



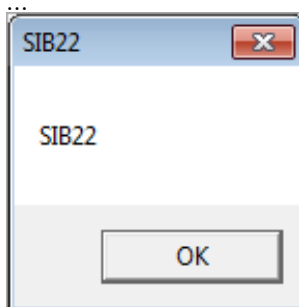
Reverse Engineering (REV3)

## UE 02 – Statische Analyse – Protokoll

Jakob Mayr

WS 2023/2024

### 1 Einleitung



## 2 Verschlüsselung/Komprimierung

### 2.1 hello1.exe

Das PE-File "hello1.exe" verwendet für die "Entschlüsselung" eine XOR-Operation über einen Adressbereich im Datensegment.

Das EBX-Register stellt für die "Entschlüsselung" den Zähler dar (Adressbereich wird von größter zu kleinster Adresse durchiteriert). Die Startadresse (401017) + "Wert im EBX-Register" zeigt auf das jeweilige zu entschlüsselnde Byte und durch das dekrementieren des EBX-Registers werden all 33 Bytes "entschlüsselt".

Der "Schlüssel" für das XOR ist 0x96 (im ECX Register).

Codesnipped für XOR-Entschlüsselung:

```

00401000 $ BB 34000000 MOV EBX, 34
00401005 . B9 96000000 MOV ECX, 96
0040100A > 4B DEC EBX
0040100B . 318B 17104000 XOR DWORD PTR DS:[EBX+401017], ECX
00401011 . 85DB TEST EBX, EBX
00401013 . 75 F5 JNZ SHORT hello1.00401000
00401015 . EB 0C JMP SHORT hello1.00401023
00401017 . DE96 F396FA96 FICOM WORD PTR DS:[ESI+96FA96F3]
0040101D FA C1

```

Die Adresse für dieses Datensegment kann in "OllyDbg" via der Option "Follow in Dump/Selection" angezeigt werden.

"Hello"  $\oplus$  0x96 ergibt folgenden Wert:

de 96 f3 96 fa 96 fa 96 f9 96

Dieser Wert kann auch in einem Hex-Editor gefunden werden. Durch ersetzen mit "SIB22"  $\oplus$  0x96 kann ein neues File erstellt werden welches "SIB22" im Fenster anzeigt.

"Hello"  $\oplus$  0x96 ergibt folgenden Wert:

c5 96 df 96 d4 96 a4 96 a4 96

### 2.2 hello2.exe

Das zweite PE-File "hello2.exe" arbeitet sehr ähnlich wie "hello1.exe", mit dem Unterschied, dass im Adressbereich in die andere Richtung entschlüsselt wird (Adresse erhöht sich nun) und dass nicht Byte-weise sondern immer 4-Byte-weise oder jeweils ein DWORD (32-bit) entschlüsselt wird.

Codesnipped für XOR-Entschlüsselung:

```

00401000 $ 33DB XOR EBX, EBX
00401002 . B9 C756D451 MOV ECX, 51D456C7
00401007 > 318B 17104000 XOR DWORD PTR DS:[EBX+401017], ECX
0040100D . 83C3 04 ADD EBX, 4
00401010 . 83FB 34 CMP EBX, 34
00401013 . 75 F2 JNZ SHORT hello2.00401007
00401015 . EB 0C JMP SHORT hello2.00401023

```

Der Key hierbei muss natürlich auch 32-Bit haben:

C7 56 D4 51

"Hello" in verschlüsselter Form hat folgende Form:

8F 56 B1 51 AB 56 B8 51 A8 56

"SIB22" in verschlüsselter Form hätte hierbei folgende Form:

94 56 9d 51 85 56 e6 51 f5 56

#### Note

Hierbei sehr auffällig ist, dass das Padding der Buchstaben (0x00) sich mit dem Schlüssel deckt, deshalb wiederholt sich "0x56" und "0x51" immer wieder.

