

[QuickNote] Techniques for decrypting BazarLoader strings

kienmanowar.wordpress.com/2022/02/24/quicknote-techniques-for-decrypting-bazarloader-strings/

February 24, 2022

1. Overview

Usually, to make it more difficult for analysts, malware authors will hide important strings and only decrypt these strings during runtime. The famous malwares like Emotet, QakBot or TrickBot often use the one or some functions to perform decrypting strings when needed.

However, on researching and analyzing some other malwares such as **Conti**, **BlackMatter** and **BazarLoader**, instead of using a separate function to decrypt strings, these malwares make it more difficult by saving the encrypted strings on the stack as stack strings. Then, strings are decrypted by XOR-ing with a key value (this value may not be fixed) or through quite complex computation. This technique consumes time of the analyst.

The images below are the pseudocode of the Conti and BlackMatter malware.

```
text:004038C5 mov [ebp+szRstrtmgr.dll], 3
text:004038C9 mov [ebp+szRstrtmgr.dll+1], 33h ; '3'
text:004038CD mov [ebp+szRstrtmgr.dll+2], 40h ; '0'
text:004038D1 mov [ebp+szRstrtmgr.dll+3], 26h ; '&'
text:004038D5 mov [ebp+szRstrtmgr.dll+4], 40h ; '0'
text:004038D9 mov [ebp+szRstrtmgr.dll+5], 64h ; 'd'
text:004038DD mov [ebp+szRstrtmgr.dll+6], 16h
text:004038E1 mov [ebp+szRstrtmgr.dll+7], 26h ; '&'
text:004038E5 mov [ebp+szRstrtmgr.dll+8], 28h ; '+'
text:004038E9 mov [ebp+szRstrtmgr.dll+9], 6Eh ; 'n'
text:004038ED mov [ebp+szRstrtmgr.dll+0Ah], 57h ; 'W'
text:004038F1 mov [ebp+szRstrtmgr.dll+0Bh], 57h ; 'W'
text:004038F5 mov [ebp+szRstrtmgr.dll+0Ch], 50h ; 'P'
text:004038F9 mov al, [ebp+szRstrtmgr.dll]
text:004038FC cmp [ebp+var_18], 0
text:00403900 jnz short loc_403934
text:00403900

27 szRstrtmgr.dll[0] = 3;
28 szRstrtmgr.dll[1] = 0x33;
29 szRstrtmgr.dll[2] = 0x40;
30 szRstrtmgr.dll[3] = 0x26;
31 szRstrtmgr.dll[4] = 0x40;
32 szRstrtmgr.dll[5] = 0x64;
33 szRstrtmgr.dll[6] = 0x16;
34 szRstrtmgr.dll[7] = 0x26;
35 szRstrtmgr.dll[8] = 0x28;
36 szRstrtmgr.dll[9] = 0x6E;
37 szRstrtmgr.dll[0xA] = 0x57;
38 szRstrtmgr.dll[0xB] = 0x57;
39 szRstrtmgr.dll[0xC] = 0x50;
40 for ( j = 0; j < 0xD; ++j )
41 szRstrtmgr.dll[j] = (0x27 + (0x50 - (unsigned __int8)szRstrtmgr.dll[j]) % 0x7F + 0x7F) % 0x7F;
42 LoadLibraryA = (HMODULE (__stdcall *) (LPCSTR))f_dynamic_resolve_api_funcs(0xF, Func_Kernel32_LoadLibraryA, 0x1C);
43 h_Rstrtmgr_dll = LoadLibraryA(szRstrtmgr_dll);
```

```
ext:004096B3 lea eax, [ebp+wsz_TimesNewRoman]
ext:004096B9 mov dword ptr [eax], 17689FAC
ext:004096BF mov dword ptr [eax+4], 17219F8B
ext:004096C5 mov dword ptr [eax+8C], 17689F8B
ext:004096D1 mov dword ptr [eax+10h], 17219F8F
ext:004096D8 mov dword ptr [eax+14h], 176E9FAA
ext:004096E2 mov dword ptr [eax+18h], 17689F95
ext:004096E9 mov dword ptr [eax+1Ch], 17819F96
ext:004096F0 mov ecx, 8
ext:004096F8
ext:004096F5
ext:004096F5 loc_4096F5: ; CODE XREF: sub
ext:004096F5 xor dword ptr [eax], 17819FF8
ext:004096FB add eax, 4
ext:004096FE dec ecx
ext:004096FF jnz short loc_4096F5
ext:00409701 push 58h ; 'X' ; index
ext:00409703 push [ebp+hdc] ; hdc
ext:00409706 call GetDeviceCaps
ext:00409706 mov ecx, eax
ext:0040970C mov eax, [ebp+cy]
ext:0040970E xor edx, edx

81 }
82
83 wsz_TimesNewRoman = wsz_TimesNewRoman;
84 wsz_TimesNewRoman[0] = 0x17689FAC;
85 wsz_TimesNewRoman[1] = 0x17219F8B;
86 wsz_TimesNewRoman[2] = 0x17219F8B;
87 wsz_TimesNewRoman[3] = 0x17649FB8;
88 wsz_TimesNewRoman[4] = 0x17219F8F;
89 wsz_TimesNewRoman[5] = 0x176E9FAA;
90 wsz_TimesNewRoman[6] = 0x17689F95;
91 wsz_TimesNewRoman[7] = 0x17819F96;
92 cnt = 8;
93 do
94 {
95 +wsz_TimesNewRoman ^= 0x17819FF8; // Times New Roman
96 ++wsz_TimesNewRoman;
97 --cnt;
98 }
99 while ( cnt );
100 nPixels = GetDeviceCaps(hdc, LOGPIXELSX);
101 hFont = CreateFontW_0(
102 7 * (cy / (__int64)nPixels),
103 0,
104 0,
105 FW_BOLD,
```

