SEMINAR PAPER

In IT-Security

Reverse Engineering and Malware Analysis Assignments

By: Benjamin Medicke, BSc

Student Number: 2010303027

Supervisors: Tibor Éliás, MSc.
Dipl.-Ing. Florian Nentwich

Wien, July 16, 2021

Contents

1 Reverse Engineering Exercise 1: Basics

[Binary Provided: `2010303027_hw_1_exercise_1.exe`]

1.1 Calculate the value that the instruction at `0x40301B` loads into the RDI register!

- `lea rdi, ds:400FFEh[rcx*8] ; RDI = 0x????????????????`
- *note: the value in my example differs slightly*

To answer this question we need to know the value of `rcx`. I've extracted the relevant instructions with IDA: see Code 1.

```
1 .text:0000000000403014                 mov     rcx, 0Ah
2 .text:000000000040301B                 lea     rdi, ds:400FFBh[rcx*8]
```

Code 1: excerpt of relevant instructions task 1.1

The mov instruction (line 1) copies the value `Ah` (`10`) into `rcx`. With that out of the way we can calculate the value with Python:
`hex(0x400ffb + 8 * 0xa)` which return `0x40104b` to us.

This means that after executing the instruction (at least for the first time) the calculated value would be: **`40104Bh`**.

Let's test that in IDA. We set a break point (*F2*) at the `lea` instruction and step over it (*F8*). As we can see in Figure 1 it is indeed correct.

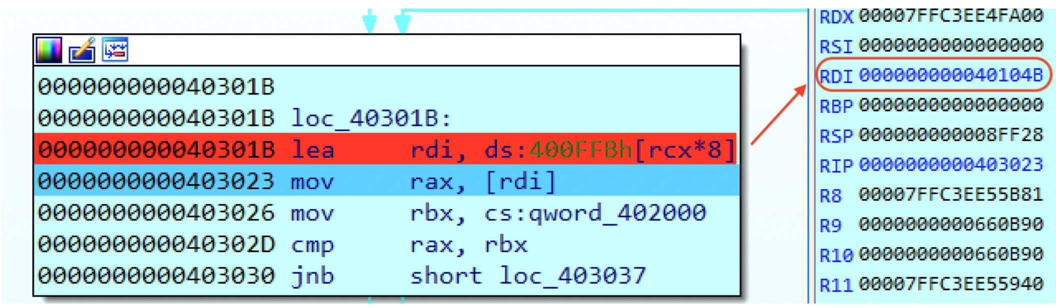


Figure 1: validating our calculation with IDA

1.2 What is loaded into RAX when the instruction at address `0x403023` is run for the first time?

- `mov rax, qword[rdi] ; RAX = 0x????????????????`

This builds on the previous example (the value of `rdi`). See Code 2 for all relevant instructions.

```
1 .text:0000000000403014            mov     rcx, 0Ah
2 .text:000000000040301B            lea     rdi, ds:400FFBh[rcx*8] ; rdi = 0x40104b
3 .text:0000000000403023            mov     rax, [rdi]
```

Code 2: excerpt of relevant instructions task 1.2

The second `mov` instruction (line 3) copies the value that the address in `rdi` is pointing to into `rax`. Note the square brackets denoting dereferencing. Now all we need to know is where `0x40104b` points to.

In IDA we can press *g* and jump to an address. We can either enter `0x40104b` or (if we're lazy and have stepped far enough) simply `rdi`. This leads us to to the data section (Code 3).

```
.data:000000000040104B db 0AFh
.data:000000000040104C db 0BEh
.data:000000000040104D db 0ADh
.data:000000000040104E db 0DEh
.data:000000000040104F db    0
.data:0000000000401050 db    0
.data:0000000000401051 db    0
.data:0000000000401052 db    0
```

Code 3: excerpt of relevant data section task 1.2

Reading it from the bottom up, our value is: **`DEADBEAFh`**, a classic.

Same as before, let's make sure with IDA by stepping over it and taking a look at the register. Figure 2 shows us that we were correct.

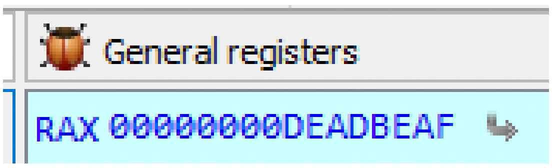


Figure 2: validating our prediction with IDA

1.3 Is it possible that the application is defining an array of data somewhere in the program? If so what's the address of the first element?

- Examine how the program access its data before you write the answer.

Let's get a rough overview of the program. IDAs graph view (*space*) is very useful for tasks like this. see Figure 3, it starts right after the `scanf()` call and setting `rcx` to `10`.

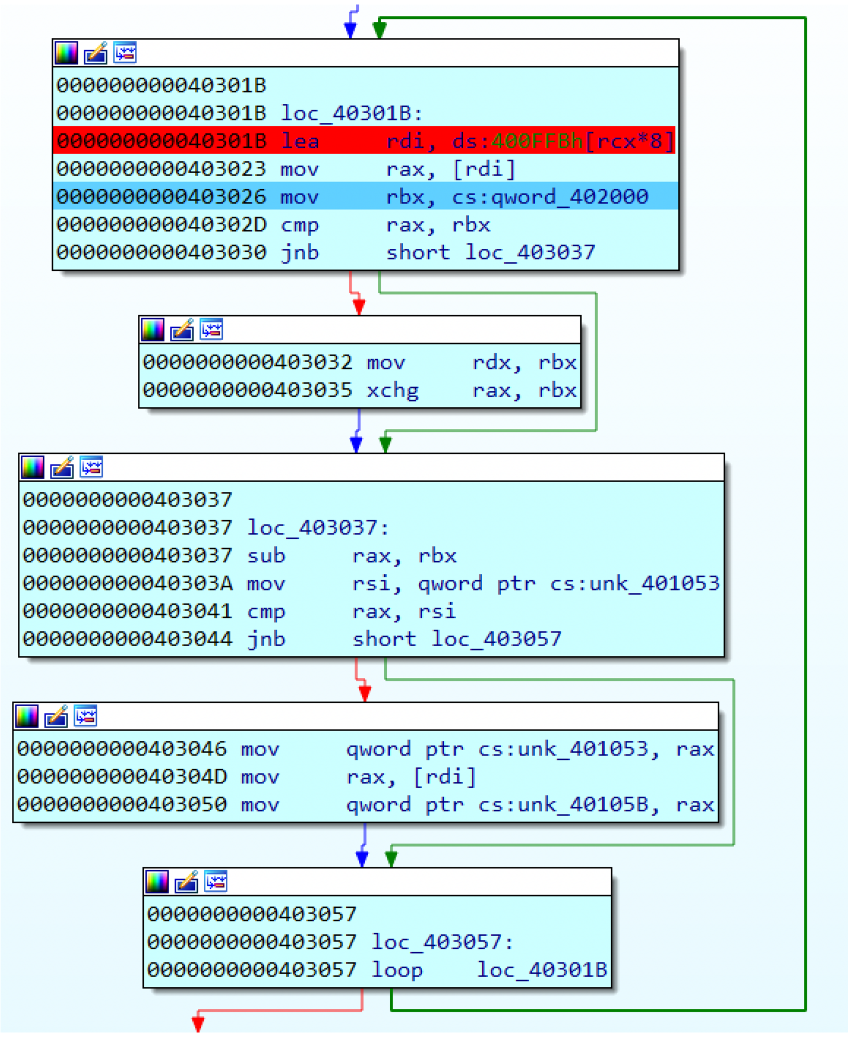


Figure 3: rough overview of the first program

The **first block** of code from `40301Bh` to `403030h` uses the value of the `rcx` register to calculate an address in the data section (notably in multiples of 8). As we have seen before the value of that address is subsequently loaded into the `rax` register.

