Using Emulation Against Anti-Reverse Engineering Techniques

Affected Platforms: Windows

Introduction

The life of a malware reverse-engineer is not always easy. You might think that it is all fun and games, but in reality, it is bits and assembly. And as it is true for the whole security industry, it is a cat and mouse game with malware developers. As we continuously expose their malware, which leads us to learn more about the attackers and develop countermeasures, they keep trying to implement new ways of slowing down analysts and reverse engineering. In a recent blog, we analyzed the Pandora ransomware and discussed that it is filled with anti-reverse-engineering techniques, and has multiple layers of code obfuscation.

In this blog post, instead of talking about what we discovered in the Pandora ransomware, I would like to discuss how we did that. More specifically, we will fight the function call obfuscation and string encryption in this sample, using emulation. We will implement an IDAPython script using the flare-emu framework to turn the disassembly in IDA Pro more readable. This will be a great help in the static analysis of the sample.

Challenges in Pandora

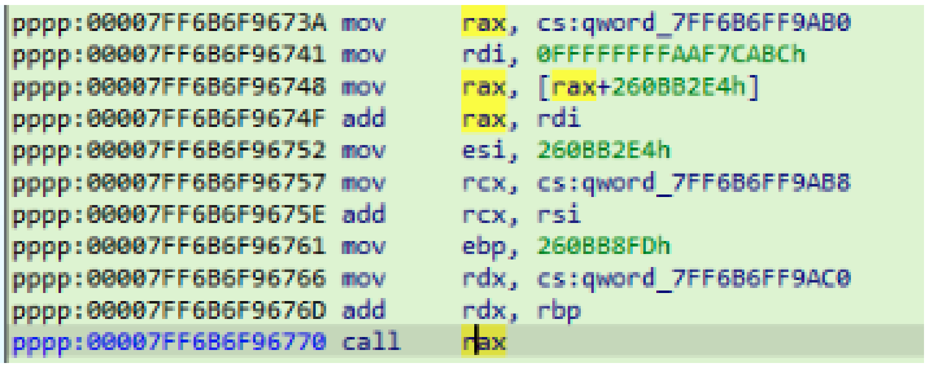
For the detailed list of challenges we faced while reversing Pandora, read our analysis blog. In this post I will discuss two specific anti-reverse-engineering techniques that we've seen in Pandora:

- Function call obfuscation with opaque predicates
- Encrypted strings

First, let's discuss what these challenges are exactly.

Function Call Obfuscation with Opaque Predicates

Figure 1 shows what a simple function call looks like in the Pandora ransomware after it was unpacked.



```
pppp:00007FF6B6F9673A mov     rax, cs:qword_7FF6B6FF9AB0
pppp:00007FF6B6F96741 mov     rdi, 0FFFFFFFFAAF7CABCh
pppp:00007FF6B6F96748 mov     rax, [rax+260BB2E4h]
pppp:00007FF6B6F9674F add     rax, rdi
pppp:00007FF6B6F96752 mov     esi, 260BB2E4h
pppp:00007FF6B6F96757 mov     rcx, cs:qword_7FF6B6FF9AB8
pppp:00007FF6B6F9675E add     rcx, rsi
pppp:00007FF6B6F96761 mov     ebp, 260BB8FDh
pppp:00007FF6B6F96766 mov     rdx, cs:qword_7FF6B6FF9AC0
pppp:00007FF6B6F9676D add     rdx, rbp
pppp:00007FF6B6F96770 call    rax
```

Figure 1 - Standard function call in Pandora

We can see that the address of the function that is being called is calculated in runtime. cs:qword_7FF6B6FF9AB8 seems to be the base address of some kind of function address table. Then we use hardcoded values to find the right function pointer in that table and that is what we load into rax before calling it. An opaque predicate generally means an expression in the program whose outcome is known to the programmer, but it still needs to be evaluated in runtime. It is used in many different ways as an obfuscation and anti-analysis technique. In this case, the value that goes in rax is fixed, but because it still has to be calculated in runtime it disrupts static-analysis tools.

If we use Figure 1 as an example, the address in rax is calculated like this:

rax = *(*address_table_base + 0x260BB2E4) + 0xFFFFFFFFAAF7CABC)

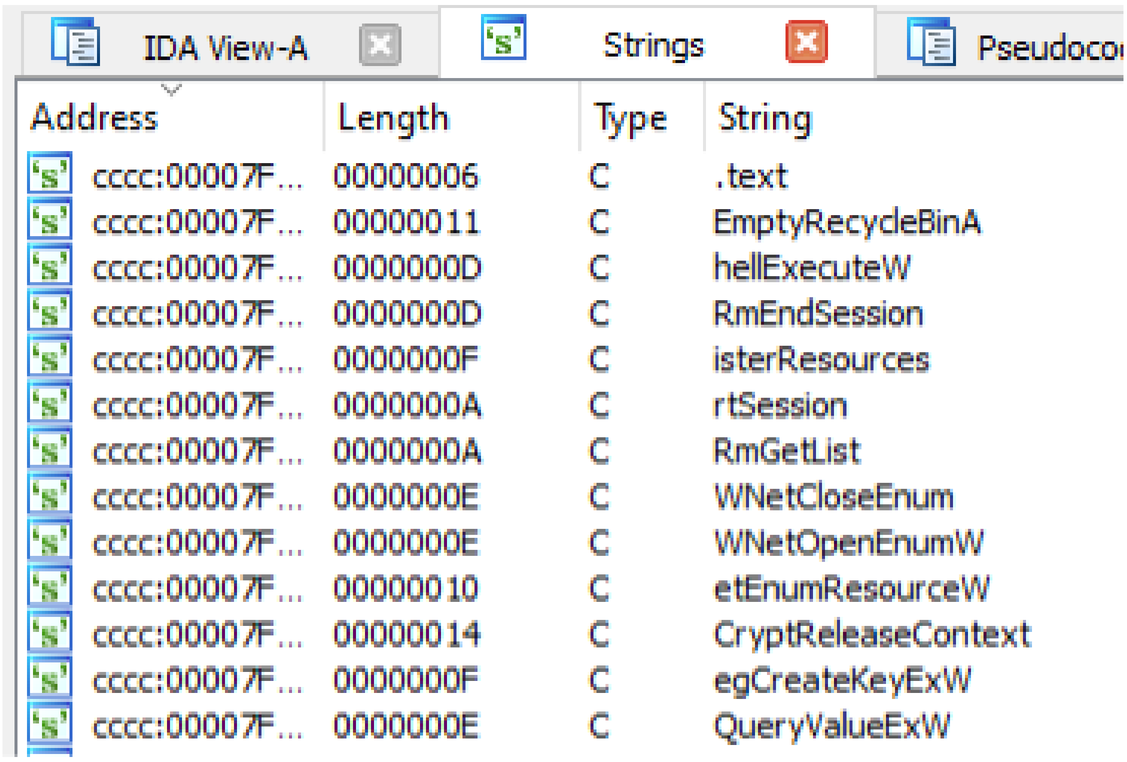
or in decimal:

rax = *(*address_table_base + 638300900) - 1426601284)

The trivial solution for such a problem is to run the malware in a debugger and get the address from there. But in this sample all function calls were like this (except those in the statically linked libraries). That means that we would have needed to break at every function call in the debugger just to have the slightest idea of what is happening in the malware. This called for automation.

Encrypted Strings

Another challenge of this particular ransomware sample was that all the interesting strings were encrypted. There were a lot of plain text strings in the binary (shown in Figure 2), but they were mostly Windows API function names and the strings in the embedded libraries. None of the strings that would help us understand what the malware is doing are available in plain text. This is very common in modern malware, thus the solution to this challenge could be used against a wide variety of malware.