This article uses the BazarLoader samples as an example to demonstrate how to decrypt strings with:

- Automate resolving with IDAPython script.
- Emulate code with IDA uEmu plugin.
- Debugging with IDA Bochs plugin.

2. BazarLoader samples

BazarLoader was first discovered in April 2020. The malware loader has been continuously evolving, allowing attackers to install additional malware, often used for ransomware attacks, dropping Cobalt Strike, and stealing sensitive data. The common assumption is that the distribution and post-exploitation activities of the loader are akin to the Trickbot malware.

These samples are all **64-bit** Windows executable.

- Unpacked sample 1: [cc522400b3fed1d2c4dcca16666ddcff](#)
- Unpacked sample 2: [63c4bb3f1044f36632ce1759b62296dc](#)

3. Decrypt strings

3.1. Using IDAPython script

Analyzing the first sample of BazarLoader, we will see that it uses the same stack strings decryption technique as in BlackMatter ransomware:

```
000000018000A622 jnz loc_18000AAA7
000000018000A622
000000018000A628 lea esi, [rdi+1]
000000018000A628 mov edx, 55FFEF1Eh
000000018000A630 mov cs:byte_18001D330, sil
000000018000A637 mov [rbp+57h+var_40], dil
000000018000A63B mov [rbp+57h+var_3C], 3B8D8A75h
000000018000A642 mov [rbp+57h+var_3C+4], 67CC837Bh
000000018000A649 mov [rbp+57h+var_3C+8], 39938B30h
000000018000A650 mov [rbp+57h+var_3C+0Ch], edx
000000018000A653 mov eax, [rbp+57h+var_3C]
000000018000A656 mov al, [rbp+57h+var_40]
000000018000A659 test al, al
000000018000A65B jnz short loc_18000A672
000000018000A65D
000000018000A65D mov ecx, edi
000000018000A65D decoding loop
000000018000A65F loc_18000A65F: ; CODE
000000018000A65F mov eax, [rbp+rcx+4+57h+var_3C]
000000018000A663 xor eax, edx
000000018000A665 mov [rbp+rcx+4+57h+var_3C], eax
000000018000A669 add rcx, rsi
000000018000A66C cmp rcx, 4
000000018000A670 jb short loc_18000A65F
000000018000A670
80 if ( !byte_18001D330 )
81 {
82 byte_18001D330 = 1;
83 v76 = 0;
84 v76[0] = 0x3B8D8A75;
85 v76[1] = 0x67CC837B;
86 v76[2] = 0x39938B30;
87 v76[3] = 0x55FFEF1E;
88 for ( i = 0i64; i < 4; ++i )
89 {
90 v76[i] ^= 0x55FFEF1Eu;
91 }
92 lstrcpyA = (void (__fastcall *) (void *, int *))f_bazar_dyn_resolve_API(1i64, func_kernel32_lstrcpyA, 0x80);
93 if ( !lstrcpyA )
94 {
95 lstrcpyA(6unk_18001D280, v76);
96 }
97 LOBYTE(v77[0]) = 0;
98 v77[1] = 0x74DDFEF3;
99 v77[2] = 0x2798F3E2;
100 v77[3] = 0x79C7FEBE;
101 v78 = 0x15AB9A92;
102 for ( j = 0i64; j < 4; ++j )
103 v77[j + 1] ^= 0x15AB9A92u;
104 }
105 }
```

To decrypt these strings, you can use **x64dbg** to debug or extract the above values and use **CyberChef** to perform the following:

Recipe	Input
Swap endianness Data format: Hex Word length (bytes): 4 <input checked="" type="checkbox"/> Pad incomplete words	0x3B8D8A75 0x67CC837B 0x39938B30 0x55FFEF1E
From Hex Delimiter: Auto	Output kernel32.dll....
XOR Key: 0x1eeffff5 Scheme: Standard <input type="checkbox"/> Null preserving	