The **last block** of code (a single instruction) at `403057h` (conditionally) loops back to the first block. The `loop` instructions does the following:

- the counter register (`rcx`) is decremented by 1
- if the counter register is 0 the loop is terminated
- if not a jump is executed (in our case back to the first block) [3]

Let's take a look at the data section: jumping to `.data` puts us at the beginning of it, here we can find the format string.

The array starts 4 byte after that (but is accessed in reverse order). After selecting the first element in IDA you can jump to the next element with *g* `+8` (and so forth).

```
.data:0000000000401003 db '#',0 ; first element
.data:0000000000401005 db    0
.data:0000000000401006 db    0
.data:0000000000401007 db    0
.data:0000000000401008 db    0
.data:0000000000401009 db    0
.data:000000000040100A db    0
.data:000000000040100B db  75h ; second element
.data:000000000040100C db    0
.data:000000000040100D db    0
.data:000000000040100E db    0
; ...
```

Code 4: beginning of array

Here is how I came to that conclusion: the addresses (in the same order as accessed by the loop above) of all the elements of the array are the following - via Python

```
for rcx in range (10, 0, -1):  print(hex(0x400ffb + rcx * 8))
```

10. `0x40104b` (our `DEADBEAFh`)
9. `0x401043` (`433A34h`)
8. `0x40103b` (`2213h`)
7. `0x401033` (`5h`)
6. `0x40102b` (`100h`)
5. `0x401023` (`443h`)
4. `0x40101b` (`ABCh`)

3. `0x401013` (`111h`)
2. `0x40100b` (`75h`)
1. **`0x401003`** (`23h`)

The counter variable starts at the bottom (`Ah` so `10` as a multiplier) and is subsequently decremented by 1. Since the loop is broken when `rcx` reaches zero the lowest `rcx` value to reach the instruction at `40301Bh` is `1`.

The first element of the array is thus: `400FFBh + 1 * 8` which equals **`401003h`**. Each element has 8 Bytes (per the multiplier above).

1.4 How many initialized global variables does this program define?

Initialized global variables are located in the `.data` section. [2]. It starts at `0401000h` and ends at `402000h`.

So far we have already encountered two:

- the format string starting at `401003h`
- the array starting at `401003h` (I'm going to count this as one)

They are joined by another two: quad-words at `401053h` and `40105Bh` (see Figure 4)

```
.data:0000000000401046                  db    0
.data:0000000000401047                  db    0
.data:0000000000401048                  db    0
.data:0000000000401049                  db    0
.data:000000000040104A                  db    0
.data:000000000040104B                  db 0AFh ; ¯
.data:000000000040104C                  db 0BEh ; ¾
.data:000000000040104D                  db 0ADh ; -
.data:000000000040104E                  db 0DEh ; Þ
.data:000000000040104F                  db    0
.data:0000000000401050                  db    0
.data:0000000000401051                  db    0
.data:0000000000401052                  db    0
.data:0000000000401053 qword_401053     dq 0FFFFFFFFFFFFFFFFh     ; DATA XREF: start+3A↓r
.data:0000000000401053                                            ; start+46↓w
.data:000000000040105B qword_40105B     dq 0                      ; DATA XREF: start+50↓w
.data:000000000040105B                                            ; start+60↓r
.data:0000000000401063                  align 1000h
.data:0000000000401063 _data            ends
```

Figure 4: end of the array and the two quad-word (followed by no more discernible variables)

The comment shows us where the two new quad-words are written and read from.

So **four** total.

1.5 What happens if the value “433000” is passed to the program through the command line?

Passing the value `433000` to `2010303027_hw_1_exercise_1` results in the value `433A34` (see Figure 5).

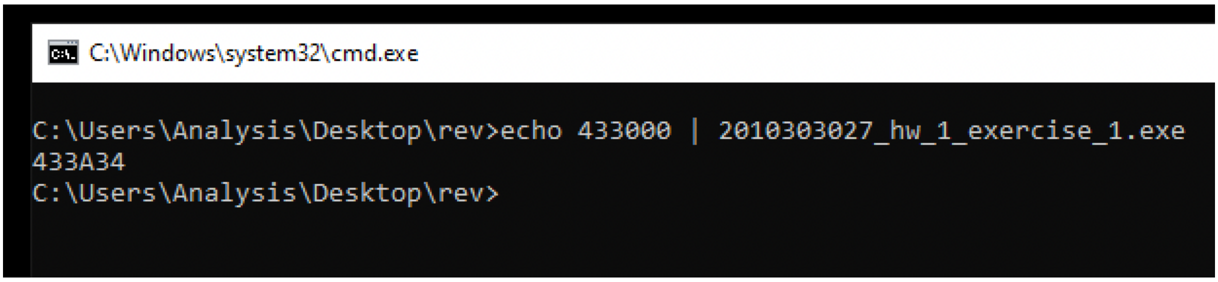


Figure 5: passing the value `433000` to `2010303027_hw_1_exercise_1.exe`

I’ve also written a small Batch script that passes a range of values to possibly get a better understanding of the program without too much work: Code 5.

```
@echo off
for /L %%G in (0, 1, 20000000) do (
  echo|set /p="%%G,␣␣"
  echo %%G | C:\Users\ben\Desktop\2010303027\2010303027_hw_1_exercise_1.exe
  echo.
)
```

Code 5: fuzz.bat

Here is an excerpt of the output: Code 6.

```
0,  5
1,  5
# ...
13,  5
14,  5
15,  23
# ...
98,  75
99,  75
100,  111
101,  111
```

```
# ...
```

Code 6: output via ‘fuzz.bat > log.txt‘

It does give us a couple of input-value borders where the output value changes but is not particularly fast so I let it run in the background for a bit.

Here is a list of values that I extracted from the log (on Linux via `sort -ut, -k2 log.txt`) before aborting the script. The left value is the first input value that results in a new output (right) value:

- 0, 5
- 15, 23
- 50, 75
- 100, 111
- 300, 443
- 780, ABC
- 1000, 1000
- 1910, 2213
- 220000, 433A34

Manually trying `-1` and random high values resulted in `DEADBEAF`, which are all (possibly) hex values. We already know all these values from our array!