Address	Length	Type	String
cccc:00007F...	00000006	C	.text
cccc:00007F...	00000011	C	EmptyRecycleBinA
cccc:00007F...	0000000D	C	hellExecuteW
cccc:00007F...	0000000D	C	RmEndSession
cccc:00007F...	0000000F	C	isterResources
cccc:00007F...	0000000A	C	rtSession
cccc:00007F...	0000000A	C	RmGetList
cccc:00007F...	0000000E	C	WNetCloseEnum
cccc:00007F...	0000000E	C	WNetOpenEnumW
cccc:00007F...	00000010	C	etEnumResourceW
cccc:00007F...	00000014	C	CryptReleaseContext
cccc:00007F...	0000000F	C	egCreateKeyExW
cccc:00007F...	0000000E	C	QueryValueExW

Figure 2 - Strings in the Pandora sample

Usually when one encounters malware with encrypted strings, there are two approaches:

- Use a dynamic approach, such as debugging or emulation, and use the malware's own string decryption functions to do the work.

- Understand the decryption function in such detail that one can reimplement it in a simple script. This is usually the way to go when the encryption is a simple one-byte XOR.

In the case of Pandora, there is not one, but at least 14 different string decryption functions, so reimplementing the decryption algorithm might not always be feasible.

Emulation

Emulation allows us to pretend that the code is running on a CPU, but instead of running on a real CPU, the emulation software runs the code. Emulation is usually very slow compared to real execution. However, it allows us full control over what we want to run and a very high level of interaction with the emulated code. With an emulator, for instance, we can emulate just one function of the malware (or even just a couple of lines of code) and evaluate the state of the program at every instruction. A big advantage of emulation in this case is that we can do it directly in IDA Pro.

flare-emu

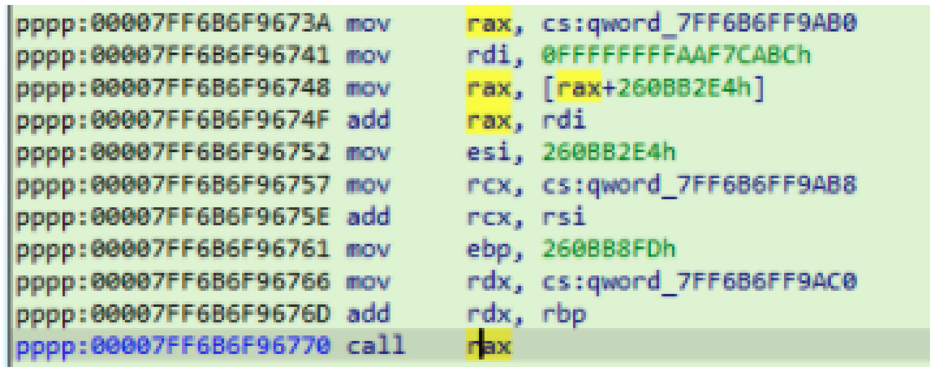
flare-emu is an emulation framework created by the FLARE team at Mandiant. It builds on the well-known emulation engine called Unicorn Engine and IDAPython. One could use the Unicorn Engine directly but flare-emu hides some of its complexities. Essentially, one can define what (which part of the code) one wants to emulate and define callback functions for specific hooks that will be called when the emulation reaches that hook. A good example is the callHook parameter, which accepts a callback function that will be called every time a CALL instruction is about to be emulated. In this callback function we can implement whatever we want to do in that situation, i.e., dump registers, change data, skip call, etc. flare-emu turned out to be very straightforward and relatively easy to use (which does not mean that I did not need to look into its source code to figure out a few things).

Solving the Challenges

After this extensive introduction, let's finally write some code to solve these challenges with an IDAPython script.

Function Call Obfuscation

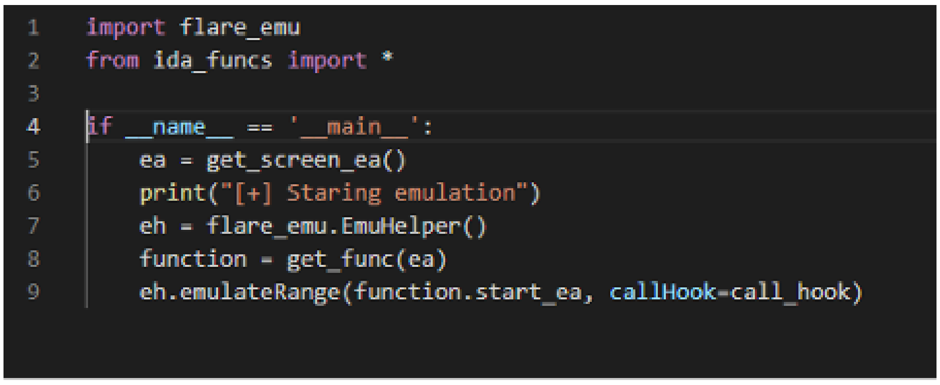
Figure 3 shows again the problem we are trying to solve first. This is the first function call in the main()function in the unpacked part of Pandora's code. At this point we can be fairly sure that if we emulate the main()function and check rax's value before it is called then we get the right result. When we are already there, we could read out the arguments of the function call as well and add all this information to the assembly code as a comment in IDA Pro.



```
pppp:00007FF6B6F9673A mov     rax, cs:qword_7FF6B6FF9AB0
pppp:00007FF6B6F96741 mov     rdi, 0FFFFFFFFAAF7CABCh
pppp:00007FF6B6F96748 mov     rax, [rax+260BB2E4h]
pppp:00007FF6B6F9674F add     rax, rdi
pppp:00007FF6B6F96752 mov     esi, 260BB2E4h
pppp:00007FF6B6F96757 mov     rcx, cs:qword_7FF6B6FF9AB8
pppp:00007FF6B6F9675E add     rcx, rsi
pppp:00007FF6B6F96761 mov     ebp, 260BB8FDh
pppp:00007FF6B6F96766 mov     rdx, cs:qword_7FF6B6FF9AC0
pppp:00007FF6B6F9676D add     rdx, rbp
pppp:00007FF6B6F96770 call    rax
```

Figure 3 - Function call obfuscation

Let's start to put together our IDAPython script. Figure 4 shows how we can initialize the emulation. When we launch the script, it should emulate the function in which the cursor stands at the moment in IDA (returned by get_screen_ea()).



```python
import flare_emu
from ida_funcs import *

if __name__ == '__main__':
    ea = get_screen_ea()
    print("[+] Staring emulation")
    eh = flare_emu.EmuHelper()
    function = get_func(ea)
    eh.emulateRange(function.start_ea, callHook=call_hook)
```

Figure 4 - Initialize the emulation

To initialize flare-emu, we just need to instantiate an EmuHelper. Flare-emu offers different ways to run our emulation. We use the emulateRange() function, which is used to specify a memory range we want to emulate. We set the start address to the beginning of the function and the end address can be omitted (None in python) which means that the emulation will run until a return type instruction is reached. Note that the iterateAllPaths() instead of the emulateRange() should also work, however that caused problems due to another obfuscation technique in Pandora, which is not in the scope of this post. But in a less complex malware iterateAllPaths()could be a better option.

When one of flare-emu's emulation functions is called (emulateRange()in this case), then the emulation starts. The framework allows us to provide additional details for the emulation, such as processor state with registers and stack, or data for the callback functions, but we don't need those at this point.

