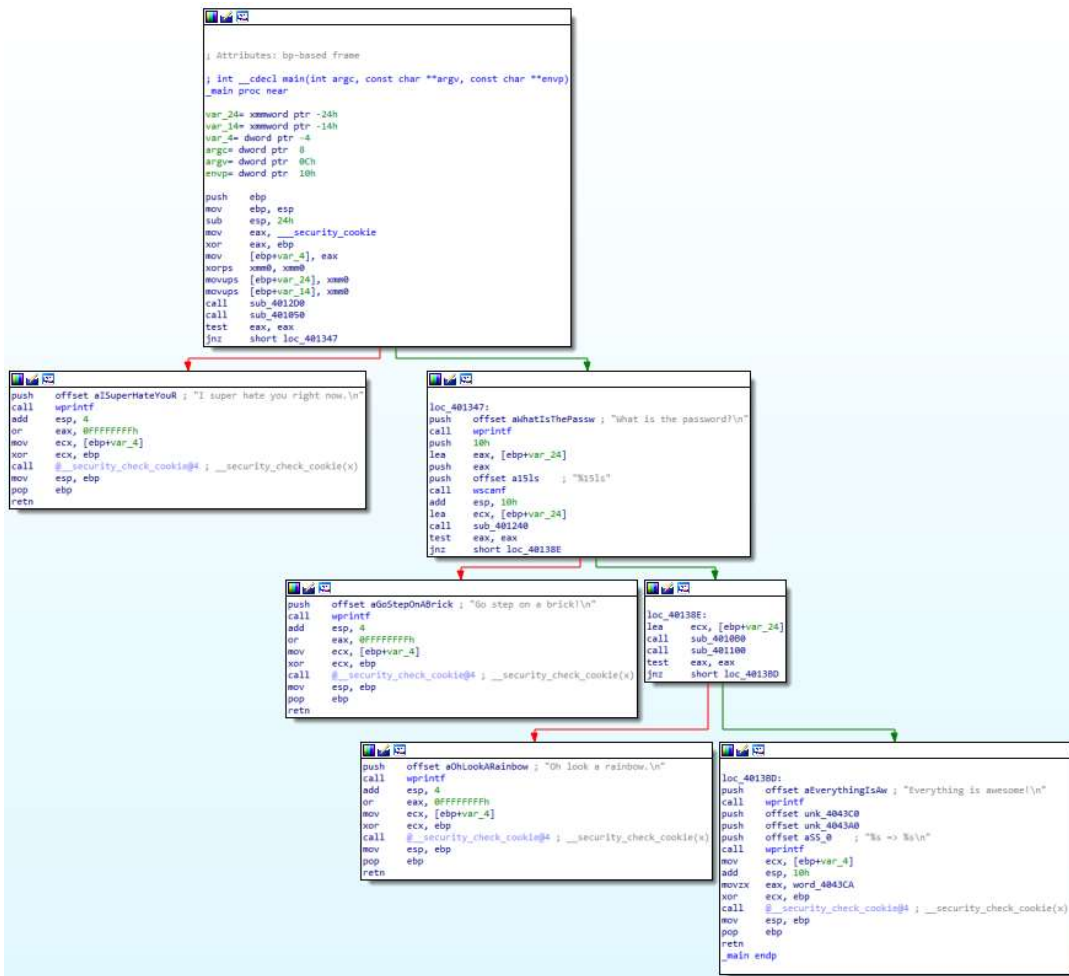


Figure 5: Disassembled main function of 1BpnGjHOT7h5vvZsV4vISSb60Xj3pX5G.exe

The function at offset 0x401050 copies 0x8150 bytes (the size of the resource data) from the BRICK resource into global memory. We define an array named ResourceData at the respective address in the file's .data section (see Figure 6). The cross-references to the copied data then show us where the program uses it.

```
.data:00404380 ?? ?? ?? ?? ?? ??+ResourceData    db 8150h dup(?)           ; DATA XREF: sub_401050+10↑o
.data:00404380 ?? ?? ?? ?? ?? ?? ??+           ; sub_401240:loc_401288↑o ...
```

Figure 6: Defined array that stores the resource data

If IDA Pro only displays cross-references to the beginning of the data, increase the “Cross reference depth” via Options – General... – Cross-references (see Figure 7). By default, the value is 16 bytes. Alternatively, we can use the text search feature to see where the program accesses the data (ALT+T or Search – text... – search for ResourceData, select Find all occurrences).