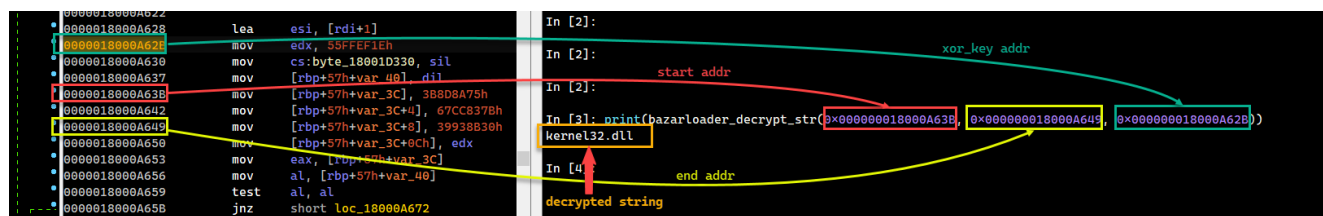
However, debugging with **x64dbg** or using **CyberChef** as above will take more time, to make static analysis easier, I will use **IDAPython** script to decrypt the strings. The code I use is as follows:

```

1  import idc, idaapi, struct
2
3  def bazarloader_decrypt_str(start_ea, end_ea, xor_key_ea):
4      enc_buf = []
5      xor_key = []
6
7      """get xor_key"""
8      xor_key_val = struct.pack("<I", idc.get_operand_value(xor_key_ea, 1))
9      for i in bytearray(xor_key_val): xor_key.append(i)
10
11     '''get encoded bytes'''
12     if start_ea is None or end_ea is None or xor_key_ea is None:
13         print('Not enough information to decrypt')
14         return
15
16     curr = start_ea
17     while curr <= end_ea:
18         enc_val = struct.pack("<I", idc.get_operand_value(curr, 1))
19
20         for i in bytearray(enc_val): enc_buf.append(i)
21         curr = idc.next_head(curr)
22
23     '''decrypt'''
24     for i in range(len(enc_buf)):
25         enc_buf[i] ^= (xor_key[i % len(xor_key)]) & 0xFF
26
27     return ''.join([chr(c) for c in enc_buf])

```

Load this script into IDA, providing the relevant addresses to perform the decryption:



Finally, by using the above script, the analysis process will be much more convenient:

```

46 char v57; // [rsp+D4h] [rbp+2Bh]
47 int sz_advapi32.dll[4]; // [rsp+D8h] [rbp+2Fh]
48
49 if ( !byte_18001D330 )
50 {
51     byte_18001D330 = 1;
52     v55 = 0;
53     sz_kernel32.dll[0] = 0x3B8D8A75;
54     sz_kernel32.dll[1] = 0x67CC837B;
55     sz_kernel32.dll[2] = 0x39938B30;
56     sz_kernel32.dll[3] = 0x55FFEF1E;
57     for ( i = 0i64; i < 4; ++i )
58     {
59         sz_kernel32.dll[i] ^= 0x55FFEF1Eu; // kernel32.dll
60     }
61     lstrcpyA = (LPSTR (__stdcall *) (LPSTR, LPCSTR)) f_bazar_dyn_resolve_API(1i64, func_kernel32_lstrcpyA, 0x80);
62     if ( lstrcpyA )
63     {
64         lstrcpyA(g_sz_kernel32_dll, (LPCSTR)sz_kernel32.dll);
65     }
66     v57 = 0;
67     sz_advapi32.dll[0] = 0x74DDFEF3;
68     sz_advapi32.dll[1] = 0x2798F3E2;
69     sz_advapi32.dll[2] = 0x79C7FEBC;
70     sz_advapi32.dll[3] = 0x15AB9A92;
71     for ( j = 0i64; j < 4; ++j )
72     {
73         sz_advapi32.dll[j] ^= 0x15AB9A92u; // advapi32.dll
74     }
75     lstrcpyA = (LPSTR (__stdcall *) (LPSTR, LPCSTR)) f_bazar_dyn_resolve_API(1i64, func_kernel32_lstrcpyA, 0x80);
76     if ( lstrcpyA )