1.6 What does this program do? Describe it using your own words.

It reads input from the users and prints the closest value from an array.

Here's a more detailed outline (reference figure 3 for the loop):

- read **`user input`** as hexadecimal via `stdint` with `scanf()` (using format string `%X`)
- set **`counter`** register (`rcx`) to 10
- write **`user input`** to `402000h` in the `.bss` section (a quad-word)
- for each **`array element`** (from 10^{th} to 1^{st}):
  - compare **`user input`** with **`array element`**:
    - if **`user input`** < **`array element`**: jump past switch
    - else: switch both values (simplifies the next step)
  - calculate absolute **`delta`**
  - compare **`delta`** with **`smallest known difference`** (that starts with max value):
    - if **`delta`** is lower update the **`smallest known difference`**
    - if **`delta`** is lower update the **`closest known element`**
  - decrement **`counter`**
- print the **`closest known element`** with `printf` (using the same format string)
- exit the program

The two quad-words we've seen in task 1.4 are used to store the `smallest known difference` (at `401053h`) and the `closest known element` (at `40105Bh`).

1.7 Write a program that carries out the same task using any programming language you know

```python
#!/usr/bin/env python3

import sys


def get_closest_element():
    """
    this program prints the closest value from an array
    when compared with the userinput.
    """
    array = [
        0x23,
        0x75,
        0x111,
        0xABC,
        0x443,
        0x100,
        0x5,
        0x2213,
        0x433A34,
        0xDEADBEAF,
    ]

    user_input = int(input(), 16)
    smallest_difference = sys.maxsize
    closest_element = None

    # iterate through reversed array:
    for element in array[::-1]:
        # calculate current absolute delta.
        delta = abs(user_input - element)
        # update if we're closer:
        if delta < smallest_difference:
            smallest_difference = delta
            closest_element = element

    print(hex(closest_element))

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

Code 7: finding the closest element with Python

2 Reverse Engineering Exercise 2: Crack me

[Binary Provided: `2010303027_hw_1_exercise_2.exe`]

2.1 Fingerprint the program using any of the hashing tools that were presented during the lecture.

Here's a small report from HashMyfiles:

- Filename: `2010303027_hw_1_exercise_2.exe`
- MD5: `456809322ef371ca13c40b2626baa6a5`
- SHA1: `57fb1a64eb6099da090b063bd3e0e140f66cfa05`
- CRC32: `f6093492`

SHA-256:
`fc3a851c9958f5cafc9b04ffa4afae346c4633355e0d2d6f8a6b0f2292cef58e`

2.2 Run the program and enter an input value.

- What input value did you use?

`123`

- What did you receive as an output?

```
****
****
***:
*#*@
```

Code 8: output pattern for input ‘123‘

I've continued to try values until I found the following pattern:

- `0`: `**`
- `1`: `*:`
- `2`: `*#`

- `3: *@`
- `4: :*`
- `5: ::`
- etc. until `Fh` which is `@@`

These can be represented as Nibbles.

Assumption: break it down further: `0` is `0b0000` so one `0b00` could represents a single `*` If we compare it with `1` which is `0b0001` this could mean `:` is `0b0001`. Let's create a mapping and compare it with the values we know from our previous experiment.

- `00: *`
- `01: :`
- `10: #`
- `11: @`

This mapping checks out.

2.3 What's the address of the program's entry point?

`403000h`

This is set in the PE header. In IDA you can load it by:

- loading the binary with the `Manual load` option
- answering yes to `Load the file header?` dialogue

Now you can jump to the header (*g* `HEADER:0`) and look for the address of the entry point.

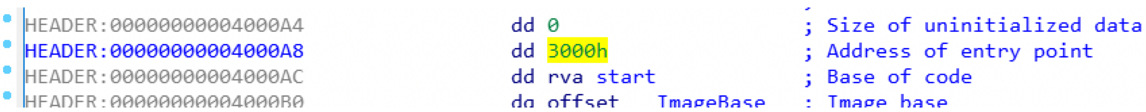


Figure 6: header of the

Adding this to the `image base` results in an address that is indeed the `start` of our program, see Figure 7.

```
.text:0000000000403000
.text:0000000000403000 ; =============== S U B R O U T I N E
.text:0000000000403000
.text:0000000000403000 ; Attributes: noreturn
.text:0000000000403000
.text:0000000000403000                 public start
.text:0000000000403000 start           proc near
.text:0000000000403000                 lea     rcx, Format
.text:0000000000403007                 xor     rax, rax
```

Figure 7: the entry point of the second program

2.4 Identify the strings present in this program.

- Use one of the tools installed in the virtual analysis environment.

I'm really taking a shine to IDA so let's stick with that. There is a strings sub view, see Figure 8.

Address	Length	Type	String
.data:0000000000401000	00000011	C	Enter a value:\r\n
.data:0000000000401031	0000002E	C	The following secret diagram was generated:\r\n
.data:000000000040105F	00000019	C	@*##\r\n@@@#\r\n#@##\r\n#@@#\r\n
.data:0000000000401078	00000019	C	@:@#\r\n##@:\r\n#@@#\r\n@#@@\r\n
.data:0000000000401091	00000019	C	@:@#\r\n##@:\r\n#@@#\r\n##@@\r\n
.data:00000000004010AA	00000019	C	@@@#\r\n@#@:\r\n#@@#\r\n@#@@\r\n
.data:00000000004010C3	00000019	C	##@*\r\n@##@\r\n@:@#\r\n@#@@\r\n
.data:00000000004010DC	00000019	C	@@@#\r\n@#@:\r\n#@@#\r\n@#@@\r\n
.data:00000000004010F5	00000019	C	@#@#\r\n@:@#\r\n@#::\r\n:*#:\r\n
.data:000000000040110E	00000019	C	#@@*\r\n@:@#\r\n@@##\r\n##@@\r\n
.data:0000000000401127	00000019	C	#*:#\r\n#*@:\r\n@#@#\r\n@@@@\r\n
.data:0000000000401140	00000019	C	@*##\r\n@:@:\r\n::::\r\n#@##\r\n
.data:0000000000401159	00000019	C	#@@*\r\n@:@#\r\n###@\r\n@*@:\r\n
.data:0000000000401172	00000019	C	@#@:\r\n#@@#\r\n::*@\r\n*#*#\r\n
.data:000000000040118B	00000019	C	#:::\r\n*#*:\r\n:*:@\r\n*@:#\r\n
.data:00000000004011A4	00000019	C	:@:*\r\n*:::\r\n#::#\r\n*@*#\r\n
.data:00000000004011BD	00000019	C	*@::\r\n:@*:\r\n::#:\r\n:*:#\r\n
.data:00000000004011D6	00000019	C	#@@#\r\n@#@@\r\n@@##\r\n@*@#\r\n
.data:00000000004011EF	00000011	C	Correct value!\r\n
.data:0000000000401200	00000011	C	Invalid value!\r\n

Figure 8: string sub view in IDA

There are a couple missing (like the DOS warning). Let's try it with `BinText`:

- load the binary
- include linefeed and carriage return in the `Filter` tab
  - to include strings from the secret diagram
- press `GO`

