[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.

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:

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

Recipe

Swap endianness

Data format: Hex | Word length (bytes): 4 | Pad incomplete words

From Hex

Delimiter: Auto

XOR

Key: 0x1eefff55 (HEX)

Scheme: Standard | Null preserving

Input

```
0x3B8D8A75
0x67CC837B
0x39938B30
0x55FFEF1E
```

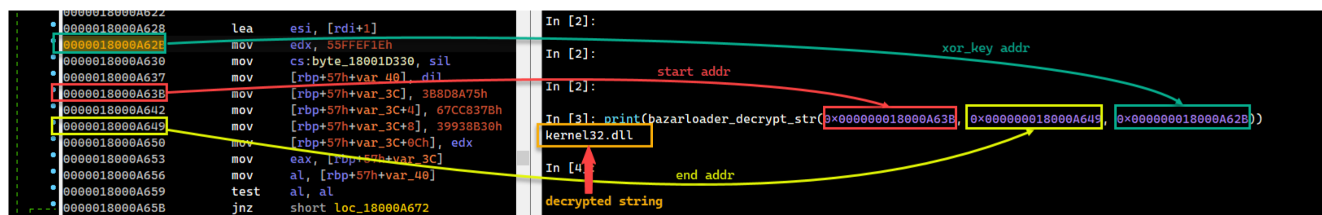
Output

```
kernel32.dll....
```

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:

```python
import idc, idaapi, struct

def bazarloader_decrypt_str(start_ea, end_ea, xor_key_ea):
    enc_buf = []
    xor_key = []

    """get xor_key"""
    xor_key_val = struct.pack("<I", idc.get_operand_value(xor_key_ea, 1))
    for i in bytearray(xor_key_val): xor_key.append(i)

    '''get encoded bytes'''
    if start_ea is None or end_ea is None or xor_key_ea is None:
        print ('Not enough information to decrypt')
        return

    curr = start_ea
    while curr <= end_ea:
        enc_val = struct.pack("<I", idc.get_operand_value(curr, 1))

        for i in bytearray(enc_val): enc_buf.append(i)
        curr = idc.next_head(curr)

    '''decrypt'''
    for i in range(len(enc_buf)):
        enc_buf[i] ^= (xor_key[i % len(xor_key)]) & 0xFF

    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:000000020414E986    sub     rsp, 68h
t:000000020414E98A    mov     rax, 1D01443E45754F03h
t:000000020414E994    mov     [rsp+88h+enc_buf], rax
t:000000020414E999    mov     r10, rcx
t:000000020414E99C    lea     rdi, [rsp+88h+dec_buf]
t:000000020414E9A1    mov     r13, r8
t:000000020414E9A4    mov     [rsp+88h+var_41], 6Fh ; 'o'
t:000000020414E9A9    lea     rsi, [rsp+88h+enc_buf]
t:000000020414E9AE    mov     r8, r9
t:000000020414E9B1    mov     r12d, edx
t:000000020414E9B4    mov     ecx, 13h
t:000000020414E9B9    mov     [rsp+88h+var_2D], 0
t:000000020414E9BE    mov     rax, 6C4F1D4433230E4Fh
t:000000020414E9C8    mov     r9d, 7Fh
t:000000020414E9CE    mov     [rsp+88h+var_4B], rax
t:000000020414E9D3    mov     [rsp+88h+var_43], 3422h
t:000000020414E9DA    rep movsb
```

```
7   char v13; // [rsp+5Bh] [rbp-2Dh]
8
9   *enc_buf = 0x1D01443E45754F03i64;
10  enc_buf[0x12] = 0x6F;
11  v13 = 0;
12  *&enc_buf[8] = 0x6C4F1D4433230E4Fi64;
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] = ((0xFFFFFFE9 * (dec_buf[i] - 0x6F)) % 0x7F + 0x7F) % 0x7F;
18  }
19  v1 = sub_20414AB1A(a1, dec_buf);
20  return (v1)(dwFlags, lpModuleName, phModule);
21 }
```