```

3.2. Using uEmu plugin

In the second sample of BazarLoader, the code that decrypt the stack strings is similar to the Conti ransomware and quite complicated:

```

t:00000020414E986      sub     rsp, 68h
t:00000020414E98A      mov     rax, 6B81043E4875FA28
t:00000020414E994      mov     [rsp+88h+enc_buf], rax
t:00000020414E999      mov     r10, rcx
t:00000020414E99C      lea    rdi, [rsp+88h+dec_buf]
t:00000020414E9A1      mov     r13, r8
t:00000020414E9A4      mov     [rsp+88h+var_41], 6Fh; 'o'
t:00000020414E9A9      lea    rsi, [rsp+88h+enc_buf]
t:00000020414E9AE      mov     r8, r9
t:00000020414E9B1      mov     r12d, edx
t:00000020414E9B4      mov     ecx, 13h
t:00000020414E9B9      mov     [rsp+88h+var_2D], 0
t:00000020414E9BE      mov     rax, 6C4F1D443320E4Fh
t:00000020414E9C8      mov     r9d, 7Fh
t:00000020414E9CE      mov     [rsp+88h+var_4B], rax
t:00000020414E9D3      mov     [rsp+88h+var_43], 3422h
t:00000020414E9DA      rep     movsb
t:00000020414F9A4      rep     movsb
t:00000020414F41C      sub     rsp, 58h
t:00000020414F420      xor     r8d, r8d
t:00000020414F423      mov     r9d, 7Fh
t:00000020414F429      mov     rax, 6B70702306676B73h
t:00000020414F433      mov     qword ptr [rsp+58h+enc_buf], rax
t:00000020414F438      mov     rax, 2C6B743E700B6703h
t:00000020414F442      mov     qword ptr [rsp+58h+enc_buf+8], rax
t:00000020414F447      movups  xmm0, xmmword ptr [rsp+58h+enc_buf]
t:00000020414F44C      mov     word ptr [rsp+58h+enc_buf+10h], 752Ch
t:00000020414F453      movups  xmmword ptr [rsp+58h+dec_buf], xmm0
t:00000020414F458      mov     [rsp+58h+var_E], 0
t:00000020414F45D      mov     word ptr [rsp+58h+dec_buf+10h], 752Ch
t:00000020414F464      loc_20414F464:
; CODE XREF: sub_2

```

```

7 char v13; // [rsp+58h] [rbp-2Dh]
8
9 *enc_buf = 0x1D01443E45754F03164;
10 enc_buf[0x12] = 0x6F;
11 v13 = 0;
12 *enc_buf[8] = 0x6C4F1D443320E4F164;
13 *enc_buf[0x10] = 0x3422;
14 qmemcpy(dec_buf, enc_buf, sizeof(dec_buf));
15 for ( i = 0i64; i != 19; ++i )
16 {
17     dec_buf[i] = ((0xFFFFFFFF * (dec_buf[i] - 0x6F)) % 0x7F + 0x7F) % 0x7F;
18 }
19 v1 = sub_20414AB1A(a1, dec_buf);
20 return (v1)(dwFlags, lpModuleName, phModule);
21
22
23

```

```

8 i = 0i64;
9 *enc_buf = 0x6B70702306676B73164;
10 *enc_buf[8] = 0x2C6B743E700B6703164;
11 *dec_buf = *enc_buf;
12 v6 = 0;
13 *dec_buf[0x10] = 0x752C;
14
15 do
16 {
17     dec_buf[i] = (0x1C * (dec_buf[i] - 0x75) % 0x7F + 0x7F) % 0x7F;
18     ++i;
19 }
20 while ( i != 18 );
21 v2 = sub_20414AB1A(a1, dec_buf);
22 return v2();
23