```
00000000004D    0000C000008D    0    !This program cannot be run in DOS mode.
000000000188    000000000127    0    .data
0000000001D8    000000000177    0    .text
000000000200    00000000019F    0    .idata
# [...]
```

Code 9: more strings

Indeed a couple more that IDA hid from us!

2.5 What sections are present in this program, what are their names and where are they located when the application is loaded into memory?

- Provide an address for each section

IDA provides a sub view for sections as well: see Figure 9.

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class
.data	0000000000401000	0000000000402000	R	W	.	.	L	para	0001	public	DATA
.bss	0000000000402000	0000000000403000	R	W	.	.	L	para	0002	public	BSS
.text	0000000000403000	0000000000404000	R	.	X	.	L	para	0003	public	CODE
.idata	0000000000404000	0000000000405000	R	.	.	.	L	para	0004	public	DATA

Figure 9: sections sub view in IDA

For the sake of readability, here are the section names and their respective start addresses:

- `.data`: `401000h`
- `.bss`: `402000h`
- `.text`: `403000h`
- `.idata`: `404000h`

2.6 Identify all symbols that are imported by the program and their import directory (library)!

- Make sure you list all libraries that are linked to each import symbol and the address of each symbol
- Use a tool in the virtual analysis environment to make the search easier

Address	Ordinal	Name	Library
0000000000404068		ExitProcess	kernel32
00000000004040A0		printf	msvcrt
00000000004040A8		scanf	msvcrt

Figure 10: imports sub view in IDA

2.7 Identify all exported symbols and their addresses

Name	Address	Ordinal
start	0000000000403000	[main entry]

Figure 11: exports sub view in IDA

2.8 What calling convention is used by the following functions:

- `printf`
  - **x64 calling convention**
    - first parameter via `rcx`
    - similar to `__fastcall` but up to 4 instead of 2 registers
      - `rcx`, `rdx`, `r8`, `r9`
      - after that via the stack
- `scanf`
  - **x64 calling convention**
    - two parameters: `rcx` and `rdx`
- `sub_4030C7` (location `0x4030C7`)
  - **`__cdecl`**
    - cleaned up by caller (`add rsp, 10h`)
- `sub_403139` (location `0x403139`)
  - **`__stdcall`**
    - pushed to the stack: `401127h` and `402000h`
    - cleaned up by callee (`retn 10h`)

```
.text:000000000040303C call    cs:scanf                    ;    x64 calling convention
.text:0000000000403042 mov     rax, cs:qword_402017
.text:0000000000403049 push    rax
.text:000000000040304A push    402000h
.text:000000000040304F call    sub_4030C7                  ; __cdecl
.text:0000000000403054 add     rsp, 10h
.text:0000000000403058 mov     rcx, offset byte_402000     ; Format
.text:000000000040305F xor     rax, rax
.text:0000000000403062 call    cs:printf                   ; x64 calling convention
.text:0000000000403068 push    401127h
.text:000000000040306D push    402000h
.text:0000000000403072 call    sub_403139                  ; __stdcall
.text:0000000000403077 test    rax, rax
```

Figure 12: calling conventions

I've used Compile Explorer to play around with the different calling conventions with both x64 and x86 version of the `msvc` compiler: https://godbolt.org/z/TcsToh.

2.9 Explain in your own words. What does this function do?

- `sub_4030A7` (location `0x4030A7`)

This function takes two bytes as input (via `rbx`) and returns a mapped character as an output (via `rax`).

Possible input/output mappings:

- `00 -> *`
- `01 -> :`
- `10 -> #`
- `11 -> @`

Detailed order of operations (see figure 13):

This function is called in `sub_4030C7` via `.text:004030F5 call sub_4030A7`. Before the call the following happens:

- `rbx` is nulled out everywhere but the 2 least significant bits (`and rbx, 3`)
- `rbx` is pushed to the stack as parameter for the call

If we assume that the user input is `ABCD1234`, the two least significant bits of `Ah` (`0b1010`) are `0b10`. This is the argument for the first time the function is called.

In the function the following happens:

- `rbx` is pushed to the stack and popped to the lower `ebx`
- `rax` is set to `@` (`40h`, as a default value)
- the lowest byte of `ebx` is written written back, the upper filled with zeros
- `ebx` is compared with `0b11`:
  - if `ebx` is not bigger than `3`: use it as array lookup index and write value to `rax`
  - else: jump past array lookup (keeping the default value for `rax`)
- restore `ebp`
- exit

After the function the rest of the program can work with the return value (stored in `rax`).

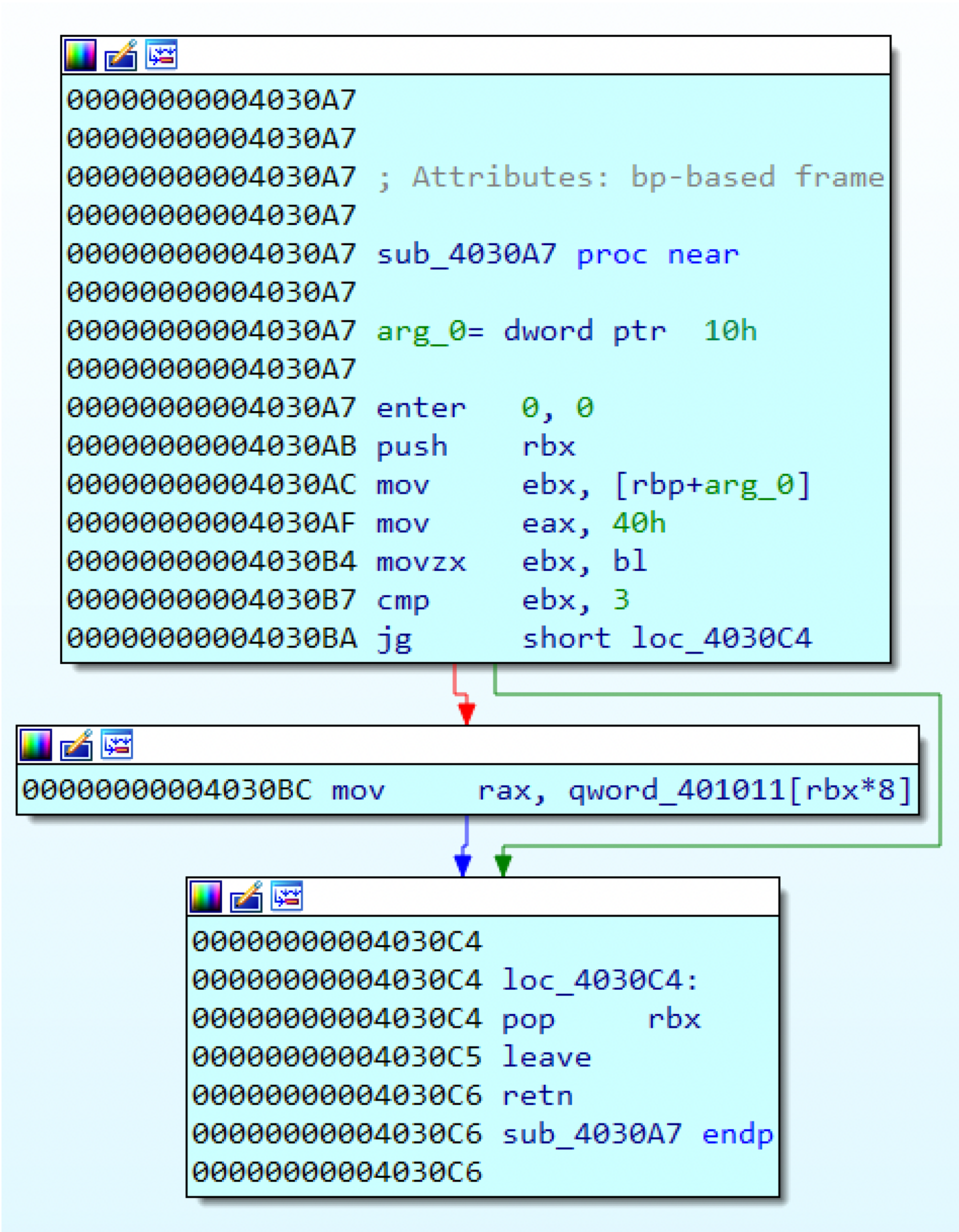


Figure 13: the function `sub_4030A7`

2.10 Figure out what code needs to be entered as an input in the console to match the diagram that is displayed. How did you find out?

- After the correct input has been provided, the following message should appear “Correct value!”

The pattern:

```
#*:#
#*@:
@#@#
@@@@
```

Code 10: target secret pattern

Using the mapping from task 2.2 (plus the confirmation from task 2.9) and a bit of help from `calc.exe` we get: `868DEEFFh`. See Figure 14 and Figure 15.

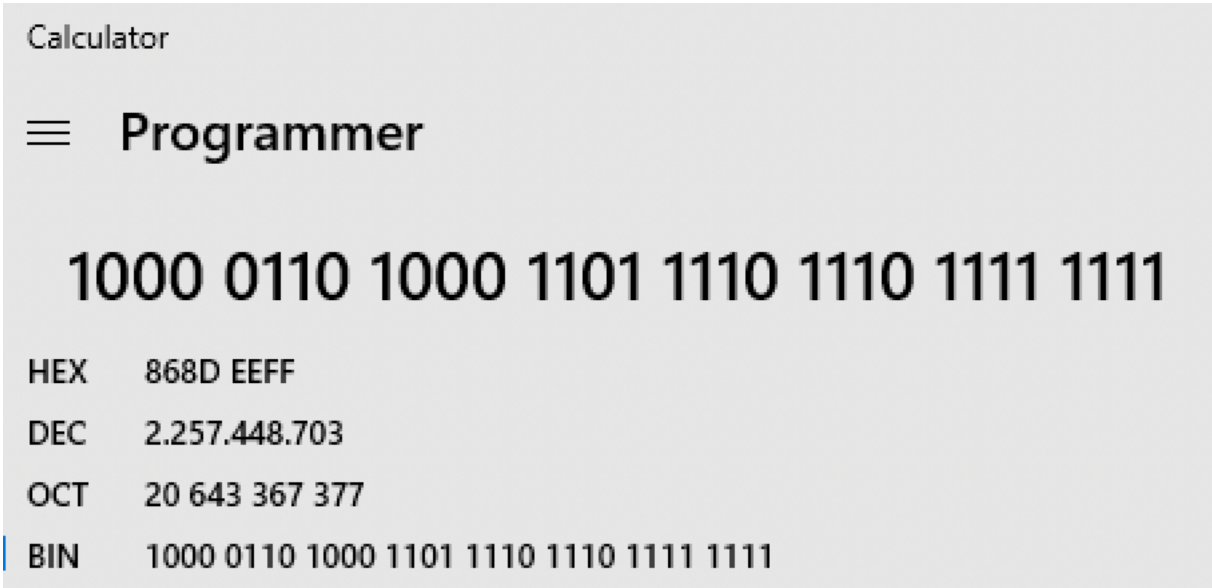


Figure 14: converting the key from binary to hex

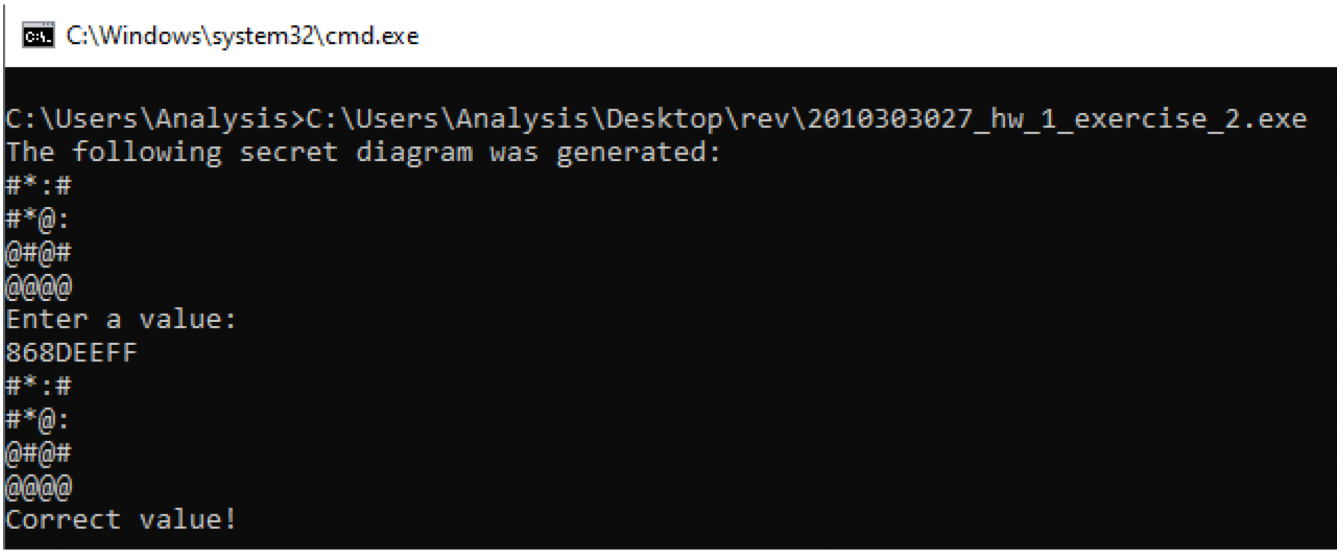


Figure 15: checking the secret key

3 Reverse Engineering Exercise 3: Malware Analysis

Imagine you are an analyst in an anti-virus company and given the malware sample in the archive. Users report that they get blackmailed to transfer some amount of bitcoins to an evil person, otherwise access to the files will be denied forever.
Here is some useful information:

- The main function of this program starts at `0x4013F4`. This is the perfect starting point for your debugging session (i.e. set a breakpoint at that location with your favorite debugger).
- This executable does actually encrypt files on the disk. So please execute with care and take a snapshot before working on it

3.1 Anti-Analysis Protection

The piece of malware has anti-analysis protection. What is done to prevent malicious code execution in your VirtualBox environment? Please explain in detail and describe how you bypassed the check.

The binary tries to detect whether it is called from within Virtual Box by checking for a DLL that ships with the Virtual Box Guest Additions: `C:\windows\syswow64\vboxnine-x86.dll`.

See Figure 16. This code does the following:

- load `Shlwapi.dll` via `LoadLibraryA` so we can use its `PathFileExistsA`
- use `PathFileExistsA` to check for `C:\windows\syswow64\vboxnine-x86.dll`
  - it returns `true` if the file exists
  - it returns `false` if not
- if it does not exist jump past setting `byte_405638` to `1` and calling `sub_4012C4`
  - the variable keeps track of if we are in a VM/debugger
  - the subroutine is used to generate an exception in the debugger
    - by trying to write to `0x0` in a new thread (See Code 17.)
      - jumping to `mov byte ptr [r14], 1` while `r14` points to `0x0`
      - which is written in to the process memory of the thread
      - which throws the exception when created
    - this code is reused by all the other checks (as a consequence)
- so **if we are in a VM (the file exists) the crash code is executed** to stop further analysis

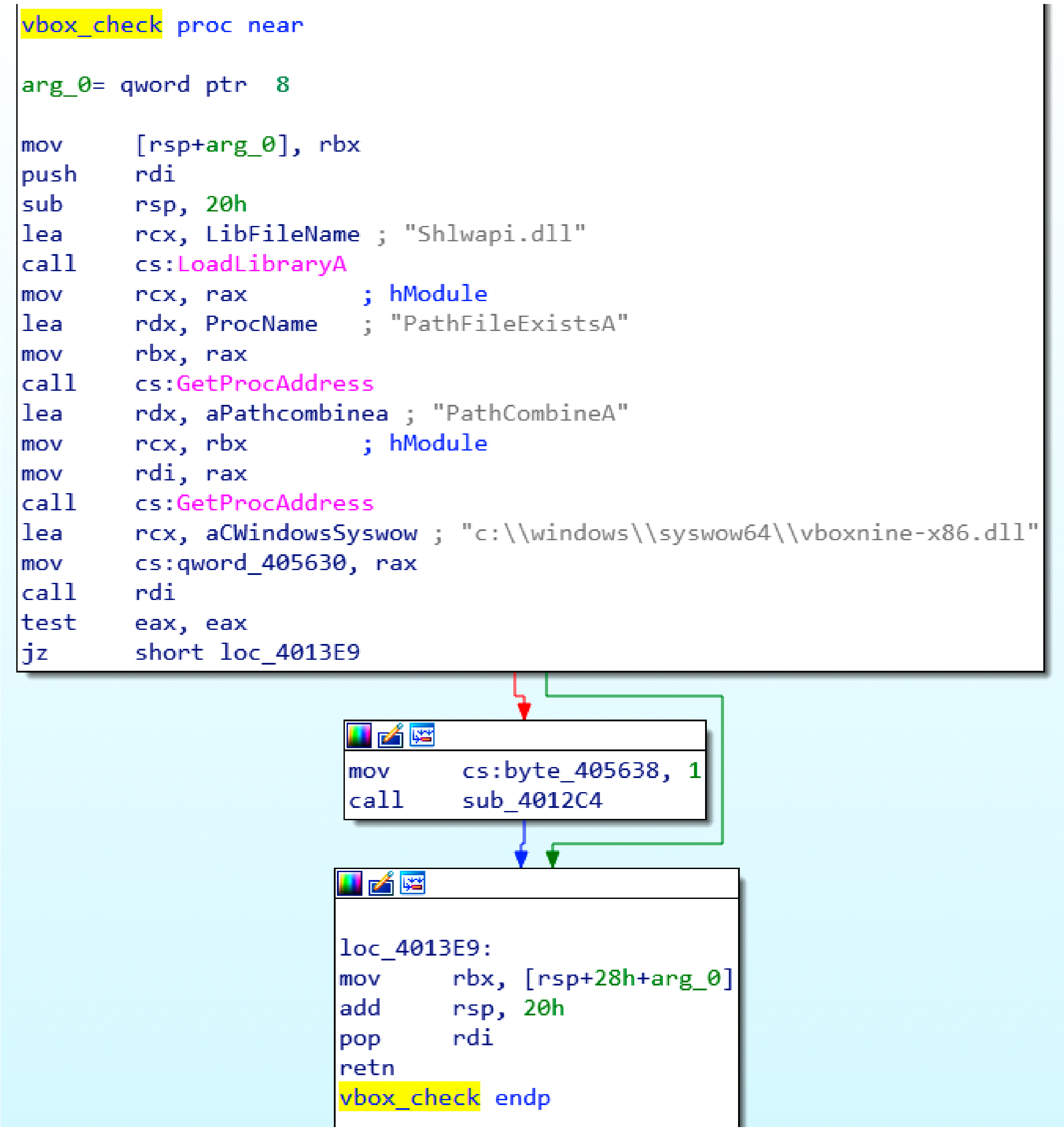


Figure 16: the check that looks for the VBox DLL