1

```
t:000000020414F41C    sub     rsp, 58h
t:000000020414F420    xor     r8d, r8d
t:000000020414F423    mov     r9d, 7Fh
t:000000020414F429    mov     rax, 6B70702306676B73h
t:000000020414F433    mov     qword ptr [rsp+58h+enc_buf], rax
t:000000020414F438    mov     rax, 2C6B743E700B6703h
t:000000020414F442    mov     qword ptr [rsp+58h+enc_buf+8], rax
t:000000020414F447    movups  xmm0, xmmword ptr [rsp+58h+enc_buf]
t:000000020414F44C    mov     word ptr [rsp+58h+enc_buf+10h], 752Ch
t:000000020414F453    movups  xmmword ptr [rsp+58h+dec_buf], xmm0
t:000000020414F458    mov     [rsp+58h+var_E], 0
t:000000020414F45D    mov     word ptr [rsp+58h+dec_buf+10h], 752Ch
t:000000020414F45D
t:000000020414F464
t:000000020414F464 loc_20414F464:                  ; CODE XREF: sub_2
```

```
9   i = 0i64;
10  *enc_buf = 0x6B70702306676B73i64;
11  *&enc_buf[8] = 0x2C6B743E700B6703i64;
12  *dec_buf = *enc_buf;
13  v6 = 0;
14  *&dec_buf[0x10] = 0x752C;
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 }
```

2

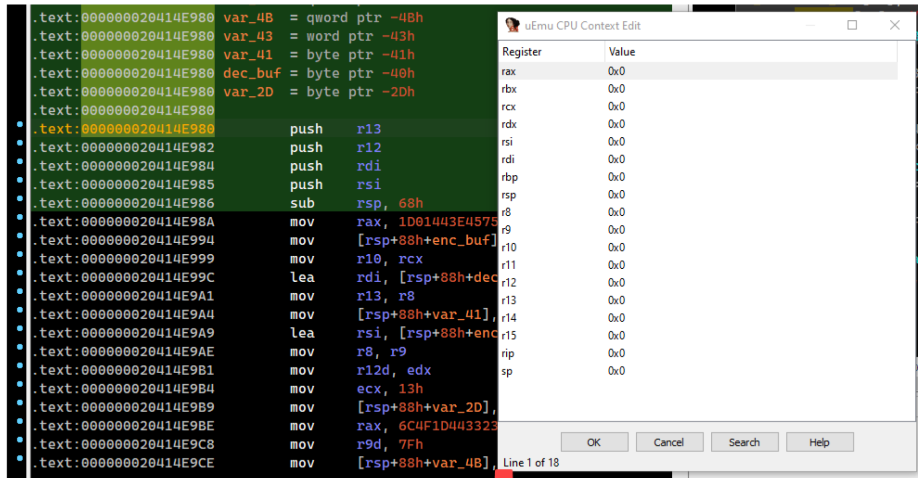
With the code as shown in the figure, the implementation of using IDApython script will be difficult and not feasible. The most suitable solution for this case is to use an emulator to emulate the code. Here, I will use uEmu, a tiny cute emulator plugin for IDA based on unicorn engine.