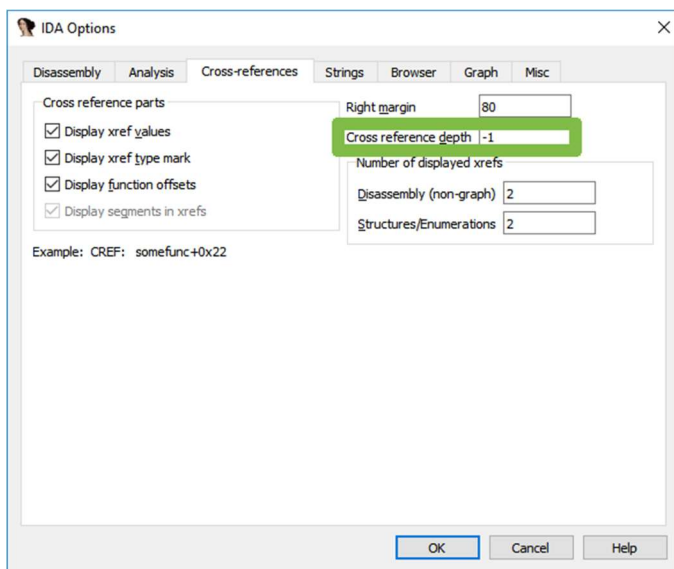


Figure 7: Modifying IDA Pro's cross reference depth

The program references ResourceData in the function at offset 0x4012D0. This is the first function called by main. Here the program initializes parts of the ResourceData to zero. The data appears to contain multiple fields that we are going to analyze in more detail later.

After the call to `wscanf` at offset `0x40135C`, the program passes the user input as a wide character string to the function at offset `0x401240`. Based on the function's return value, the program either prints the friendly recommendation to step on a brick and terminates (return value 0) or continues its execution (all other return values).

The function at offset `0x401240` performs an inlined wide character string comparison between the string `IronManSucks` and the user input. If the strings are equal, the program prints "Oh, hello Batman..." and returns 0 (error case). Otherwise, the program compares the user input to the beginning of the copied resource data. If both strings are equal, the function returns 1 and program execution continues. For this binary the resource data starts with the string `ZImIT7DyCM0eF6`.

Figure 8 shows the program's output when provided the correct password. In addition, the application creates an image file in the executable's current directory. Figure 9 displays the created PNG image. Performing the same steps for other binaries provides identical results. Each binary creates a different image file and provides output that associates the file name to a letter.