## 2.3 hello3.exe

Auch das PE-File "hello3.exe" arbeitet sehr ähnlich. Hierbei wird der Adressbereich jedoch zweimal "ge-xor'd". Einmal mit einem statischen Wert "0x69" und einmal mit dem Wert aus dem EBX-Register welches auch gleichzeitig wieder den Zähler darstellt. (Hierbei wird die Adresse wieder vermindert)

Einfach erklärt wird die das Byte an der höchsten Adresse wie folgt entschlüsselt:

"5B"  $\oplus$  0x69  $\oplus$  0x32

Das folgenden Bytes werden dann wie folgt entschlüsselt:

"18"  $\oplus$  0x69  $\oplus$  0x31

"79"  $\oplus$  0x69  $\oplus$  0x30

...

Da dies händlich ein wenig aufwendig ist, kann die auch via Python realisiert werden ():

```

1  def xor_process(hex_data):
2      xor_value = 0x69
3      decrementing_value = 0x32
4
5      result = []
6
7      for byte in hex_data:
8          # XOR with 0x69 and the decrementing value
9          xor_result = byte ^ xor_value ^ decrementing_value
10
11         print(f"input_byte: {hex(byte)}\ndecrementing_value: {hex(decrementing_value)}\
xor_result{hex(xor_result)}\ntascii: {chr(xor_result)}\n")
12         result.append(xor_result)
13
14         if decrementing_value == 0:
15             print("end of decrementing")
16             break
17         decrementing_value -= 1
18
19     return result
20
21 # Input data (hexadecimal bytes)
22 hex_str_hello = (
23     "BB 33 00 00 00 B9 69 00 00 00 03 C3 4B 31 8B 1F 10 40 00 31 9B 1F 10 40 00 85 DB 75
24     EF EB 0C 21 68 0E 6A 01 6C 03 6E 0E 60 63 62 0F 64 0F 79 69 38 7B 12 62 6C 3F 7E 1B
25     70 9B 7C 75 75 77 1C 49 A0 4A 4A 4D 4C 83 B1 64 40 63 02 45 BB 62 4E 79 18 5B"
26 )
27
28 hex_str_sib = (
29     "BB 33 00 00 00 B9 69 00 00 00 03 C3 4B 31 8B 1F 10 40 00 31 9B 1F 10 40 00 85 DB 75
30     EF EB 0C 3a 68 22 6A 2F 6C 5D 6E 53 60 63 62 0F 64 0F 79 69 38 7B 12 62 6C 3F 7E 1B
31     70 9B 7C 75 75 77 1C 49 A0 4A 4A 4D 4C 83 B1 64 40 63 02 45 BB 62 4E 79 18 5B"
32 )
33
34 print("----- Hello part -----")
35 hex_bytes_hello = [int(h, 16) for h in hex_str_hello.split()]
36 hex_bytes_hello = [int(h, 16) for h in hex_str_hello.split()]
37
38 processed_bytes_hello = xor_process(hex_bytes_hello[::-1])
39 processed_hex_hello = ['{:02X}'.format(b) for b in processed_bytes_hello]
40 print(' '.join(processed_hex_hello))
41
42 print("----- SIB part -----")
43
44 hex_bytes_sib = [int(h, 16) for h in hex_str_sib.split()]
45 hex_bytes_sib = [int(h, 16) for h in hex_str_sib.split()]
46
47 processed_bytes_sib = xor_process(hex_bytes_sib[::-1])
48 processed_hex_sib = ['{:02X}'.format(b) for b in processed_bytes_sib]
49 print(' '.join(processed_hex_sib))

```

## Codesnipped für XOR-Entschlüsselung:

```

00401000 $ BB 33000000 MOV EBX, 33
00401005 B9 69000000 MOV ECX, 69
0040100A 83C9 MOV EBX, ECX
0040100C > 4B DEC EBX
0040100E 3180 1F104000 XOR DWORD PTR DS:[EBX-40101F], ECX
00401013 319B 1F104000 XOR DWORD PTR DS:[EBX-40101F], ECX
00401015 75 FF JNZ SHORT hello3.0040100C
0040101D EB 0C JMP SHORT hello3.0040102B
0040101F 2180E6G DO 608E6221
00401023 016C836E DO 6E836C81
00401027 0FC08362 DO 6C83608E

```

## Note

Der Teil der hierbei in Python entschlüsselt wird ist natürlich länger als das eigentliche Python, es müssen wieder nur 5 Bytes geändert werden:

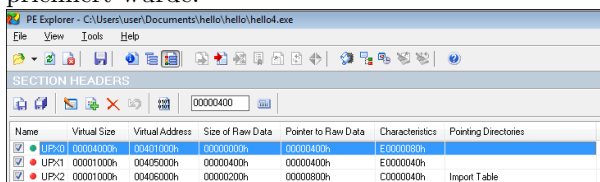
```

00 02 00 00 00 00 04 00 00 00
00 BB 33 00 00 00 B9 69 00 00 00
0C 21 68 0E 6A 01 6C 03 6E 0E 60
77 1C 49 A0 4A 4A 4D 4C 83 B1 64
00 00 00 00 00 00 00 00 00 00

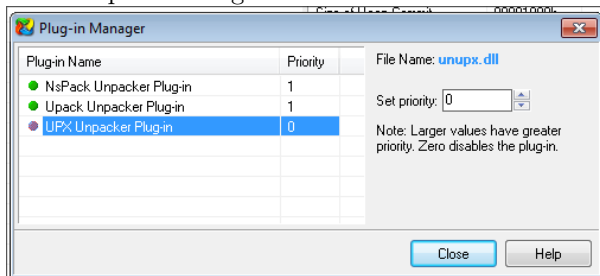
```

## 2.4 hello4.exe

Das "hello4.exe" PE-File verwendet keine Verschlüsselung sondern eine Komprimierung. Deaktiviert man das Plugin "UPX Unpacker Plug-in" im PE-Explorer und öffnet das File, so sieht man, dass die "Section-Headers" untypisch "UPX0", "UPX1" und "UPX2" heißen. Ebenfalls erkennt man z.b., dass die UPX0 eine "Size of Raw Data" von 0 hat, aber eine "Virtual Size" von 4000. Durch diese Dinge kann man bereits darauf schließen, dass das PE-File mit UPX (Ultimate Packer for Executables) komprimiert wurde.



## UPX Unpacker Plug-in:



Aktiviert man nun das Plugin, öffnet das File und speichert es "entpackt" (entpackt sich beim öffnen), dann hat man wieder ein "normales" PE-File. Anschließend kann in einem Hex-Editor der Hex-Wert für "Hello" (48 65 6c 6c 6f, bzw 48 00 65 00 6c 00 6c 00 6f 00) gesucht und durch den Hex-Wert für "SIB22" (53 49 42 32 32, bzw 53 00 49 00 42 00 32 00 32 00) ersetzt werden.

Folglich ein Screenshot der Stelle im Hex-Editor, welche "SIB22" zeigt.

```

00 00 00 00 00 00 00 00 00 00
00 53 00 49 00 42 00 32 00 32 00
00 00 00 00 00 00 00 00 00 00

```





### 3.4 hello4.exe

Da im PE-File "hello4.exe" keine Entschlüsselung wie in den Beispielen zuvor durchgeführt wird, kann etwas anderes angezeigt werden.