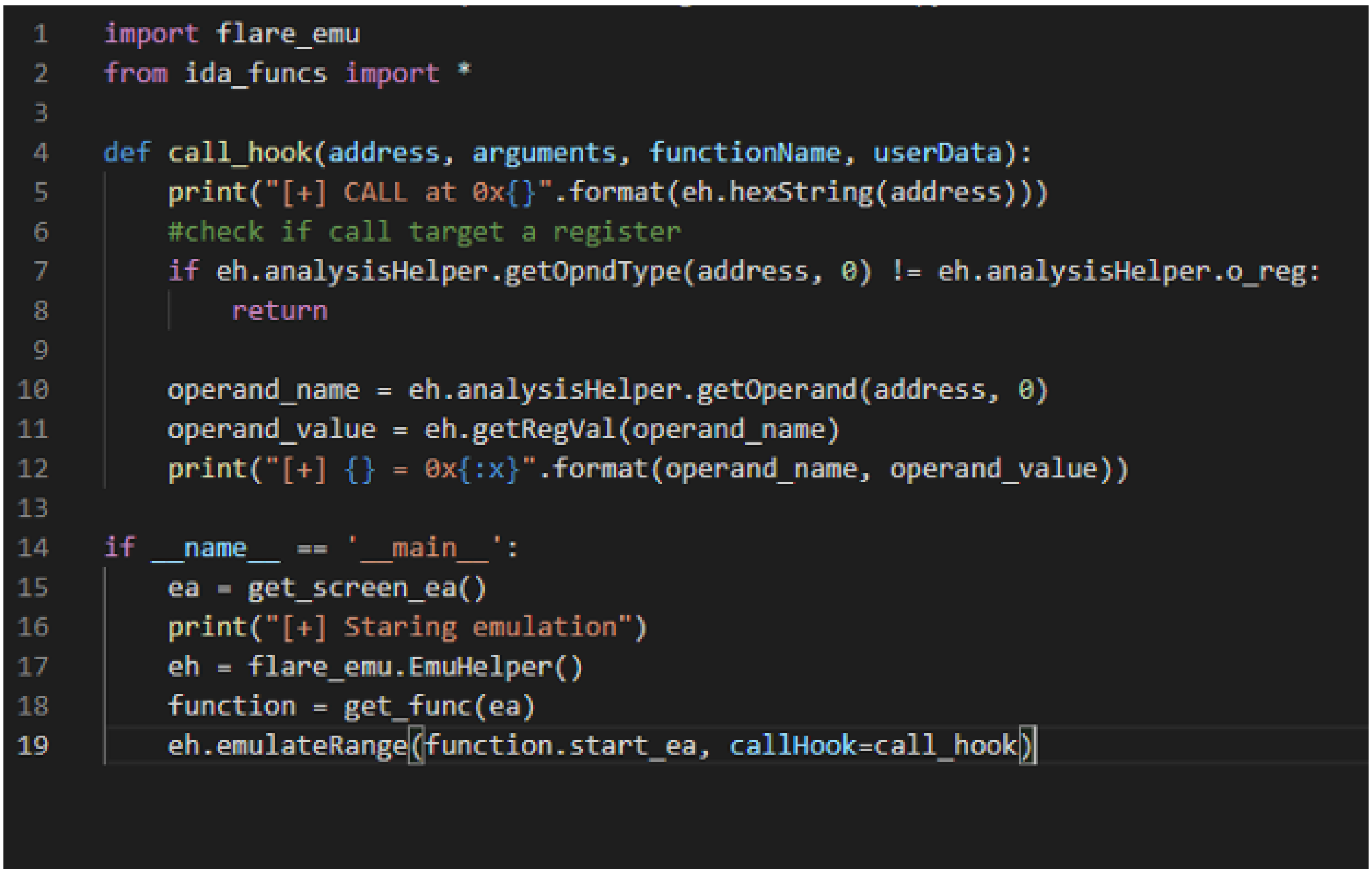
emulateRange()allows us to define callback functions for different hooks:

- **instruction hook**: called before the emulation of every instruction. I used this to color every instruction that was emulated in IDA to visualize the coverage of the emulation.

- **call hook**: called whenever a CALL type of instruction will be emulated. Note that the called function is not emulated by default.

- **memory access hook**: called whenever memory is accessed for reading or writing.

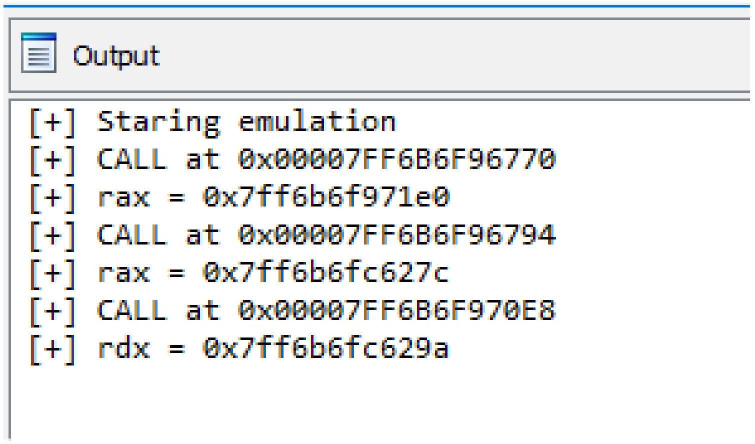
For our current task we only need the callHook. As you can see in line 9 in Figure 4, we already passed the call_hook function name as the callHook parameter (might not be the cleverest naming). Next, we need to define the callHook function, which can be seen in Figure 5.



```python
import flare_emu
from ida_funcs import *

def call_hook(address, arguments, functionName, userData):
    print("[+] CALL at 0x{}".format(eh.hexString(address)))
    #check if call target a register
    if eh.analysisHelper.getOpndType(address, 0) != eh.analysisHelper.o_reg:
        return

    operand_name = eh.analysisHelper.getOperand(address, 0)
    operand_value = eh.getRegVal(operand_name)
    print("[+] {} = 0x{:x}".format(operand_name, operand_value))

if __name__ == '__main__':
    ea = get_screen_ea()
    print("[+] Staring emulation")
    eh = flare_emu.EmuHelper()
    function = get_func(ea)
    eh.emulateRange(function.start_ea, callHook=call_hook)
```

Figure 5 - First implementation of call_hook ()

We created the call_hook() function, that will be called by the emulator every time before a CALL instruction is emulated. In its current state, this function will log that it was executed, then uses the analysisHelper to check whether the operand in the current CALL instruction is a register or not. If not, then we can return because only the register case is interesting for us. Then we recover the register's name (operand_name) and its value (operand_name) and log them for now. If we run the script against the main function, then we get the results in Figure 6. Note that due to the numerous other evil obfuscations in the Pandora code, this simple script won’t be able to emulate the whole function. But this can be done by extending the script.



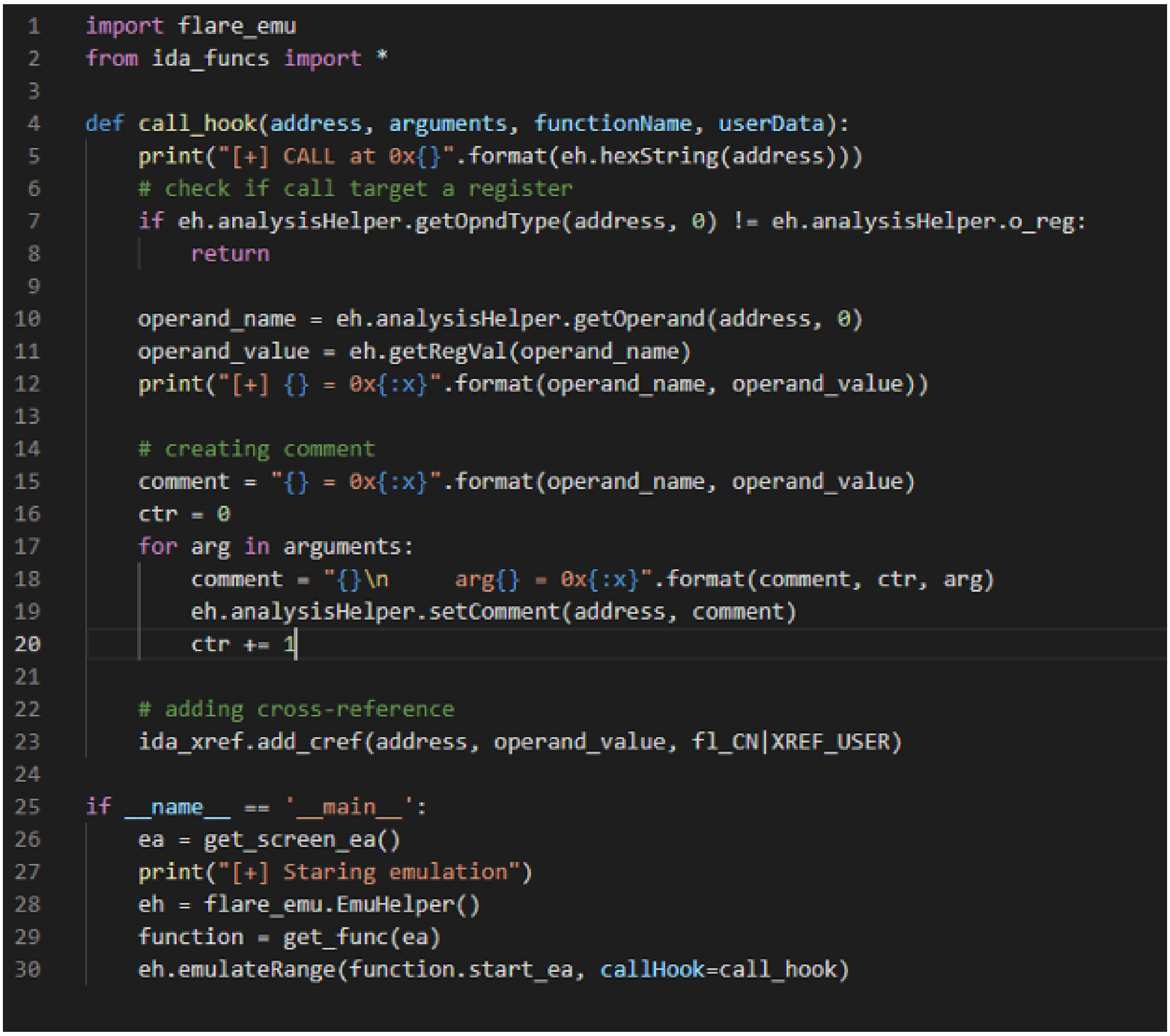
```
[+] Staring emulation
[+] CALL at 0x00007FF6B6F96770
[+] rax = 0x7ff6b6f971e0
[+] CALL at 0x00007FF6B6F96794
[+] rax = 0x7ff6b6fc627c
[+] CALL at 0x00007FF6B6F970E8
[+] rdx = 0x7ff6b6fc629a
```