```
xor     rax, rsp
mov     [rsp+58h+var_10], rax
call    cs:GetCurrentProcessId
xor     edx, edx        ; bInheritHandle
mov     ecx, 1FFFFFh    ; dwDesiredAccess
mov     r8d, eax        ; dwProcessId
call    cs:OpenProcess
mov     rdi, rax
cmp     rax, 0FFFFFFFFFFFFFFFFh
jz      short loc_401373
```

```
xor     edx, edx            ; lpAddress
mov     [rsp+58h+flProtect], 40h ; lpParameter
mov     rax, 106C641F63345h
mov     r9d, 3000h          ; flAllocationType
mov     rcx, rdi            ; hProcess
mov     [rsp+58h+Buffer], rax
lea     r8d, [rdx+8]        ; dwSize
call    cs:VirtualAllocEx
and     qword ptr [rsp+58h+flProtect], 0
lea     r8, [rsp+58h+Buffer] ; lpBuffer
mov     rdx, rax            ; lpBaseAddress
mov     r9d, 8              ; nSize
mov     rcx, rdi            ; hProcess
mov     rbx, rax
call    cs:WriteProcessMemory
and     [rsp+58h+var_28], 0
mov     r9, rbx             ; lpStartAddress
and     [rsp+58h+var_30], 0
xor     r8d, r8d            ; dwStackSize
and     qword ptr [rsp+58h+flProtect], 0
xor     edx, edx            ; lpThreadAttributes
mov     rcx, rdi            ; hProcess
call    cs:CreateRemoteThread
mov     rcx, rdi            ; hObject
call    cs:CloseHandle
```

Figure 17: the code that ultimately throws an exception

To bypass this (and other) checks we have a couple of options, among them:

- dirty hack: uninstall the Guest Additions or delete/rename the DLL
- patch: change the string to a non-existing file
- patch: change the conditional jump for this check
- register manipulation: keep changing the zero flag to get past the crash code
- patch: **jump past the exception code in `sub_4012C4`** (the crash code)
  - this allows us to work around all checks with a single patch
  - the checks will still trigger but they won't throw an exception
  - I've chosen this option

I've patched the instruction at `0x4012FA` from `je 0x401373` to `jmp 0x401373`. This effectively **always** jumps past the crash code allowing us to use the binary both inside and outside a VM and/or debugger.

I think this is a rather clean solution but I've also mentioned possible other patches for all further checks.

Here is a binary diff via Radare2 that shows the entirety of the patch I've ended up with: Code 11.