Das PE-File kann auch via CommandLine entpackt werden und anschließend kann in radare2 nach dem Hex-Wert für "Hello" gesucht werden und dieser ausgegeben werden:[1]

```
mendacium@mendacium ~/tmp$ r2 hello4_unpacked.exe
[0x004013e8]> /x 480065006c006c006f00
Searching 10 bytes in [0x403200-0x404000]
hits: 0
Searching 10 bytes in [0x403000-0x403200]
hits: 1
Searching 10 bytes in [0x402200-0x403000]
hits: 0
Searching 10 bytes in [0x402000-0x402200]
hits: 0
Searching 10 bytes in [0x401600-0x402000]
hits: 0
Searching 10 bytes in [0x401000-0x401600]
hits: 0
0x00403000 hit0_0 480065006c006c006f00
[0x004013e8]> s 0x403000
[0x00403000]> px
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x00403000 4800 6500 6c00 6c00 6f00 0000 0000 0000 H.e.l.l.o.....
0x00403010 0000 0000 0000 0000 0000 0000 0000 .....
0x00403020 0000 0000 0000 0000 0000 0000 0000 .....
0x00403030 0000 0000 0000 0000 0000 0000 0000 .....
0x00403040 0000 0000 0000 0000 0000 0000 0000 .....
0x00403050 0000 0000 0000 0000 0000 0000 0000 .....
0x00403060 0000 0000 0000 0000 0000 0000 0000 .....
0x00403070 0000 0000 0000 0000 0000 0000 0000 .....
0x00403080 0000 0000 0000 0000 0000 0000 0000 .....
0x00403090 0000 0000 0000 0000 0000 0000 0000 .....
0x004030a0 0000 0000 0000 0000 0000 0000 0000 .....
0x004030b0 0000 0000 0000 0000 0000 0000 0000 .....
0x004030c0 0000 0000 0000 0000 0000 0000 0000 .....
0x004030d0 0000 0000 0000 0000 0000 0000 0000 .....
0x004030e0 0000 0000 0000 0000 0000 0000 0000 .....
0x004030f0 0000 0000 0000 0000 0000 0000 0000 .....
[0x00403000]>
```

```
mendacium@mendacium ~/tmp$ upx -d hello4.exe -o hello4_unpacked.exe
Ultimate Packer for eXecutables
Copyright (C) 1996 - 2020
UPX 3.96      Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

-----
File size      Ratio      Format      Name
-----
3584 ←         2560      71.43%     win32/pe    hello4_unpacked.exe

Unpacked 1 file.
mendacium@mendacium ~/tmp$
```

Ebenfalls kann im komprimierten File der Einsprungspunkt und somit der Beginn der UPX-Entschlüsselung betrachtet werden:

```
r2 -c 'aa; s entry0; pd' hello4.exe
INFO: Analyze all flags starting with sym. and entry0 (aa)
INFO: Analyze all functions arguments/locals (afva@@@F)
393: entry0 ();
0x004050f0 60      pushal
0x004050f1 b0050400 mov esi, section.UPX1 ; 0x405000
0x004050f6 8db00c0fff lea edi, [esi - 0x4000]
0x004050fc 57      push edi
0x004050fd 83cfff   or ebp, 0xffffffff ; -1
0x00405100 eb10     jmp 0x405112
0x00405101 90      nop
0x00405104 90      nop
0x00405105 90      nop
0x00405106 90      nop
0x00405107 90      nop
0x00405108 8a06     mov al, byte [esi]
0x0040510a 46      inc esi
0x0040510b 8b07     mov byte [edi], al
0x0040510d 47      inc edi
; CODE XREFS from entry0 @ 0x4051a6(x), 0x4051bd(x)
0x0040510e 01db     add ebx, ebx
0x00405110 7507     jne 0x405119
; CODE XREF from entry0 @ 0x405100(x)
0x00405112 8b1e     mov ebx, dword [esi]
0x00405114 83efc   sub esi, 0xffffffff
0x00405117 11db     adc ebx, ebx
0x00405119 72ed     jb 0x405108
0x0040511b b8010000 mov eax, 1
0x00405120 01db     add ebx, ebx
0x00405122 7507     jne 0x40512b
0x00405124 8b1e     mov ebx, dword [esi]
0x00405126 83efc   sub esi, 0xffffffff
0x00405129 11db     adc ebx, ebx
0x0040512b 01db     add ebx, ebx
0x0040512f 73ef     jae 0x405120
0x00405131 7509     jne 0x40513c
0x00405133 8b1e     mov ebx, dword [esi]
0x00405135 83efc   sub esi, 0xffffffff
0x00405138 11db     adc ebx, ebx
0x0040513a 73e4     jae 0x405120
0x0040513c 31c9     xor ecx, ecx
0x0040513e 83e803   sub eax, 3
0x00405141 720d     jb 0x405150
0x00405143 c1e008   shl eax, 8
0x00405146 8a06     mov al, byte [esi]
0x00405148 46      inc esi
0x00405149 83f0ff   xor eax, 0xffffffff ; -1
0x0040514c 7474     je 0x4051c2
```

## References

- [1] *The Official Radare2 Book*, [Online; abgerufen im Oktober 2023], <https://book.rada.re/>.