Figure 6 - Results of the first test

The emulation found three CALL instructions and printed the values of the operand registers. If we think about it, we have mostly solved the problem of function call obfuscation, because we now know which addresses are called by the different CALL instructions. Now we just need to add this to the disassembly in IDA. These are the things we want to do in IDA whenever we resolve a CALL instruction:

- Add a comment with the address of the function that is being called
- Add a comment with the arguments for that function call
- Add a cross-reference in IDA to the function that is called

Figure 7 shows the updated code.

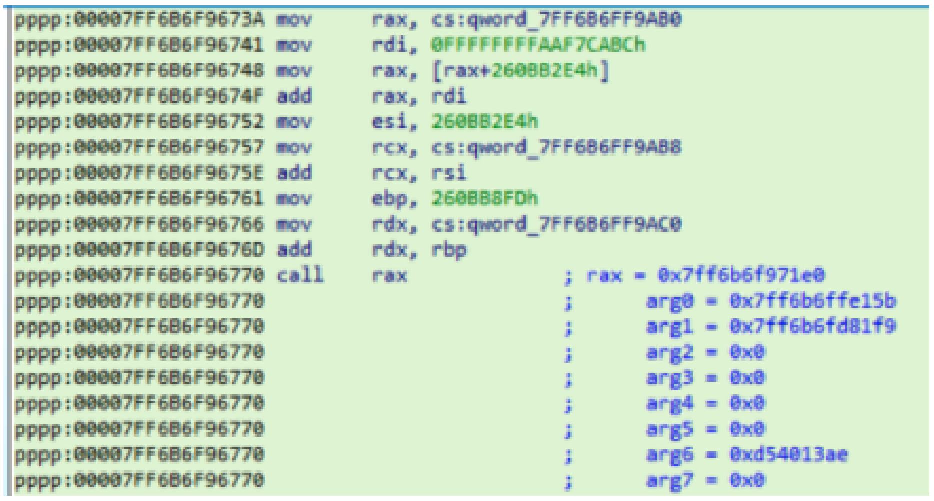


```
1   import flare_emu
2   from ida_funcs import *
3
4   def call_hook(address, arguments, functionName, userData):
5       print("[+] CALL at 0x{}".format(eh.hexString(address)))
6       # check if call target a register
7       if eh.analysisHelper.getOpndType(address, 0) != eh.analysisHelper.o_reg:
8           return
9
10      operand_name = eh.analysisHelper.getOperand(address, 0)
11      operand_value = eh.getRegVal(operand_name)
12      print("[+] {} = 0x{:x}".format(operand_name, operand_value))
13
14      # creating comment
15      comment = "{} = 0x{:x}".format(operand_name, operand_value)
16      ctr = 0
17      for arg in arguments:
18          comment = "{}\n     arg{} = 0x{:x}".format(comment, ctr, arg)
19          eh.analysisHelper.setComment(address, comment)
20          ctr += 1
21
22      # adding cross-reference
23      ida_xref.add_cref(address, operand_value, fl_CN|XREF_USER)
24
25  if __name__ == '__main__':
26      ea = get_screen_ea()
27      print("[+] Staring emulation")
28      eh = flare_emu.EmuHelper()
29      function = get_func(ea)
30      eh.emulateRange(function.start_ea, callHook=call_hook)
```

Figure 7 - Adding comment and cross-reference

When creating the comment we use a nice feature from flare-emu. It allows us to get the function parameters in an architecture-independent way. This malware is x86_64 so we could just take rcx, rdx, r8, r9, and the stack, but this way we don't have to deal with that. One of the arguments the call hook gets is the arguments variable and this will contain the values flare-emu thinks are the parameters of this function call. Of course, without analyzing the called function, we won't know how many parameters are expected so we will just print all of them.

At the end (line 23) we add an IDA cross-reference, which will be a great help further along in our analysis. If we run this code again on the main function, we get the result in Figure 8.



```
pppp:00007FF6B6F9673A mov     rax, cs:qword_7FF6B6FF9AB0
pppp:00007FF6B6F96741 mov     rdi, 0FFFFFFFFAAF7CABCh
pppp:00007FF6B6F96748 mov     rax, [rax+2608B2E4h]
pppp:00007FF6B6F9674F add     rax, rdi
pppp:00007FF6B6F96752 mov     esi, 2608B2E4h
pppp:00007FF6B6F96757 mov     rcx, cs:qword_7FF6B6FF9AB8
pppp:00007FF6B6F9675E add     rcx, rsi
pppp:00007FF6B6F96761 mov     ebp, 2608B8FDh
pppp:00007FF6B6F96766 mov     rdx, cs:qword_7FF6B6FF9AC0
pppp:00007FF6B6F9676D add     rdx, rbp
pppp:00007FF6B6F96770 call    rax                  ; rax = 0x7ff6b6f971e0
pppp:00007FF6B6F96770                              ;       arg0 = 0x7ff6b6ffe15b
pppp:00007FF6B6F96770                              ;       arg1 = 0x7ff6b6fd81f9
pppp:00007FF6B6F96770                              ;       arg2 = 0x0
pppp:00007FF6B6F96770                              ;       arg3 = 0x0
pppp:00007FF6B6F96770                              ;       arg4 = 0x0
pppp:00007FF6B6F96770                              ;       arg5 = 0x0
pppp:00007FF6B6F96770                              ;       arg6 = 0xd54013ae
pppp:00007FF6B6F96770                              ;       arg7 = 0x0
```

Figure 8 - Results of the function call resolution

Encrypted String

Now that we have the first problem out of the way, and have an emulation framework to work with, we can move onto our second challenge, decrypting the strings. To be able to know which function to emulate to get the strings decrypted, our only requirement is that we need to know which functions are decryption functions. As always, reverse engineering is an iterative process. Once we run the script we wrote on the main function, then we can start to analyze the called functions. So how do we figure out if a function is a decryption function? (Spoiler: 0x7ff6b6f971e0 points to a decryption function.)

A) We see it in IDA. Without diving deep in the function at 0x7ff6b6f971e0, we can see in the graph view (Figure 10) that it is fairly simple and has some loops.