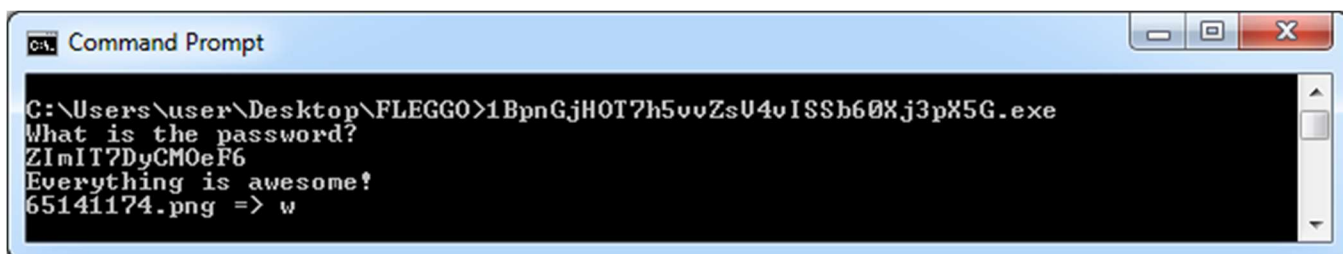


Figure 8: Executing the binary with the correct password

7

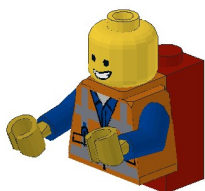
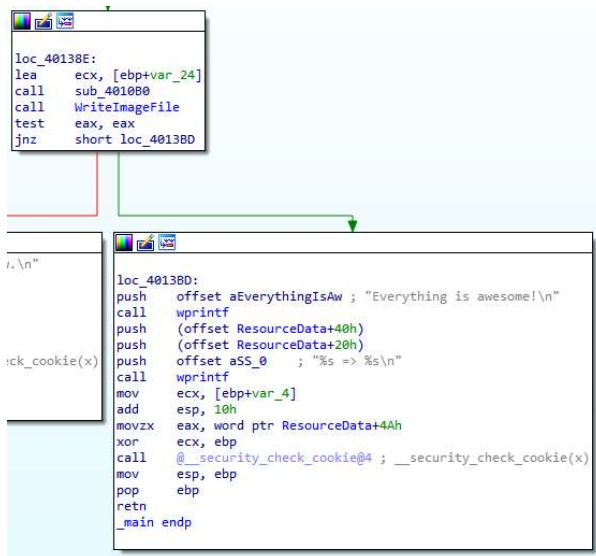


Figure 9: Created PNG image file `65141174.png`

The program implements the file creation in the function at offset 0x401100. The disassembly in Figure 10 shows that the program prints two strings after the image file has been created. The association between filename and letter uses fields from the resource data. Lastly, the program returns a WORD from ResourceData offset 0x4A.



```

loc_40138E:
lea    ecx, [ebp+var_24]
call   sub_4010B0
call   WriteImageFile
test   eax, eax
jnz    short loc_40138D

loc_4013BD:
push   offset aEverythingIsAw ; "Everything is awesome!\n"
call   wprintf
push   (offset ResourceData+40h)
push   (offset ResourceData+20h)
push   offset aSS_0 ; "%s => %s\n"
call   wprintf
mov    ecx, [ebp+var_4]
add    esp, 10h
movzx  eax, word ptr ResourceData+4Ah
xor    ecx, ebp
call   @_security_check_cookie@4 ; __security_check_cookie(x)
mov    esp, ebp
pop    ebp
retn
_main endp
  
```

Figure 10: References to ResourceData offsets at the end of the program's main function

Challenge Solutions

Executing a binary with the correct password provides us an image file and an association between the file name and a letter. Each image includes a number that indicates the letter's position in the challenge's flag. Now we could solve the remaining binaries and recover the solution by hand. We could also write a script that runs each program with the correct password and processes their outputs. Both solutions then require us to manually associate the letters and their respective offsets.

The following sections describe an in-depth program analysis including details on how to recover the resource data structure, and the usage of a script to automatically extract the final flag.

In-depth Analysis

We analyze the few functions that have not been already identified. The function at offset 0x4010B0 references multiple ResourceData offsets and calls two functions. The first function (0x401080) contains a small loop with a non-zeroing XOR instruction at offset 0x401097. This function performs a NULL-preserving byte-wise XOR using three arguments that are passed to it using Microsoft's `__fastcall` calling convention: a pointer to a buffer (ecx), a length (edx), and the key byte (on the stack). This function is used to XOR decode 32 bytes at resource data offset 0x20 using key byte 0x85 and 10 bytes at resource data offset 0x40 using key byte 0x1A.

The second function (0x401660) takes four arguments: the user input (the correct password); a pointer to ResourceData offset 0x50; a DWORD from ResourceData offset 0x4C; and, again, a pointer to ResourceData offset 0x50. The function makes two interesting calls. The function at offset 0x401540 contains several small loops and uses the constant 0x100 while it modifies a buffer. The function at offset 0x4015D0 uses the generated buffer and performs various data moves and a non-zeroing XOR instruction. These are good indicators for the RC4 algorithm. The first function performs the key-scheduling algorithm (KSA) followed by the pseudo-random generation algorithm (PRGA). Hence, the function at offset 0x401660 RC4 encrypts/decrypts data with a provided password.

Table 1 shows the identified fields from the resource data given our current analysis results.