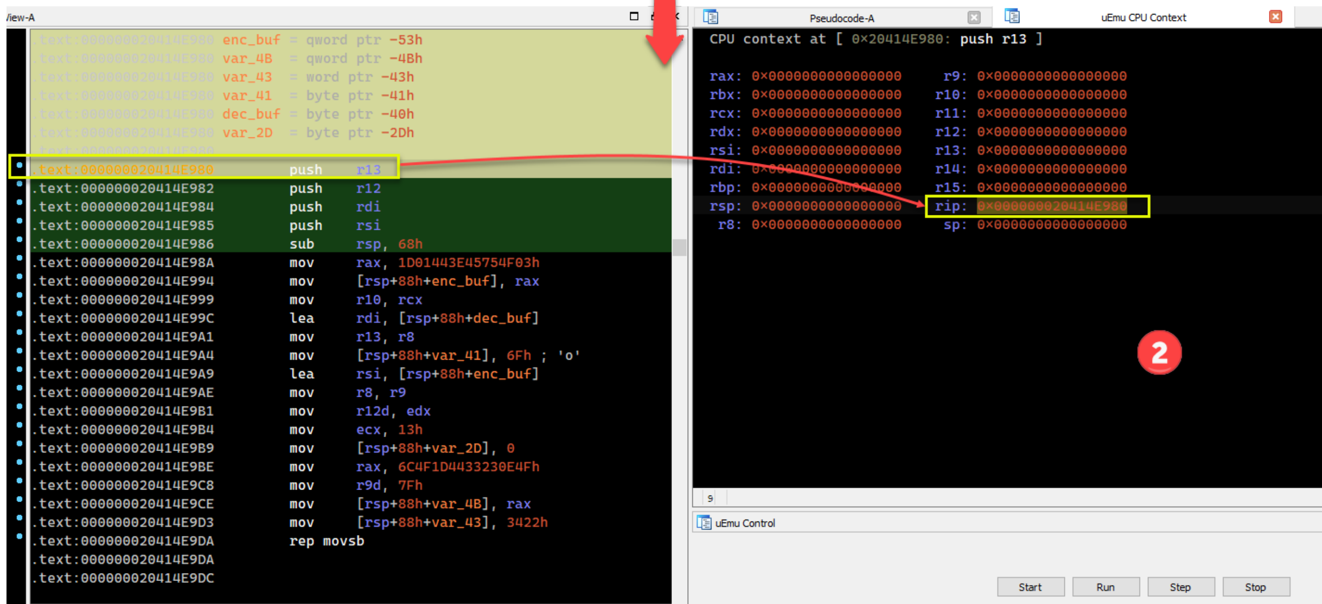
Very easy to emulate the decoding code with uEmu:

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

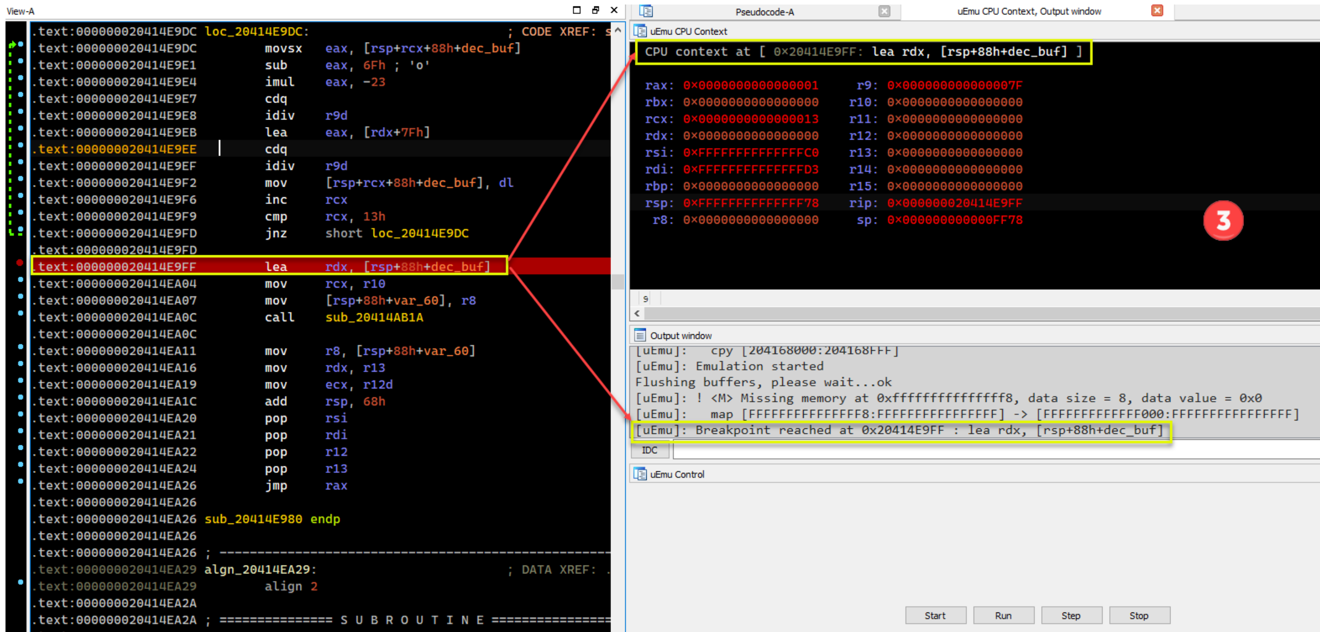
```
.text:000000020414E9DC         movsx    eax, [rsp+rcx+88h+dec_buf]
.text:000000020414E9E1         sub      eax, 6Fh ; 'o'
.text:000000020414E9E4         imul     eax, -23
.text:000000020414E9E7         cdq
.text:000000020414E9E8         idiv     r9d
.text:000000020414E9EB         lea      eax, [rdx+7Fh]
.text:000000020414E9EE         cdq
.text:000000020414E9EF         idiv     r9d
.text:000000020414E9F2         mov      [rsp+rcx+88h+dec_buf], dl
.text:000000020414E9F6         inc      rcx
.text:000000020414E9F9         cmp      rcx, 13h