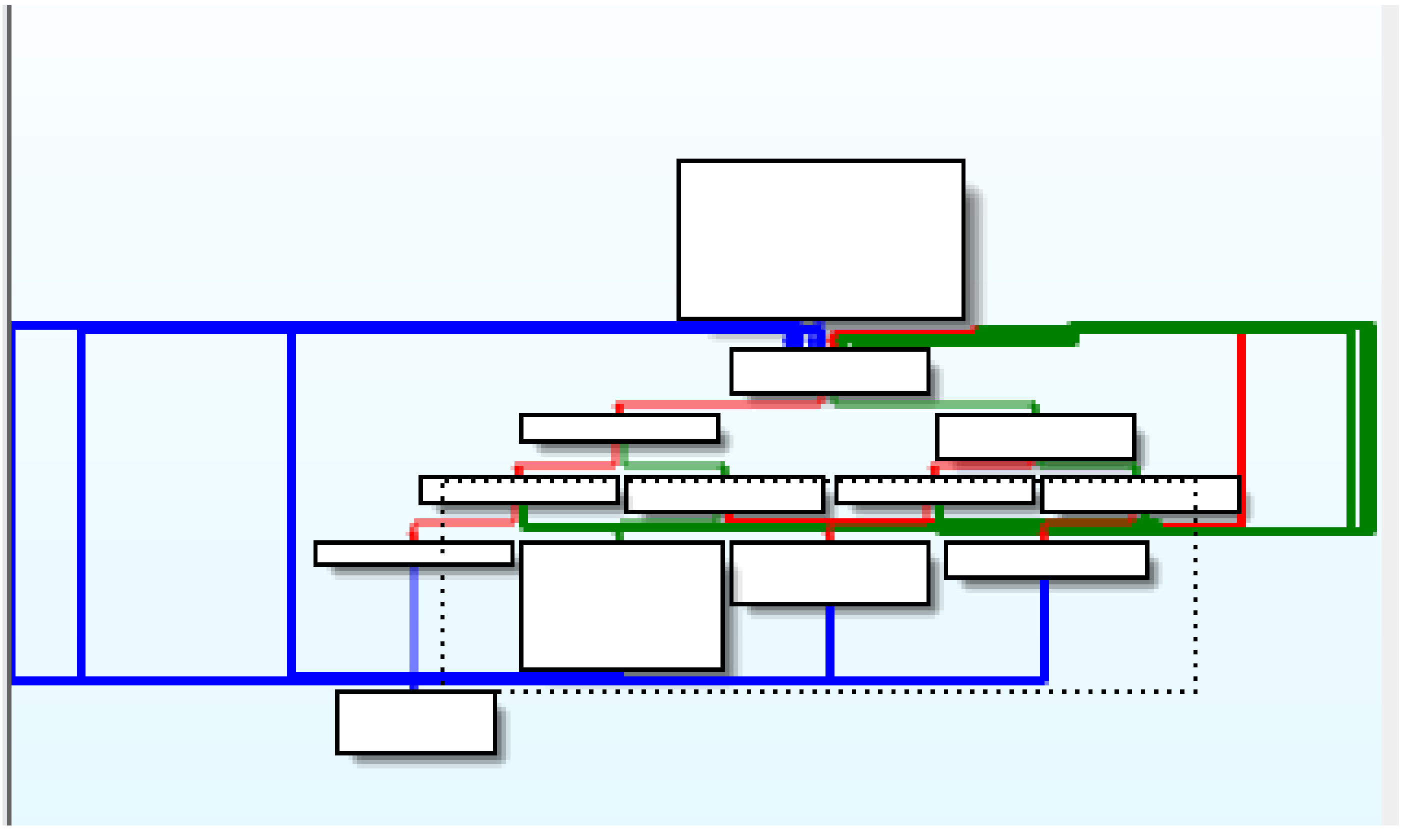
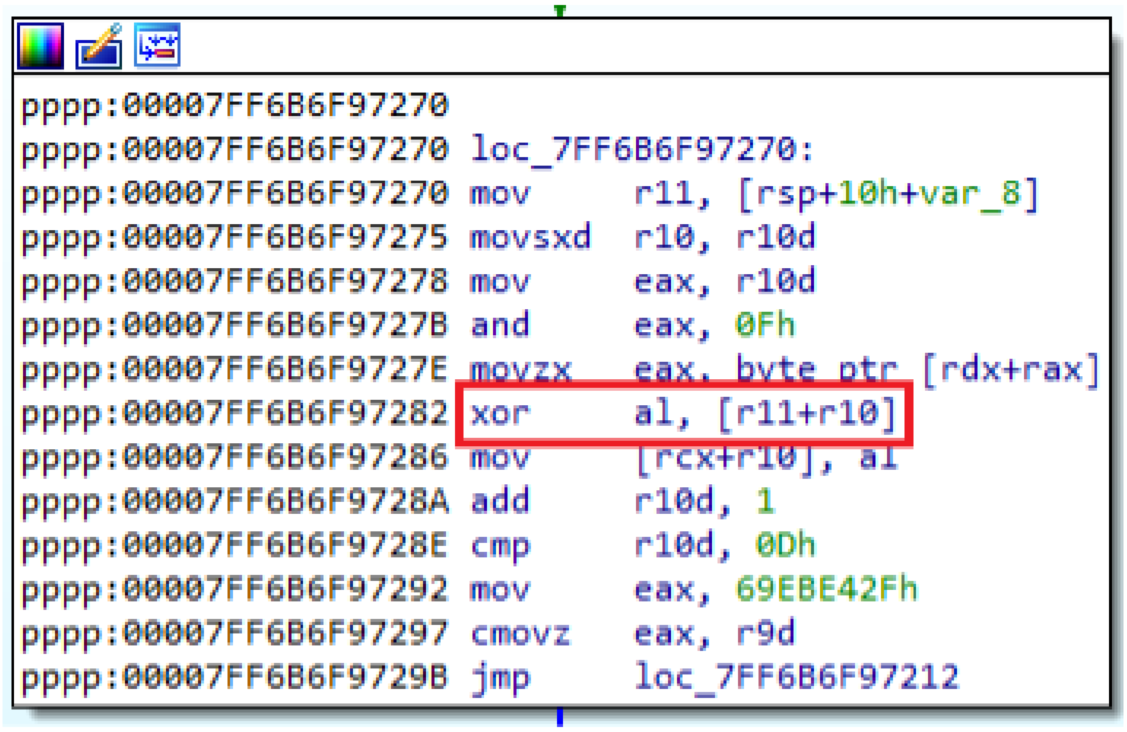


Figure 9. - Graph view at the function at 0x7ff6b6f971e0

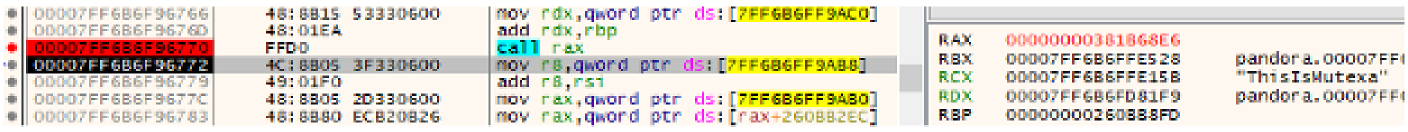
If we scroll through the code, we find the basic block in Figure 10, where we see that it iterates through some value and XORs it. This indicates that it might be an XOR-based encoding/encryption.



```
pppp:00007FF6B6F97270
pppp:00007FF6B6F97270 loc_7FF6B6F97270:
pppp:00007FF6B6F97270 mov     r11, [rsp+10h+var_8]
pppp:00007FF6B6F97275 movsxd  r10, r10d
pppp:00007FF6B6F97278 mov     eax, r10d
pppp:00007FF6B6F9727B and     eax, 0Fh
pppp:00007FF6B6F9727E movzx   eax, byte ptr [rdx+rax]
pppp:00007FF6B6F97282 xor     al, [r11+r10]
pppp:00007FF6B6F97286 mov     [rcx+r10], al
pppp:00007FF6B6F9728A add     r10d, 1
pppp:00007FF6B6F9728E cmp     r10d, 0Dh
pppp:00007FF6B6F97292 mov     eax, 69EBE42Fh
pppp:00007FF6B6F97297 cmovz   eax, r9d
pppp:00007FF6B6F9729B jmp     loc_7FF6B6F97212
```