Offset	Description	Length in bytes
0x0	Password	0x20 (32)
0x20	Unknown (XOR encoded)	0x20 (32)
0x40	Unknown (XOR encoded)	0xA (10)
0x4C	Length RC4 encrypted data	4
0x50	RC4 encrypted data	Variable

Table 1: Identified ResourceData fields

The cross-references to ResourceData help us to recover the remaining data elements. In the function at offset 0x401100 we see that ResourceData offset 0x20 is used to create the file path of the dropped file, offset 0x50 is the start of the data buffer, and offset 0x4C stores the buffer length.

In the main function, ResourceData offsets 0x20 and 0x40 are used to format the output “%s => %s”. Offset 0x20 is the filename and offset 0x40 is the letter (part of the final flag). ResourceData offset 0x4A stores a WORD that the program returns when terminating successfully. Figure 11 shows the recovered ResourceData structure.

```

struct _ResourceData {
    WCHAR    Password[0x10];
    WCHAR    EncodedFilename[0x10]; // XOR encoded, key 0x85
    WCHAR    EncodedLetter[0x5];    // XOR encoded, key 0x1A
    WORD     ReturnValue;
    DWORD    DataSize;
    BYTE     Data[DataSize];        // RC4 encrypted using Password
};
    
```

Figure 11: ResourceData structure

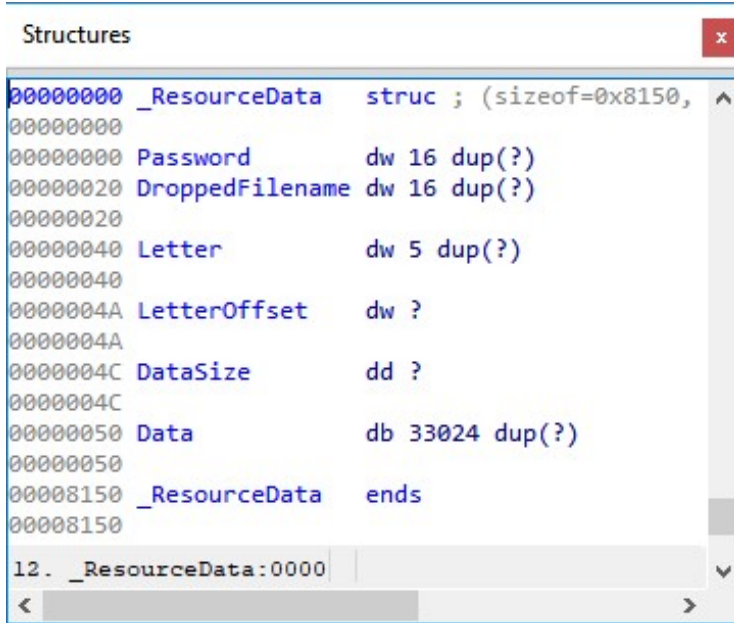
Table 2 shows the recovered values for the file 1BpnGjHOT7h5vvZsV4vISSb60Xj3pX5G.exe.

Offset	Description	(Decoded) value for 1BpnGjHOT7h5vvZsV4vISSb60Xj3pX5G.exe
0x0	Password	ZlmlT7DyCMOeF6
0x20	Filename (XOR encoded)	65141174.png
0x40	Letter (XOR encoded)	w
0x4A	Return value	7
0x4C	Length RC4 encrypted data	0x47ED
0x50	RC4 encrypted data	<PNG image data>

Table 2: Recovered resource data values for 1BpnGjHOT7h5vvZsV4vISSb60Xj3pX5G.exe

Extracting and decoding the resource data for multiple binaries shows that the return value fields and the numbers in the created images are equal. This allows us to order the final flag letters automatically using the return value as an offset without manually inspecting the image files.