```

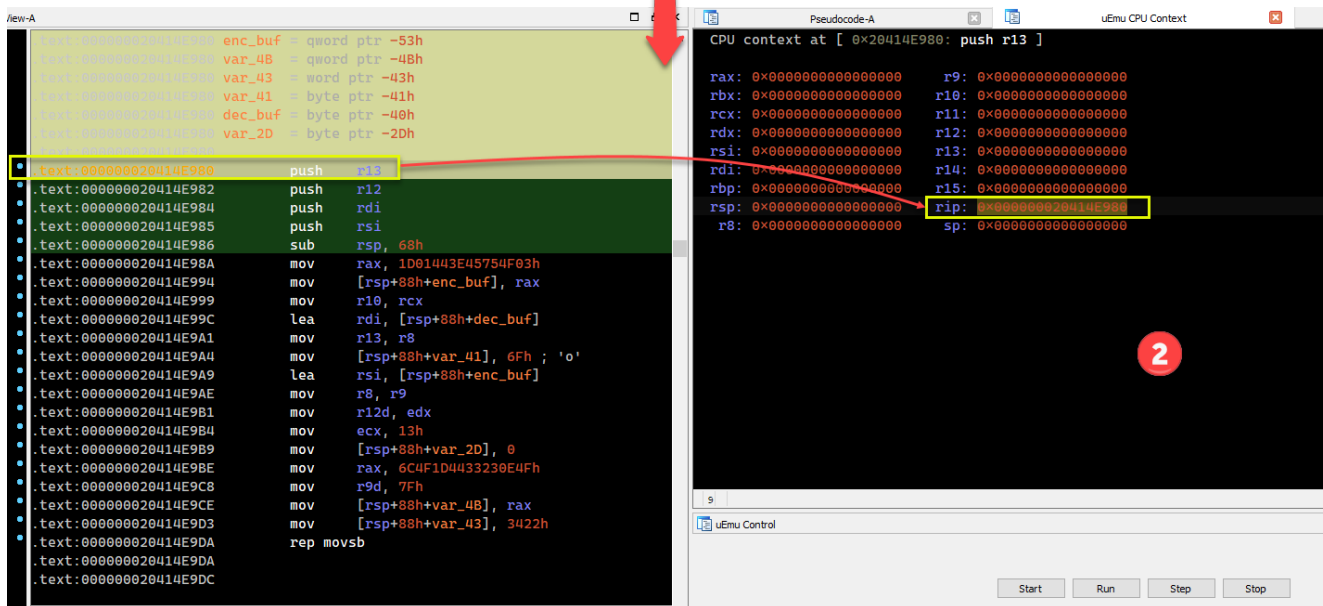
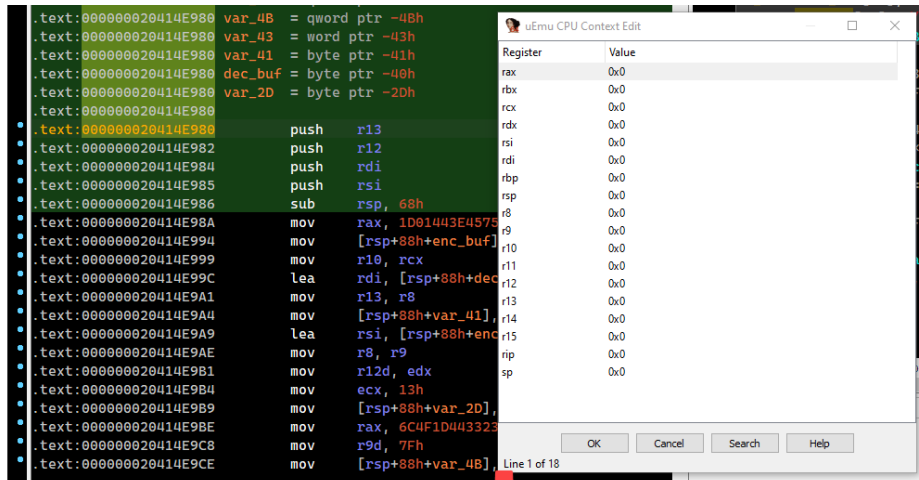
With the code as shown in the figure, the implementation of using IDAPython script will be difficult and not feasible. The most suitable solution for this case is to use an emulator to emulate the code. Here, I will use uEmu, a tiny cute emulator plugin for IDA based on unicorn engine.

Very easy to emulate the decoding code with uEmu:

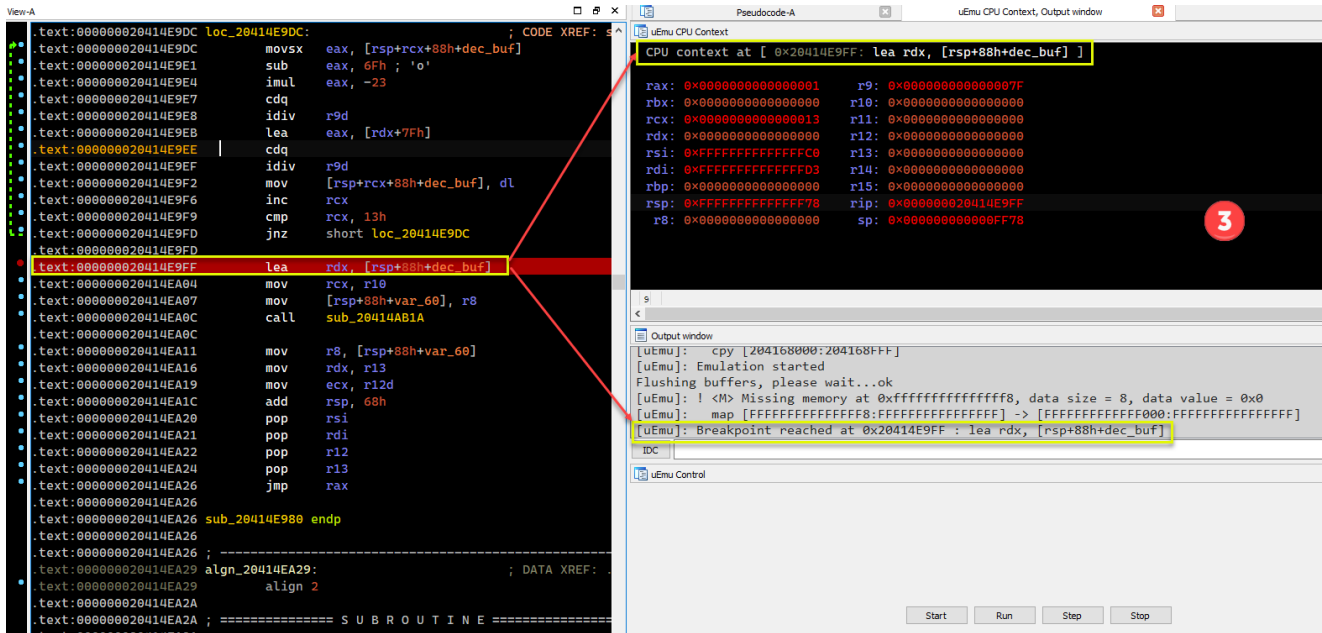
First, set a breakpoint at the address after the string has been decrypted.

```
.text:0000000020414E9DC    movsx  eax, [rsp+rcx+88h+dec_buf]
.text:0000000020414E9E1    sub    eax, 6Fh ; 'o'
.text:0000000020414E9E4    imul  eax, -23
.text:0000000020414E9E7    cdq
.text:0000000020414E9E8    idiv  r9d
.text:0000000020414E9EB    lea   eax, [rdx+7Fh]
.text:0000000020414E9EE    cdq
.text:0000000020414E9EF    idiv  r9d
.text:0000000020414E9F2    mov   [rsp+rcx+88h+dec_buf], dl
.text:0000000020414E9F6    inc   rcx
.text:0000000020414E9F9    cmp   rcx, 13h
.text:0000000020414E9FD    jnz   short loc_20414E9DC
.text:0000000020414E9FE
.text:0000000020414E9FF    lea   rdx, [rsp+88h+dec_buf]
.text:0000000020414EA04    mov   rcx, r10
.text:0000000020414EA07    mov   [rsp+88h+var_60], r8
```