Figure 10 - XOR indicates decoding/decryption

B) We see it in a debugger. Parallel to our static analysis, we of course can also debug the malware (in a safe environment). In the debugger when we see a function that gets some addresses as an input and returns a string, that could mean that it is a decryption function. Figure 11 shows when the function at 0x7ff6b6f971e0 returns and indeed it returns the string “ThisIsMutexa” in rcx.



```
00007FF6B6F96766    48:8B15 53330600    mov rdx,qword ptr ds:[7FF6B6FF9AC0]
00007FF6B6F9676D    48:01EA             add rdx,rbp
00007FF6B6F96770    FFD0                call rax
00007FF6B6F96772    4C:8B05 3F330600    mov r8,qword ptr ds:[7FF6B6FF9AB8]
00007FF6B6F96779    49:01F0             add r8,rsi
00007FF6B6F9677C    48:8B05 2D330600    mov rax,qword ptr ds:[7FF6B6FF9AB0]
00007FF6B6F96783    48:8B80 ECB20826    mov rax,qword ptr ds:[rax+2608B2EC]
```

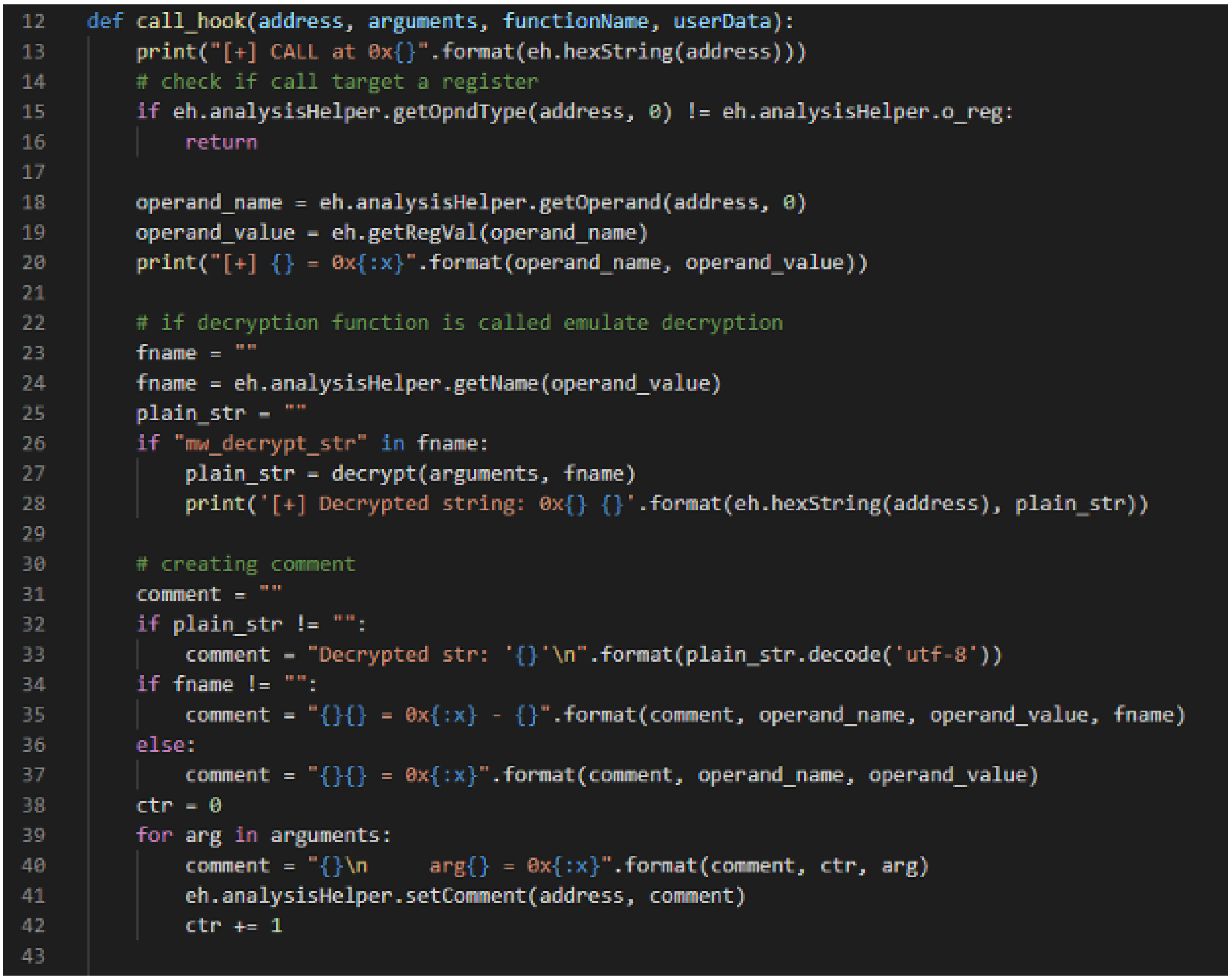
```
RAX    00000000381868E6
RBX    00007FF6B6FFE528    pandora.00007FF
RCX    00007FF6B6FFE15B    "ThisIsMutexa"
RDX    00007FF6B6FD81F9    pandora.00007FF
RBP    000000002608B8FD
```

Figure 11 - The decrypted string appears in rcx

Once we know that a function is a decryption function, we can rename it accordingly (we used mw_decrypt_str()). Interestingly, Pandora uses multiple decryption functions, which we discovered slowly as we dived deeper into the code. At the end we identified 14 different decryption functions, however most of them looked very similar to Figure 9, which allowed us to quickly see if a function is just another decryption function.

Once we know (some) of the decryption functions, we can improve our IDAPython script to emulate the function call whenever we see that a decryption function is called. This is actually very similar to one of the examples in flare-emu’s documentation, which shows how well such code can often be reused.

Figure 12 shows the updated call_hook() function. Starting from line 23, we first check whether the function at the address we are calling has a name that contains the string mw_decrypt_str. This is how we decide whether the called function is a decryption function. This is not a very scientific method, but we want to reverse malware, not get a PhD.

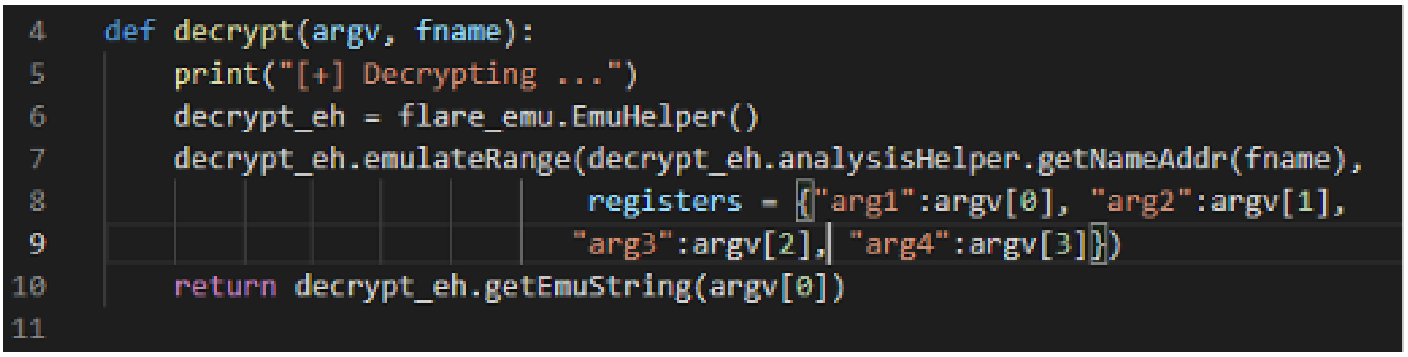


```python
def call_hook(address, arguments, functionName, userData):
    print("[+] CALL at 0x{}".format(eh.hexString(address)))
    # check if call target a register
    if eh.analysisHelper.getOpndType(address, 0) != eh.analysisHelper.o_reg:
        return

    operand_name = eh.analysisHelper.getOperand(address, 0)
    operand_value = eh.getRegVal(operand_name)
    print("[+] {} = 0x{:x}".format(operand_name, operand_value))

    # if decryption function is called emulate decryption
    fname = ""
    fname = eh.analysisHelper.getName(operand_value)
    plain_str = ""
    if "mw_decrypt_str" in fname:
        plain_str = decrypt(arguments, fname)
        print('[+] Decrypted string: 0x{} {}'.format(eh.hexString(address), plain_str))

    # creating comment
    comment = ""
    if plain_str != "":
        comment = "Decrypted str: '{}'\n".format(plain_str.decode('utf-8'))
    if fname != "":
        comment = "{}{} = 0x{:x} - {}".format(comment, operand_name, operand_value, fname)
    else:
        comment = "{}{} = 0x{:x}".format(comment, operand_name, operand_value)
    ctr = 0
    for arg in arguments:
        comment = "{}\n     arg{} = 0x{:x}".format(comment, ctr, arg)
        eh.analysisHelper.setComment(address, comment)
        ctr += 1