Figure 12 shows the final structure definition in IDA Pro.



```

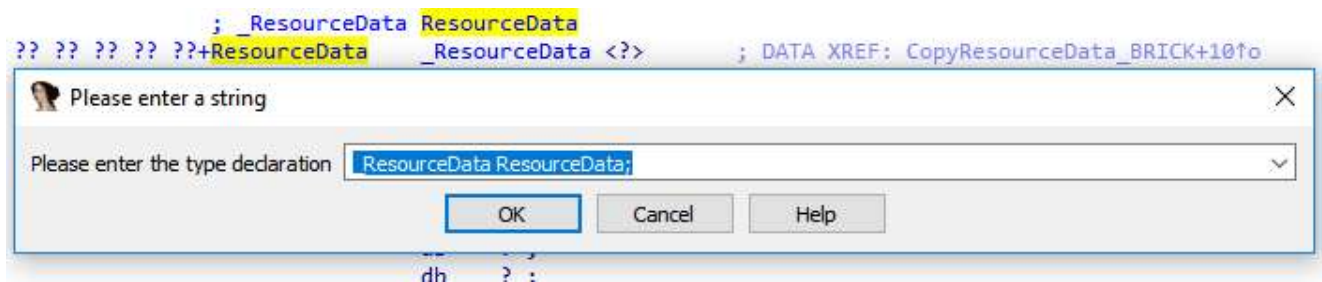
Structures
00000000 _ResourceData  struc ; (sizeof=0x8150,
00000000
00000000 Password      dw 16 dup(?)
00000020 DroppedFilename dw 16 dup(?)
00000020
00000040 Letter        dw 5 dup(?)
00000040
0000004A LetterOffset  dw ?
0000004A
0000004C DataSize      dd ?
0000004C
00000050 Data          db 33024 dup(?)
00000050
00008150 _ResourceData  ends
00008150

12. _ResourceData:0000

```

Figure 12: Final `_ResourceData` structure definition

We apply the defined structure in IDA Pro to the `ResourceData` offset by modifying its type via the Y key (see Figure 13). Alternatively select the address and go to Edit – Struct var... (ALT-Q).



```

; _ResourceData ResourceData
?? ?? ?? ?? ??+ResourceData  _ResourceData <?> ; DATA XREF: CopyResourceData_BRICK+10fo

```

Please enter a string

Please enter the type declaration: `ResourceData ResourceData;`

OK Cancel Help

Figure 13: Applying the created structure to the global `ResourceData` offset

Figure 14 shows the annotated main function in IDA Pro including the applied structure definition.