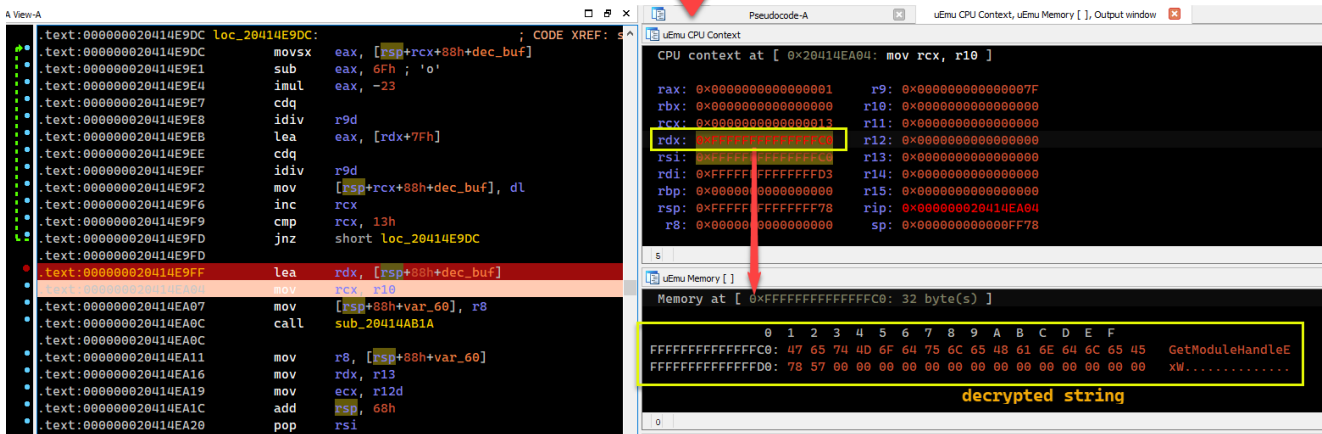
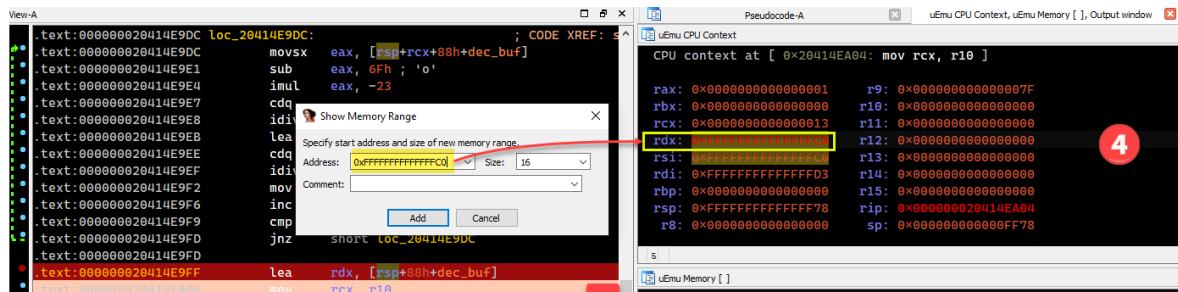
Go to the beginning of the function and select the starting address of the function, then start uEmu. The **CPU Context Edit** window will appear, click **OK** to continue. uEmu will now initialize the emulator. Check the CPU context to see if the address of the **EIP/RIP** register is pointed at the beginning of the function:



Then, through **uEmu Control**, you can trace the code by **Step** or **Run** to emulates instructions until breakpoint is reached. During execution, uEmu will ask about unmapped memory, select **No** to continue.