```

Figure 12 - Adding decryption to the call_hook()

If it is a decryption function, then we call the decrypt()function in our script. This will return the decrypted plain text string. Then we create a comment that will include the decrypted string as well.

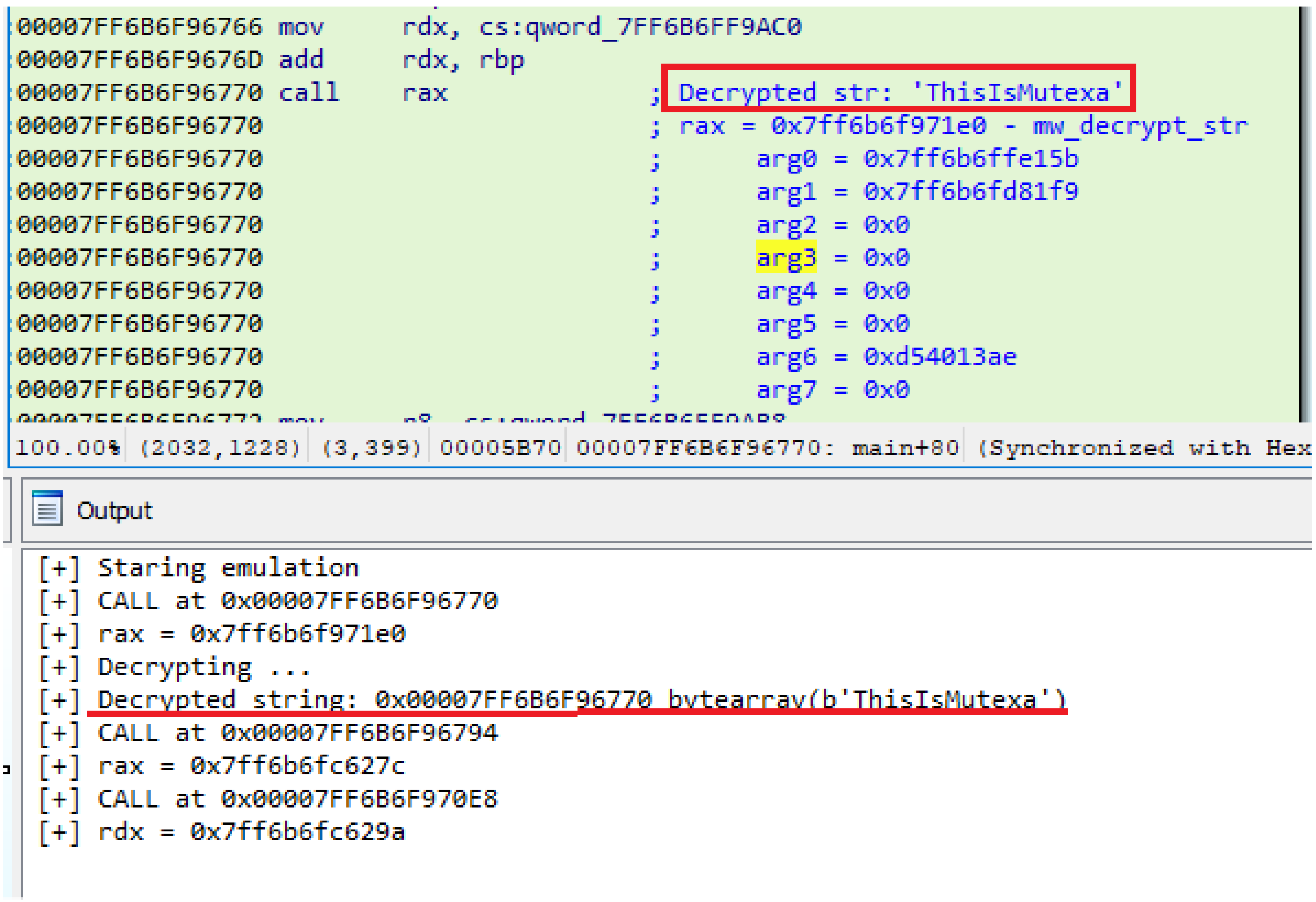
How the decryption is emulated can be seen in Figure 13. We create a new EmuHelper instance and when starting emulateRange, we use the function name (fname) to get the function's address as the start address. We also pass the first four elements of the argv array as the argument registers. At the end we return the value in argv[0], which should contain the address of the decrypted string.



```python
def decrypt(argv, fname):
    print("[+] Decrypting ...")
    decrypt_eh = flare_emu.EmuHelper()
    decrypt_eh.emulateRange(decrypt_eh.analysisHelper.getNameAddr(fname),
                            registers = {"arg1":argv[0], "arg2":argv[1],
                           "arg3":argv[2], "arg4":argv[3]})
    return decrypt_eh.getEmuString(argv[0])

```

Figure 13 - Emulating the decryption

After running the script in IDA, the results are shown in Figure 12. The decrypted string was ThisIsMutexa, which was added to the comment and logged in the output.



```
00007FF6B6F96766 mov     rdx, cs:qword_7FF6B6FF9AC0
00007FF6B6F9676D add     rdx, rbp
00007FF6B6F96770 call    rax              ; Decrypted str: 'ThisIsMutexa'
00007FF6B6F96770                          ; rax = 0x7ff6b6f971e0 - mw_decrypt_str
00007FF6B6F96770                          ;      arg0 = 0x7ff6b6ffe15b
00007FF6B6F96770                          ;      arg1 = 0x7ff6b6fd81f9
00007FF6B6F96770                          ;      arg2 = 0x0
00007FF6B6F96770                          ;      arg3 = 0x0
00007FF6B6F96770                          ;      arg4 = 0x0
00007FF6B6F96770                          ;      arg5 = 0x0
00007FF6B6F96770                          ;      arg6 = 0xd54013ae
00007FF6B6F96770                          ;      arg7 = 0x0
100.00% (2032,1228) (3,399) 00005B70 00007FF6B6F96770: main+80 (Synchronized with Hex
```

```
[+] Staring emulation
[+] CALL at 0x00007FF6B6F96770
[+] rax = 0x7ff6b6f971e0
[+] Decrypting ...
[+] Decrypted string: 0x00007FF6B6F96770 bytearray(b'ThisIsMutexa')
[+] CALL at 0x00007FF6B6F96794
[+] rax = 0x7ff6b6fc627c
[+] CALL at 0x00007FF6B6F970E8
[+] rdx = 0x7ff6b6fc629a
```

Figure 14 - String decryption is successful

Now we can automatically decrypt strings. As we progress with the analysis of the code and more decryption functions are discovered, we can just rerun the script on the functions that call these decryption functions to recover the plain text strings.

Conclusion

The Pandora ransomware is packed with obfuscation and anti-reverse-engineering techniques. In this post we looked at two of these: function call obfuscation and string encryption. We used the flare-emu emulation framework to write an IDAPython script to resolve the addresses and arguments of the function calls as well as emulate the decryption functions to recover the strings as plain text. The final script can be developed further to deal with the other anti-reverse-engineering challenges discussed in the deep dive analysis of the Pandora ransomware.