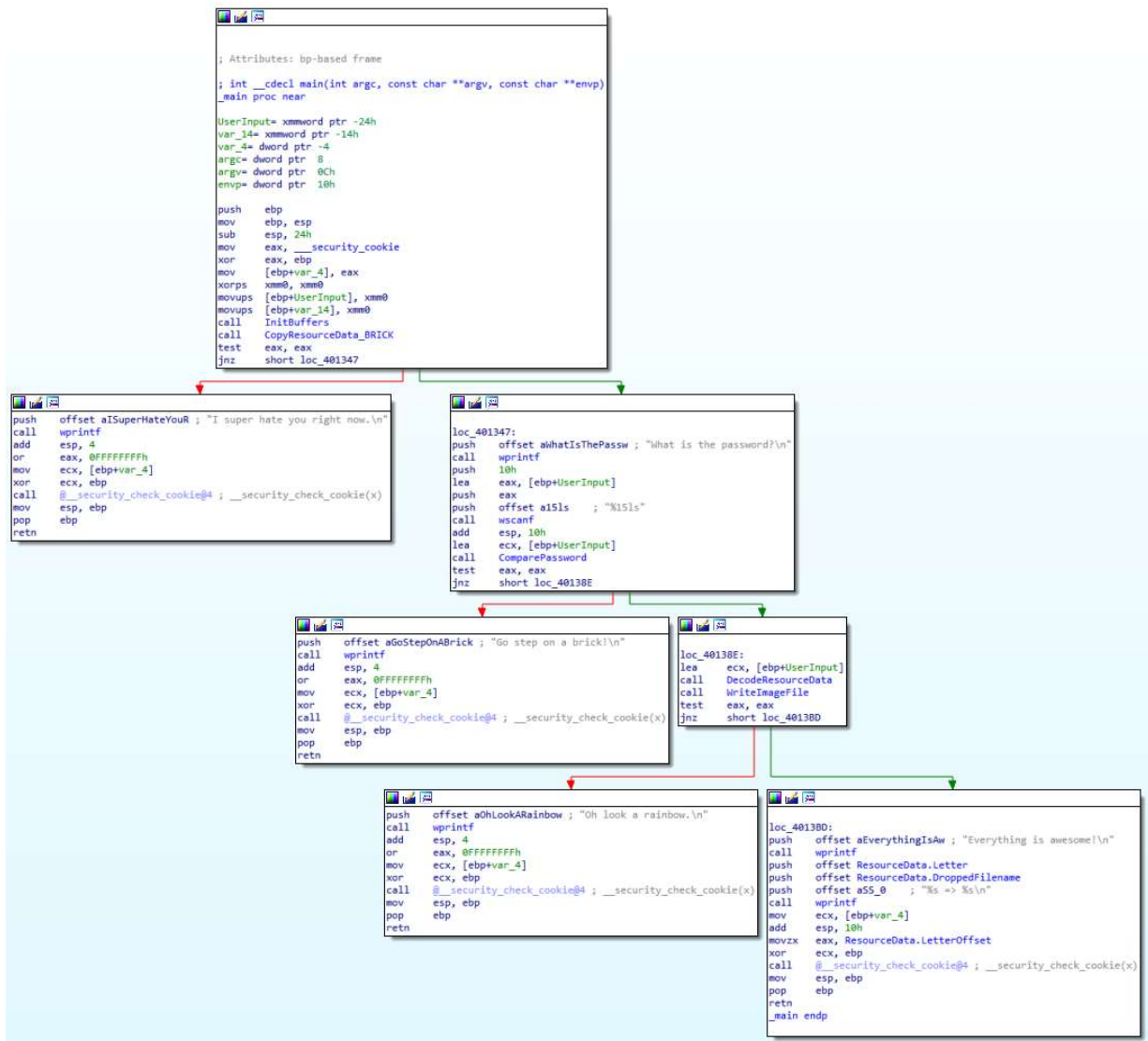


Figure 14: Annotated disassembly of the application's main function

Python Solution Script

Figure 15 lists a Python script to automatically extract the resource data and recover the final flag from all 48 binaries. The script uses the vstruct Python module to parse the binary resource data. vstruct is part of the vivisect project [5]. The article at [6] explains the vstruct module's installation and usage. The Python script RC4 decrypts the image data which is not necessary to solve the challenge.

```
"""
Solution script for FLARE On 5, 2018
Challenge 3, FLEGG0

Author: Moritz Raabe, moritz.raabe@fireeye.com, @m_r_tz

Example run:
python extract_bricks.py /path/to/dir/with/unzipped/FLEGG0.zip
The final flag is: mor3_aws0m3_th4n_an_aws0me_p0ssum@flare-on.com
"""

import os
import sys
import logging

import vstruct
from vstruct.primitives import v_wstr, v_bytes, v_int16, v_int32

from Crypto.Cipher import ARC4

RES_DATA_OFFSET = 0x2AB0
RES_DATA_SIZE = 0x8150
OUTPUT_DIR = "decrypted_images"

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class BrickConfig(vstruct.VStruct):
    def __init__(self):
        vstruct.VStruct.__init__(self)
        self.password = v_wstr(0x10) # offset 0x0
        self.filename = v_bytes(0x20) # 0x20
        self.letter = v_bytes(0xA) # 0x40
        self.letter_offset = v_int16() # 0x4A
        self.data_size = v_int32() # 0x4C
        self.data = v_bytes(size=0) # 0x50

    def pcb_filename(self):
        """ XOR decode filename """
        self.decoded_filename = bytearray(self.xor_decode(self.filename, 0x85)).decode("utf-16").replace("\x00", "")

    def pcb_letter(self):
        """ XOR decode letter """
        self.decoded_letter = bytearray(self.xor_decode(self.letter, 0x1A)).decode("utf-16").replace("\x00", "")

    @staticmethod
    def xor_decode(data, key):
        out = list(data)
```

```
    for i, c in enumerate(data):
        if c != "\0":
            out[i] = ord(c) ^ key
    return out

def pcb_data_size(self):
    """ Set data length after parsing it """
    self["data"].vsSetLength(self.data_size)

def pcb_data(self):
    """ RC4 decrypt data """
    rc4_cipher = ARC4.new(self.password)
    self.image_data = rc4_cipher.decrypt(self.data)

def __str__(self):
    return "password: %s\nfilename: %s\nletter: %s\nletter offset: %d\ndata size: %d" % (
        self.password, self.decoded_filename, self.decoded_letter, self.letter_offset,
        self.data_size)

def main():
    if len(sys.argv) != 2:
        print("Usage: %s <path to executable(s)>" % sys.argv[0])
        return -1

    input_path = sys.argv[1]

    if not os.path.exists(input_path):
        logger.error("%s does not exist", input_path)