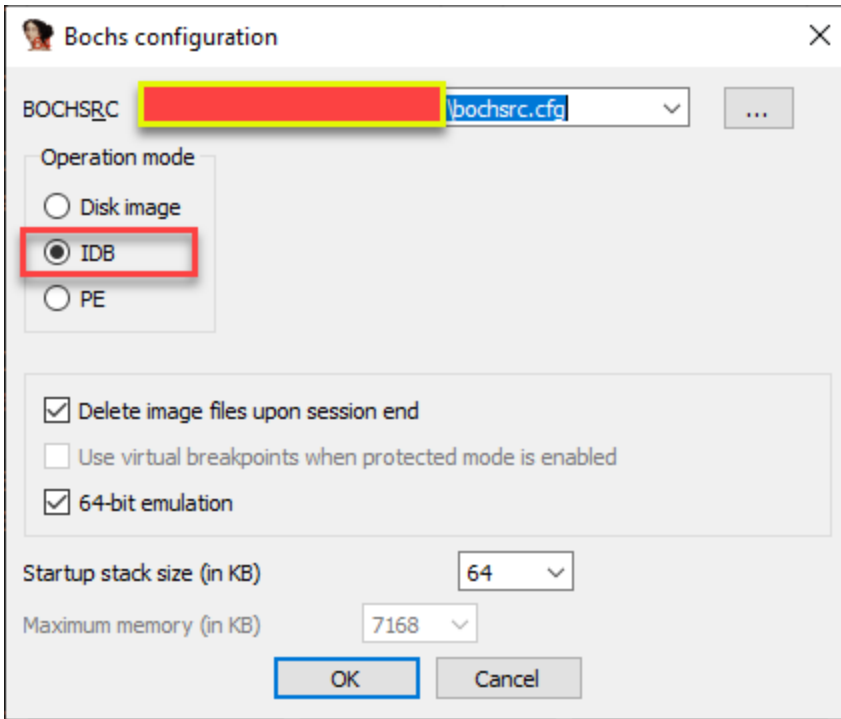
Press **Step**, to trace over the **lea** command. Then using uEmu's **Show Memory Range** feature, enter the address of the **rdx** register and select **Add**. The result will be similar to the following:



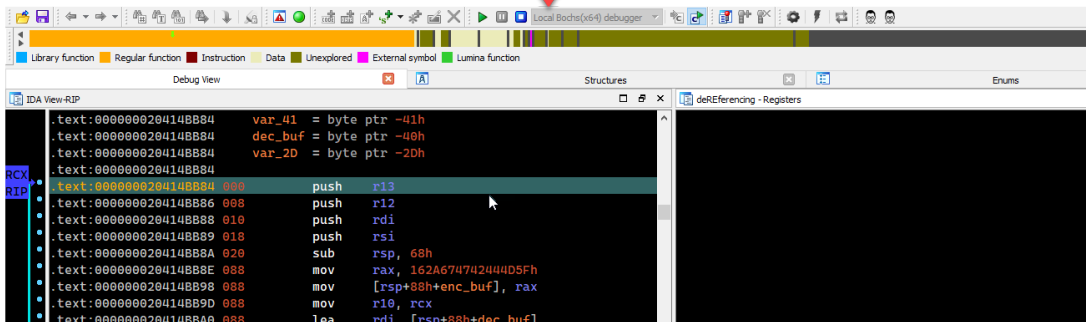
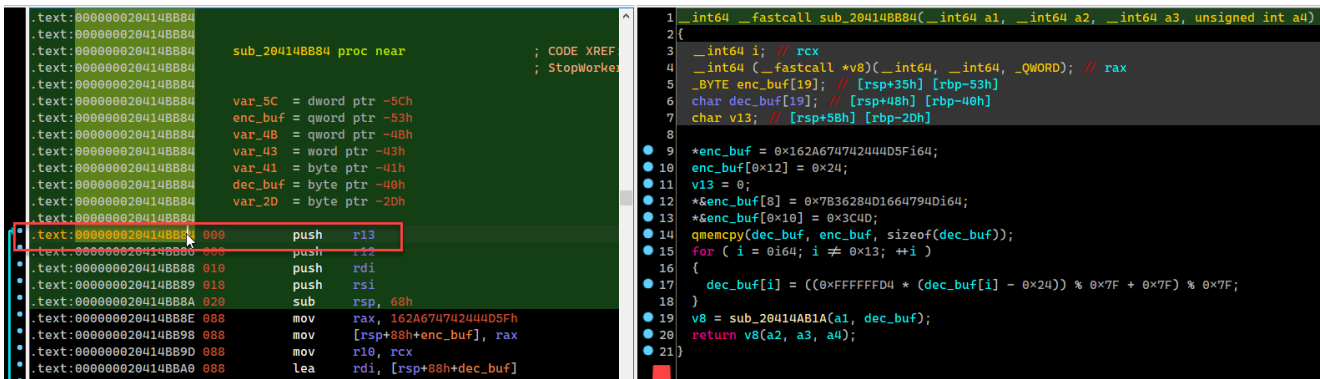
3.3. Debugging with IDA Bochs Plugin