```
ben::cyric:shared:$ radiff2 -D Crypter_A.exe Crypter_A_betterpatch.exe
--- 0x000006fa  74
- je 0x79
- xor edx, edx
+++ 0x000006fa  eb
+ jmp 0x7b
+ xor edx, edx
```

Code 11: binary diff

3.2 Three More Protections Against Analysis

Find three other ways that the executable tries to protect itself from manual analysis. Please explain them in detail and describe how you bypassed the protection measures.

- **delta time check** via `GetTickCount64()`s
  - this check actually starts before the check from the previous task
    - the current tick count is saved (`0x401409` and `0x40140`)

        * the first check is executed (`0x401412`)
        * the current tick count is saved again (`0x401417`)
        * the delta of the two counts is calculated and compared with 1000 ms (`0x40141D` to `0x401431`)
    - if it took longer (as when carefully stepping through the program) the check triggers
    - if you don't manually step through the program you don't have to patch this at all (it will only trigger if you take more than 1 second between the two calls to `GetTickCount64()`)
    - possible patch: increase the 1000 ms (`0x3E8`) to a higher value or patch the jump
- **`CheckRemoteDebuggerPresent()`**
    - called at `0x401455`
    - this checks if the process is attached to a debugger
    - the return value is tested for zero `0x40145B`
    - `0x40145D` jumps if it was non-zero (to the aforementioned crash code)
    - in conclusion: crash if the process is attached to a debugger
    - possible patch: nop out the call (`rax` is zero before it)
- I'm not sure what I would classify as the third additional countermeasure. Maybe the usage of the variable **`byte_405638`** to keep track of the outcome of the checks. All these checks are made inconsequential by my crash code patch. Maybe stack cookies? But those are not really a hindrance since we're not writing a buffer overflow and have full control over them (and they are more of a compiler feature anyway). Or the `IsProcessorFeaturePresent` fastfail-check before the main (but that never gets executed and does not prevent analysis). Some dead code? I did not find a noticeable amount that would prevent analysis.
- Or maybe it's the fact that there is no Bitcoin address in sight, even though people are supposed to send some to get the key. That would certainly stump me as an analyst ;)
- I've spent so much time thinking about what the third protection mechanism might be; maybe it's the task itself. Very meta.

3.3 Persistence

Does the executable make itself persistent in the system? If so, please explain how.

There seems to be no persistence.

I could not find any persistence, neither by looking in the binary nor by fetching all processes with `tasklist` and diffing them with a known clean state. I've also checked in the common places (registry, etc.) without avail.

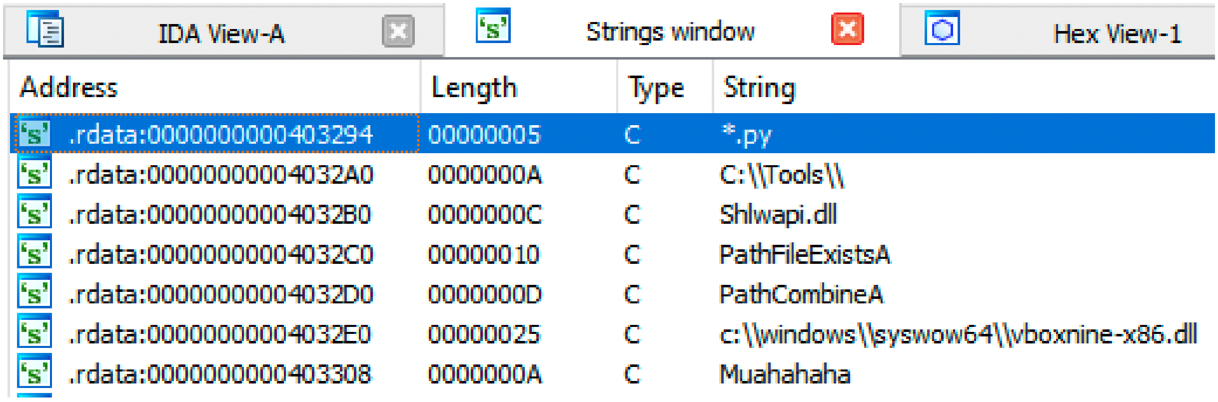
The binary does not seem to be set up for persistence either (e.g. by looking for new files and encrypting them as they are added). Simply running the encryption on startup again would not make any sense either, since the encryption function is its own inverse function.

3.4 Which Files Are Encrypted?

Which files are encrypted? Please explain how you figured that out.

All the Python (`.py`) files in the `C:\Tools` folder. (Incidentally the file type I would least want to lose in a ransomware attack.)

I have figured this out early on since I usually take a look at all strings in a binary as one of the first steps. Here the globbing pattern and the path caught my eye. See Figure 18.



Address	Length	Type	String
.rdata:0000000000403294	00000005	C	*.py
.rdata:00000000004032A0	0000000A	C	C:\\Tools\\
.rdata:00000000004032B0	0000000C	C	Shlwapi.dll
.rdata:00000000004032C0	00000010	C	PathFileExistsA
.rdata:00000000004032D0	0000000D	C	PathCombineA
.rdata:00000000004032E0	00000025	C	c:\\windows\\syswow64\\vboxnine-x86.dll
.rdata:0000000000403308	0000000A	C	Muahahaha

Figure 18: partial list of strings in malware

Looking for the occurrences of these two strings (via xrefs) quickly lead me to the piece of code that iterates over all files and directories in `C:\Tools` and the extension check (for Python files), which confirmed my suspicion. See Figure 19.

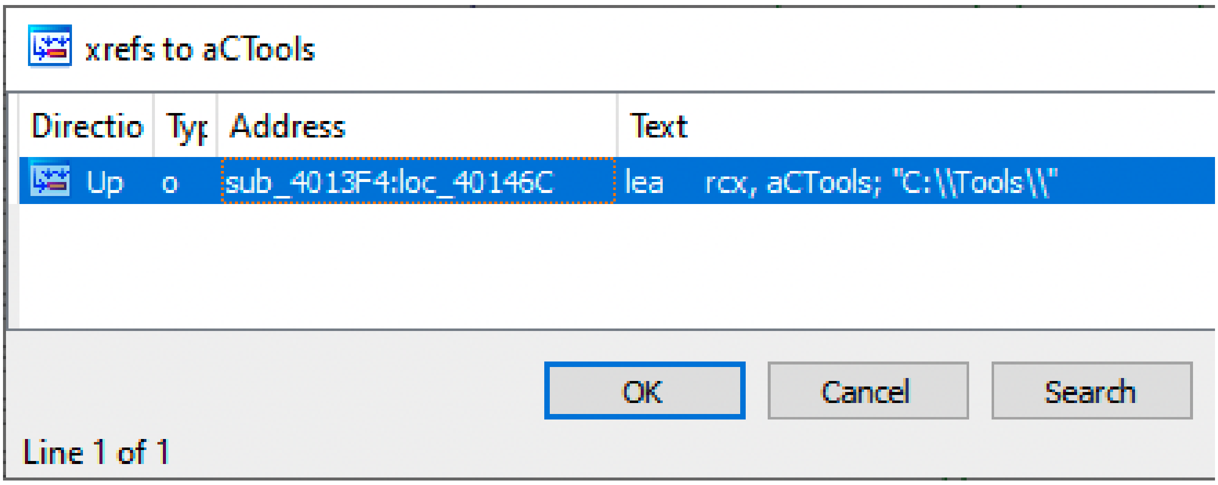


Figure 19: cross references for the `C:\Tools` variable

After patching the binary and running it I had further confirmations since all Python files in the Tools directory were indeed encrypted (and decrypted if run again).

3.5 Parameters for `CreateFileA`

> There are two different calls to CreateFileA. Besides the filename parameter, please explain the parameters given to those two calls (value and meaning in your own words).

- the first call at `0x4010BB`
  - a pointer to the string of a python filename as `lpFileName`
  - `0x80000000` for `dwDesiredAccess`: **`GENERIC_READ`**
    - get read access to resource (Python file in our case)
    - to read the content (so it can xor/encrypt it)
  - `0x3` for `dwShareMode`: **`FILE_SHARE_READ`** plus **`FILE_SHARE_WRITE`**
    - to allow both read and write access rights to the file by other processes
  - `0x0` for `lpSecurityAttributes`, so **`NULL`**
    - the file handle can't be passed to a child processes
    - and the file gets a default security descriptor
  - `0x3` for `dwCreationDisposition` **`OPEN_EXISTING`**
    - this mode will throw an error if the file does not exist as opposed to silently creating it (**`ERROR_FILE_NOT_FOUND`**)
  - `0x8000000` for `dwFlagsAndAttributes`: **`FILE_FLAG_SEQUENTIAL_SCAN`**
    - used for sequential file access from beginning to end
    - this is used as a hint to Windows which helps it with optimizing caching behaviour
      - for example: no need to keep already read content in cache (because we're only moving forward)
  - for `hTemplateFile`
    - this is ignored because we're opening an existing file
  - if everything went allright we have a read handle now
- the second call at `0x4010F3`
  - this is executed if we got a valid (read) file handle from the previous call
  - same filename as the call preceding it
  - `0x40000000` for `dwDesiredAccess`: **`GENERIC_WRITE`**
    - now the malware actually wants to write the encrypted content
  - all the other parameters are the same as for the previous call
  - if everything worked we now have a valid write handle as well

3.6 The Encryption Method

> Which encryption method is used? Please explain the piece of code responsible for it.

The encryption method boils down to a bytewise XOR.

The program iterates through all files and folders (skipping the current folder `.` and the parent folder `..`) looking for all python files along the way. If one is found the absolute path to that file is concatenated and the address that points to that string is loaded into `rcx`.

Before the real encryption starts a bit of preparation happens, starting at `0x401000`:

- the encryption key is calculated and stored in `rsi` (see next section)
- a read and write handle is created (see previous section)

Now a single byte is read (at `0x40114B`).

This single byte is `xor`ed with the calculated key at `0x40110E`:
`xor [rsp+160h+var_120], sil`.

This new encrypted byte is written back (in place) to the file at `0x40112C`:
`call cs:WriteFile`.

This repeats until the entire file is encrypted (and thus no more bytes can be read).

3.7 The Encryption Key

> What's the encryption key being used for encryption? How is it calculated?

The encryption key is 68, which is `0x44` or `0b1000100`.

The key is based on the username of the executing user which is fetched via `call cs:GetUserNameA`.

The following is an algorithm of how it is derived:

- get the username: `Analysis` (at `0x401040`)
- count chars in username string: `8` (1-indexed, `0x40104F` to `0x401055`)
  - by `inc`ing until we hit the string terminator

- add up all individual ASCII values (using our counter, `0x401064` to `0x401075`):
  - A: `0x41`
  - n: `0x6E`
  - a: `0x61`
  - l: `0x6C`
  - y: `0x79`
  - s: `0x73`
  - i: `0x69`
  - s: `0x73`
- which totals `0x344` in our case
- `AND` this value with `0x800000FF` (mask out, at `0x401077`)
  - keep only the leftmost (sign) and the 8 rightmost bits
- which gives us `0x44` our key!

The same in Python:

```
sum = 0
for c in 'Analysis': sum += ord(c)
hex(sum & 0x800000FF)  # '0x44'
```

Code 12: key_calculator.py

3.8 Writing a Decryptor

> Share some piece of pseudo-code for a decryption tool that could be distributed to customers.

I've opted for Python instead of pseudo-code so I could test it and make sure not to forget any steps (plus it's fun to write). I've used Python 2 instead of 3 because that's the one that was available on the Windows box. See Code 13.

This piece of code assumes that the customer has created a backup before running the decryptor in case anything goes wrong (and we want to build that habit for the future).

```
#!/usr/bin/env python2
import os

path = 'C:\Tools'
filetype = '.py'
key = 0x44


def decrypt(filename):
    print 'decrypting ', filename
```

```
    decrypted = ''

    # read and decrypt content of file:
    with open(filename, 'rb') as file:
        for byte in file.read():
            decrypted += chr(ord(byte) ^ key)  # xor each byte with key.

    # write back decrypted content:
    with open(filename, 'wb') as file:
        file.writelines(decrypted)


# recursively iterate through path:
for subdir, dirs, files in os.walk(path):
    for filename in files:

        # decrypt affected files:
        if filename.endswith(filetype):
            decrypt(subdir + os.sep + filename)
```

Code 13: decryptor.py

Bibliography

[1] ANDRIESSE, D. and S. FRANCISCO: *PRACTICAL BINARY ANALYSIS Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly*. Techn. Rep., 2019.

[2] HOEY, J. V.: *Beginning x64 Assembly Programming: From Novice to AVX Professional Paperback*. Apress, 1 ed., 2019.

[3] INTEL: *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes*. System, 3(253665), 2011.

List of Figures

List of Code