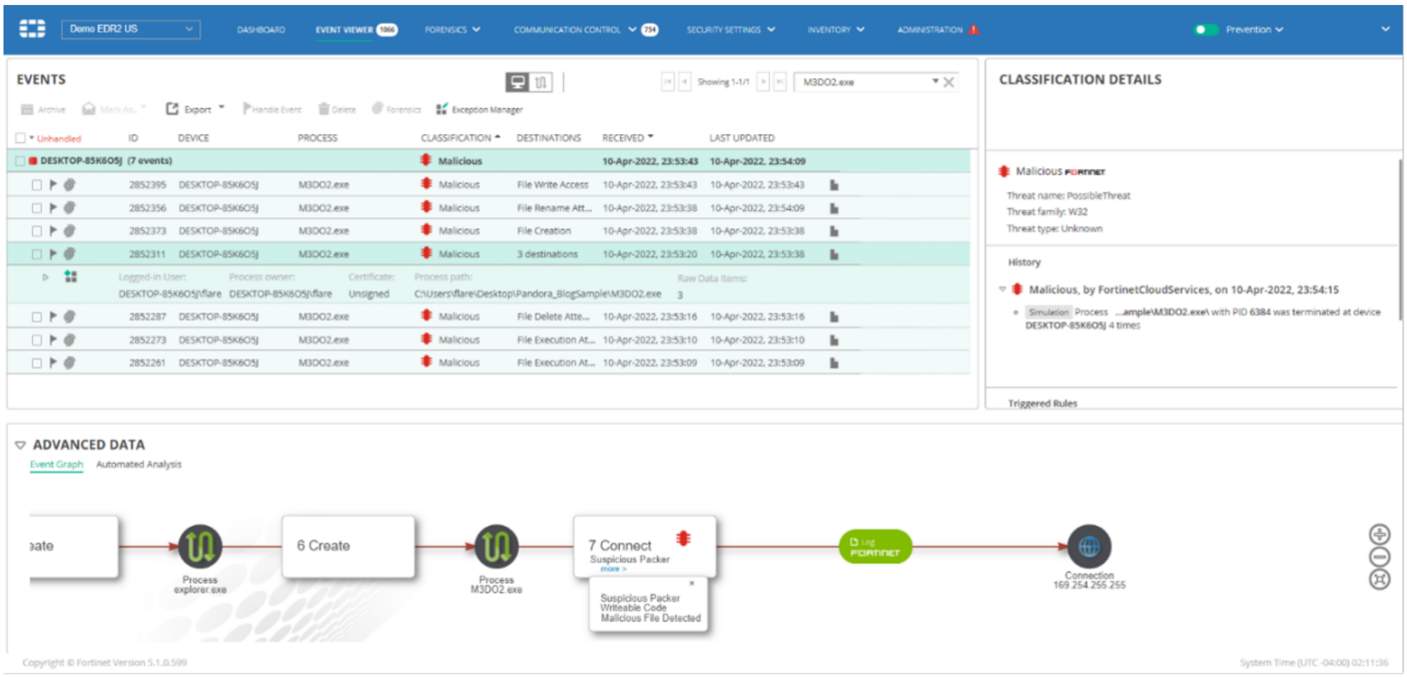
Fortinet Protection

The analyzed Pandora ransomware sample is detected by the following (AV) signature:

W64/Filecoder.EGYTYFD!tr.ransom

FortiEDR also detects and mitigates execution of Pandora ransomware through the combination of behavioral analysis, and integration with machine learning and threat intelligence feeds. Execution of the Pandora sample analyzed as part of this blog triggers seven rules resulting in nine security events.

Triggered rules were a result of pre-execution analysis and post-execution behaviors. These security events can be observed below in Figure 15.



EVENTS

Unhandled	ID	DEVICE	PROCESS	CLASSIFICATION	DESTINATIONS	RECEIVED	LAST UPDATED
DESKTOP-85K6O5J (7 events)				Malicious		10-Apr-2022, 23:53:43	10-Apr-2022, 23:54:09
	2852395	DESKTOP-85K6O5J	M3DO2.exe	Malicious	File Write Access	10-Apr-2022, 23:53:43	10-Apr-2022, 23:53:43
	2852356	DESKTOP-85K6O5J	M3DO2.exe	Malicious	File Rename Att...	10-Apr-2022, 23:53:38	10-Apr-2022, 23:54:09
	2852373	DESKTOP-85K6O5J	M3DO2.exe	Malicious	File Creation	10-Apr-2022, 23:53:38	10-Apr-2022, 23:53:38
	2852311	DESKTOP-85K6O5J	M3DO2.exe	Malicious	3 destinations	10-Apr-2022, 23:53:20	10-Apr-2022, 23:53:38
	2852287	DESKTOP-85K6O5J	M3DO2.exe	Malicious	File Delete Atte...	10-Apr-2022, 23:53:16	10-Apr-2022, 23:53:16
	2852273	DESKTOP-85K6O5J	M3DO2.exe	Malicious	File Execution At...	10-Apr-2022, 23:53:10	10-Apr-2022, 23:53:10
	2852261	DESKTOP-85K6O5J	M3DO2.exe	Malicious	File Execution At...	10-Apr-2022, 23:53:09	10-Apr-2022, 23:53:09

Logged-in User:	Process owner:	Certificate:	Process path:	Raw Data Items:
DESKTOP-85K6O5J\flare	DESKTOP-85K6O5J\flare	Unsigned	C:\Users\flare\Desktop\Pandora_BlogSample\M3DO2.exe	3

CLASSIFICATION DETAILS

Malicious FORTINET

Threat name: PossibleThreat
Threat family: W32
Threat type: Unknown

History

Malicious, by FortinetCloudServices, on 10-Apr-2022, 23:54:15

- Simulation Process ...ample\M3DO2.exe\ with PID 6384 was terminated at device DESKTOP-85K6O5J 4 times

Triggered Rules

ADVANCED DATA

Copyright © Fortinet Version 5.1.0.599

System Time (UTC -04:00) 02:11:36

Figure 15 - FortiEDR security events generated following execution of Pandora ransomware sample. Note that during this execution, FortiEDR was set to only log events rather than mitigate, to properly demonstrate detections post-execution.

Pre-execution detections included: identifying the malicious file (hash based), detection of a suspicious packer, and presence of writeable code. Post-execution detections included: detection of each file encryption attempt, detection of encrypted file rename attempt, dropping of the ransom note, and attempts to access SMB shares.

In Protect mode, FortiEDR will detect and mitigate detected behavior. In the case of Pandora, this will prevent execution of the ransomware, mitigating malicious activity before it occurs, and will prevent subsequent file encryption attempts if the adversary is able to execute the sample. The post-exploitation detections are not dependent on signature, meaning they will effectively mitigate this activity for newer Pandora variants even with no prior knowledge of the samples.

IOCs

Mutex: ThisIsMutexa
Ransom note: Restore_My_Files.txt
SHA256 hash of hardcoded public key:
7b2c21eea03a370737d2fe7c108a3ed822be848cce07da2ddc66a30bc558af6b
SHA256 hash of sample: 5b56c5d86347e164c6e571c86dbf5b1535eae6b979fede6ed66b01e79ea33b7b

ATT&CK TTPs

TTP Name	TTP ID	Description
Obfuscated Files or Information: Software Packing	T1027.002	Modified UPX packer

Impair Defenses: Disable Windows Event Logging	T1562.002	Disable event logging
Impair Defenses: Disable or Modify Tools	T1562.001	Bypass AMSI
Data from Local System	T1005	Searches unmounted drives and partitions
Modify Registry	T1112	Cryptographic keys are stored in the registry
Data Encrypted for Impact	T1486	As a ransomware it encrypts files
Command and Scripting Interpreter	T1059	Uses cmd.exe to remove the shadow copies
System Information Discovery	T1082	Collects system information with GetSystemInfo()
File and Directory Discovery	T1083	Discovers drives and enumerates filesystems
Inhibit System Recovery	T1490	Deletes shadow copies
Service Stop	T1489	Terminates processes if they lock a file