.text:000000020414E9FD         jnz      short loc_20414E9DC
.text:000000020414E9FD
.text:000000020414E9FF         lea      rdx, [rsp+88h+dec_buf]
.text:000000020414EA04         mov      rcx, r10
.text:000000020414EA07         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:

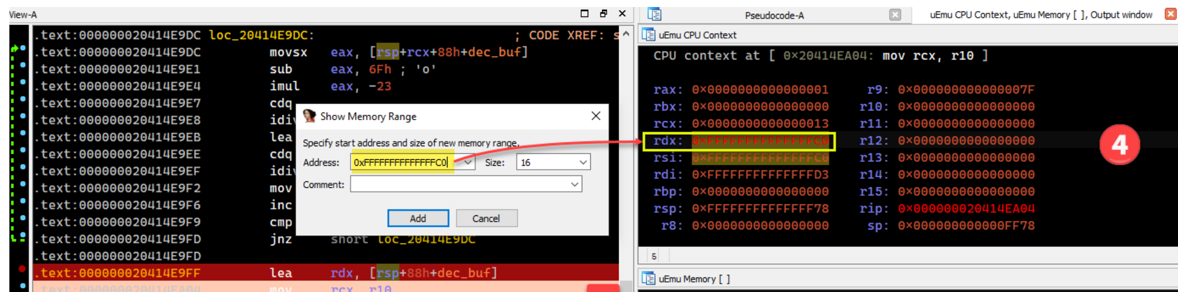


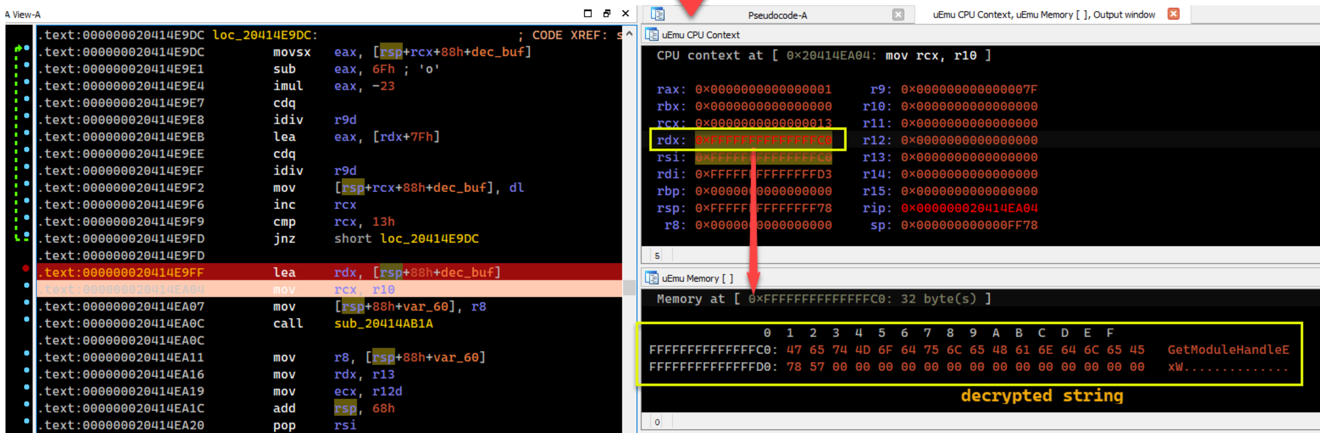


Then, through **uEmu Control**, you can trace the code by **Step** or **Run** to emulates instructions until breakpoint is reached. During execution, uEmu will ask about unmapped memory, select **No** to continue.



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:

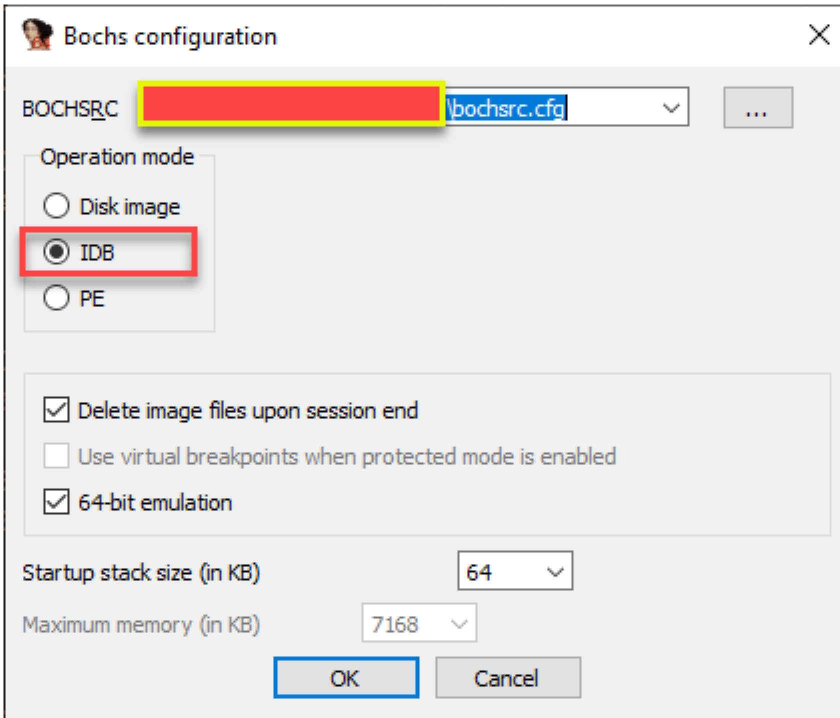




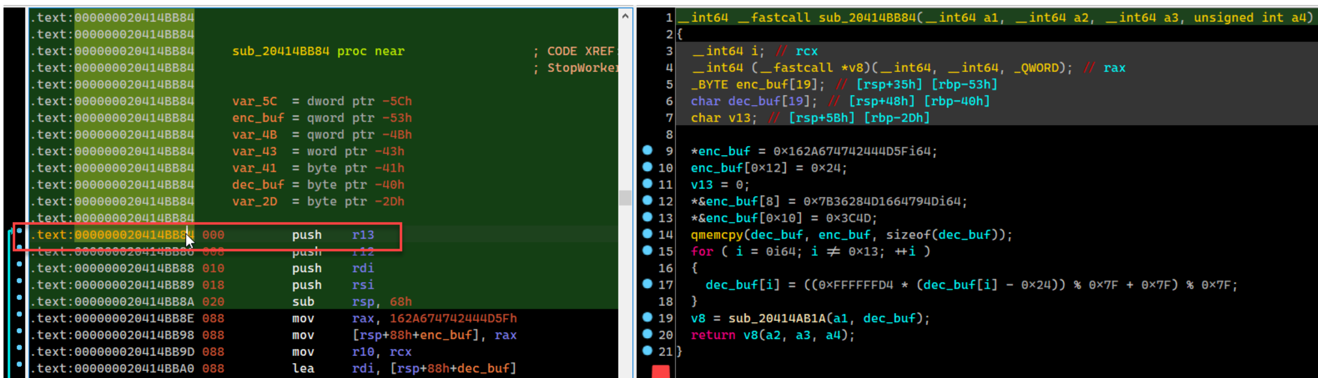
3.3. Debugging with IDA Bochs Plugin

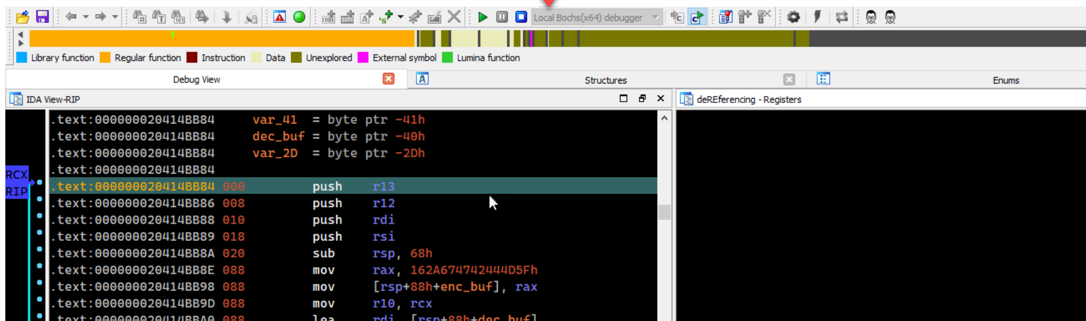
IDA Bochs debugger plugin allows malware researchers to debug malicious code in a safe/emulated environment. For more information please visit [1], [2].

In order to debug the code that decrypt 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.

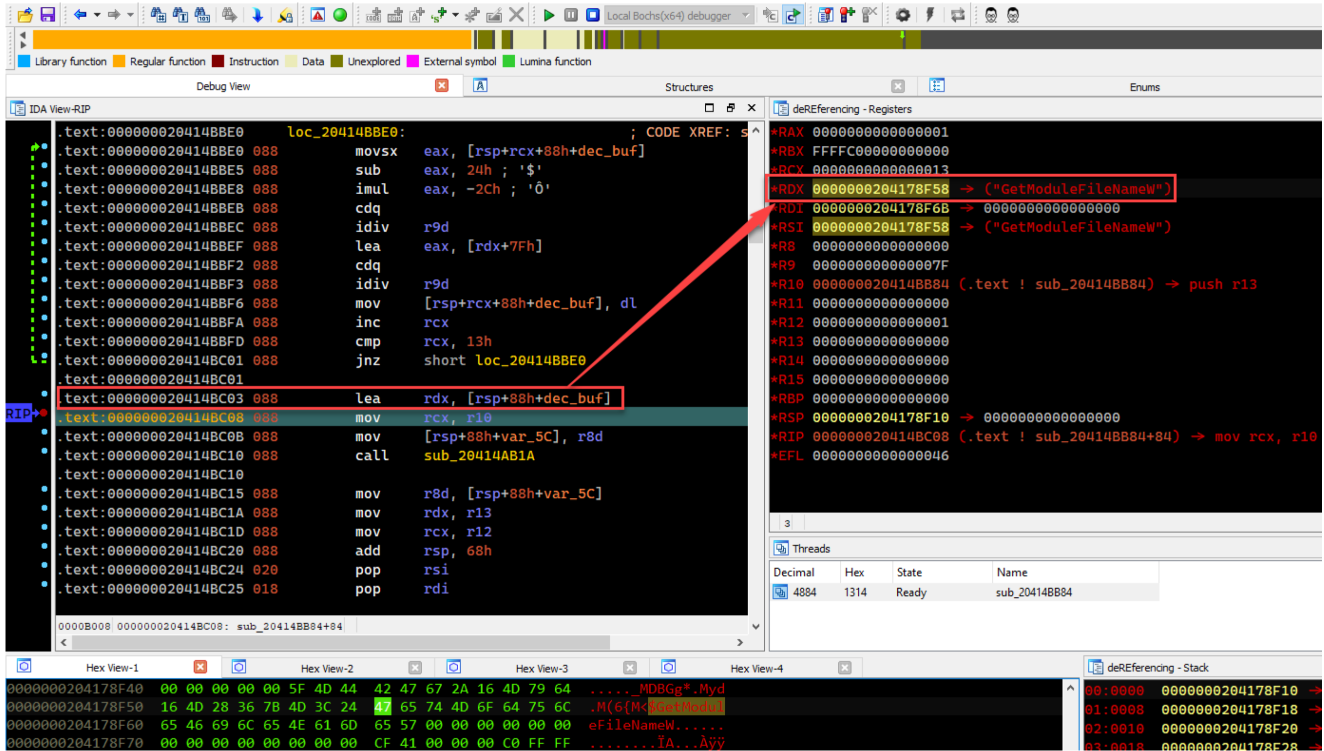


Next, select the position or code snippets to debug, then press **F9** to start debugging:





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



4. References

End.

m4n0w4r