        sys.exit(-1)

    if not os.path.isdir(input_path):
        exe_file_paths = [input_path]
    else:
        exe_file_paths = []
        for f in os.listdir(input_path):
            if f.endswith(".exe"):
                exe_file_paths.append(os.path.join(input_path, f))

    letters = process_exe_files(exe_file_paths)

    print_final_flag(letters)

def process_exe_files(exe_file_paths):
    """ Return list of tuples with letter offsets and letters """
    letters = []
    if not os.path.exists(OUTPUT_DIR):
        os.mkdir(OUTPUT_DIR)
    for fp in exe_file_paths:
        letters.append(extract_resource_data(fp))
    return letters
```

```
def extract_resource_data(file_path):
    """ Create image file and return tuple with letter offset and letter """
    resource_data = get_raw_resource_data(file_path)
    filename, letter, letter_offset, data_size, image_data = parse_resource(resource_data)
    t_file = os.path.join(OUTPUT_DIR, "%02d_%s.png" % (letter_offset, letter.encode("ascii")))
    with open(t_file, "wb") as f:
        logger.debug("Creating %s", t_file)
        f.write(image_data)
    return (letter_offset, letter)

def get_raw_resource_data(file_path):
    """ Return resource data using hard-coded offset and size, could also use pefile module """
    with open(file_path, "rb") as f:
        d = f.read()
    return d[RES_DATA_OFFSET:RES_DATA_OFFSET + RES_DATA_SIZE]

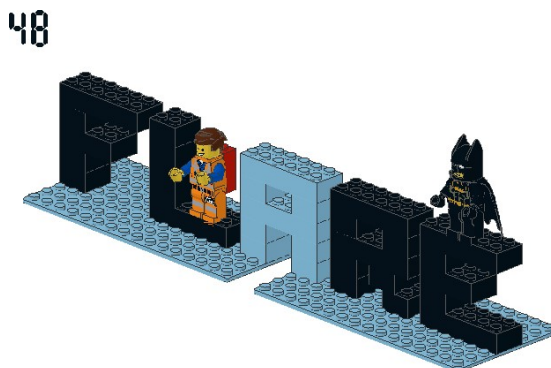
def parse_resource(resource_data):
    """ Return decoded resource data parsed via vstruct """
    brick_config = BrickConfig()
    brick_config.vsParse(resource_data)
    logger.debug("%s", brick_config)
    return brick_config.decoded_filename, brick_config.decoded_letter, brick_config.letter_offset, \
        brick_config.data_size, brick_config.image_data

def print_final_flag(letters):
    flag = []
    for d in sorted(letters):
        flag.append(d[1])
    if len(flag) > 1:
        print "The final flag is: %s" % "".join(flag)

if __name__ == "__main__":
    main()
```

Figure 15: Python solution script

Credits



A big shout-out to the LEGO community – especially BrickLink [7]. The instruction images were created using MLCad 3.5 [8].

Links and Resources

[1] FLOSS, <https://github.com/fireeye/flare-floss/>

[2] VBinDiff, <https://www.cjmweb.net/vbindiff/>

[3] BinDiff, <https://www.zynamics.com/bindiff.html>

[4] Diaphora, <https://github.com/joxeankoret/diaphora>

[5] vivisect, <https://github.com/vivisect/vivisect/>

[6] Parsing Binary Data with `vstruct`, William Ballenthin,
<http://www.williballenthin.com/blog/2015/09/08/parsing-binary-data-with-vstruct/>

[7] BrickLink, <https://www.bricklink.com>

[8] MLCad, <https://www.lm-software.com/mlcad/>