IDA Bochs debugger plugin allows malware researchers to debug malicious code in a safe/emulated environment. For more information please visit [1], [2].

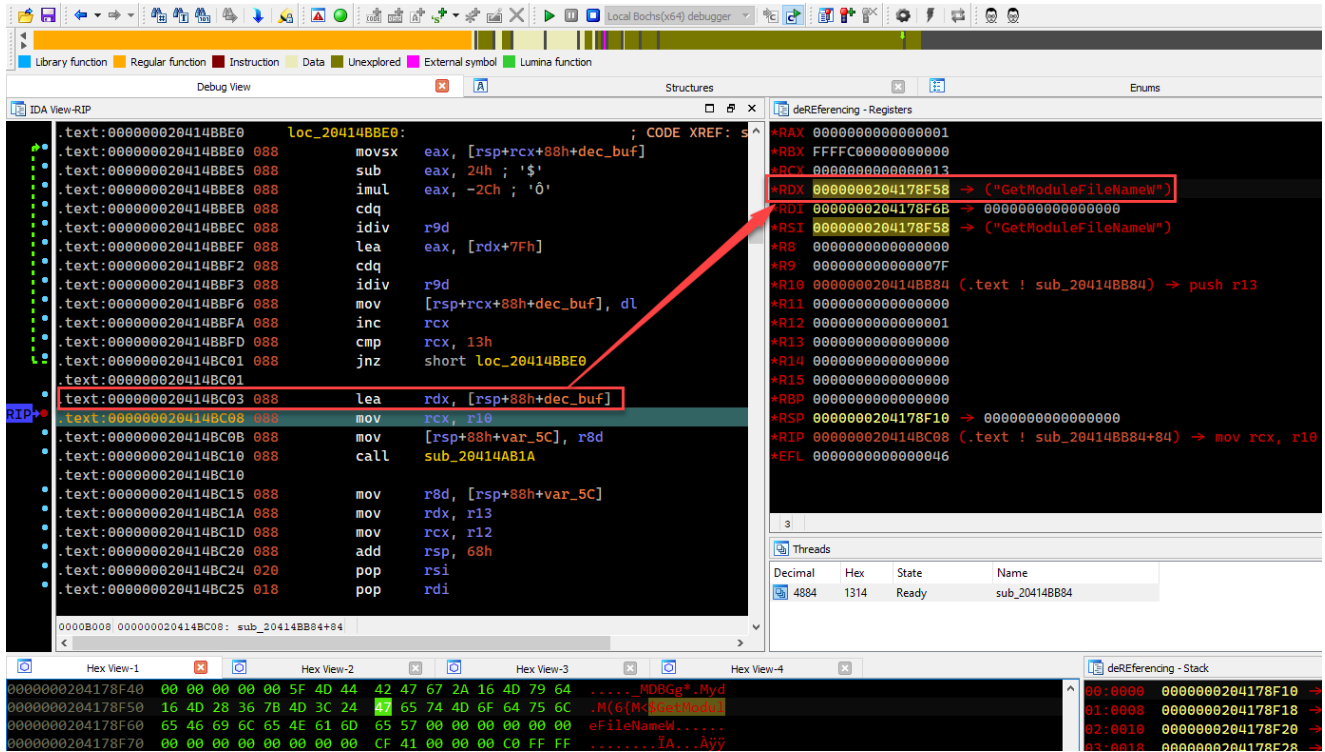
In order to debug the code that decrypts the string, we configure the Bochs plugin to work in **IDB mode**. This mode is used to debug code snippets by simply selecting the code from the database.



Next, select the position or code snippets to debug, then press **F9** to start debugging:



From here you can trace the code as usual or simply set a breakpoint at the address after finished decrypting the string and press **F9**. The resulting at **rdx** register will point to the decrypted string as follows:



4. References

End.

m4n0w4r