



Using IDAPython to Make Your Life Easier: Part 1

57,338 people reacted

2 5 min. read

SHARE



By Josh Grunzweig
December 29, 2015 at 3:00 PM
Category: Unit 42
Tags: IDA Pro, IDAPython, malware

As a malware reverse engineer, I often find myself using IDA Pro in my day-to-day activities. It should come as no surprise, seeing as IDA Pro is the industry standard (although alternatives such as radare2 and Hopper are gaining traction). One of the more powerful features of IDA that I implore all reverse engineers to make use of is the Python addition, aptly named ‘IDAPython’, which exposes a large number of IDA API calls. Of course, users also get the added benefit of using Python, which gives them access to the wealth of capabilities that the scripting language provides.

Unfortunately, there’s surprisingly little information in the way of tutorials when it comes to IDAPython. Some exceptions to this include the following:

- “The IDA Pro Book” by Chris Eagle
- “The Beginner’s Guide to IDAPython” by Alex Hanel
- “IDAPython Wiki” by Magic Lantern

In the hopes of increasing the amount of IDAPython tutorial material available to analysts, I’m providing examples of code I write as interesting use-cases arise. For Part 1 of this series, I’m going to walk through a situation where I was able to write a script to thwart multiple instances of string obfuscation witnessed in a malware sample.

Background

While reverse-engineering a malicious sample, I encountered the following function:

```
.text:00405C7D ; -----
.text:00405C7D
.text:00405C7D loc_405C7D:      mov     eax, 1          ; CODE XREF: sub_405BF0+6B j
.text:00405C82      cmp     edi, eax
.text:00405C84      jle     short loc_405CA3
.text:00405C86      mov     ecx, dword_41C050
.text:00405C8C      shl     ecx, 9
.text:00405C8F      lea     ecx, unk_41E477[ecx]
.text:00405C95      loc_405C95:      ; CODE XREF: sub_405BF0+B1 j
.text:00405C95      mov     dl, [eax+esi]
.text:00405C98      xor     dl, [esi]
.text:00405C9A      inc     eax
.text:00405C9B      cmp     eax, edi
.text:00405C9D      mov     [ecx+eax-1], dl
.text:00405CA1      jl      short loc_405C95
.text:00405CA3      loc_405CA3:      ; CODE XREF: sub_405BF0+8B j
.text:00405CA3      ; sub_405BF0+94 j
.text:00405CA3      push    offset CriticalSection ; lpCriticalSection
.text:00405CA8      call    ds:LeaveCriticalSection
.text:00405CAE      mov     eax, dword_41C050
.text:00405CB3      shl     eax, 9
.text:00405CB6      add     eax, offset unk_41E478
.text:00405CBB      pop     edi
.text:00405CBB      retn
.text:00405CBC      sub_405BF0      endp
.text:00405CBC ; -----
```

Figure 1 String decryption function

Based on experience, I suspected this might be used to decrypt data contained in the binary. The number of references to this function supported my suspicion.

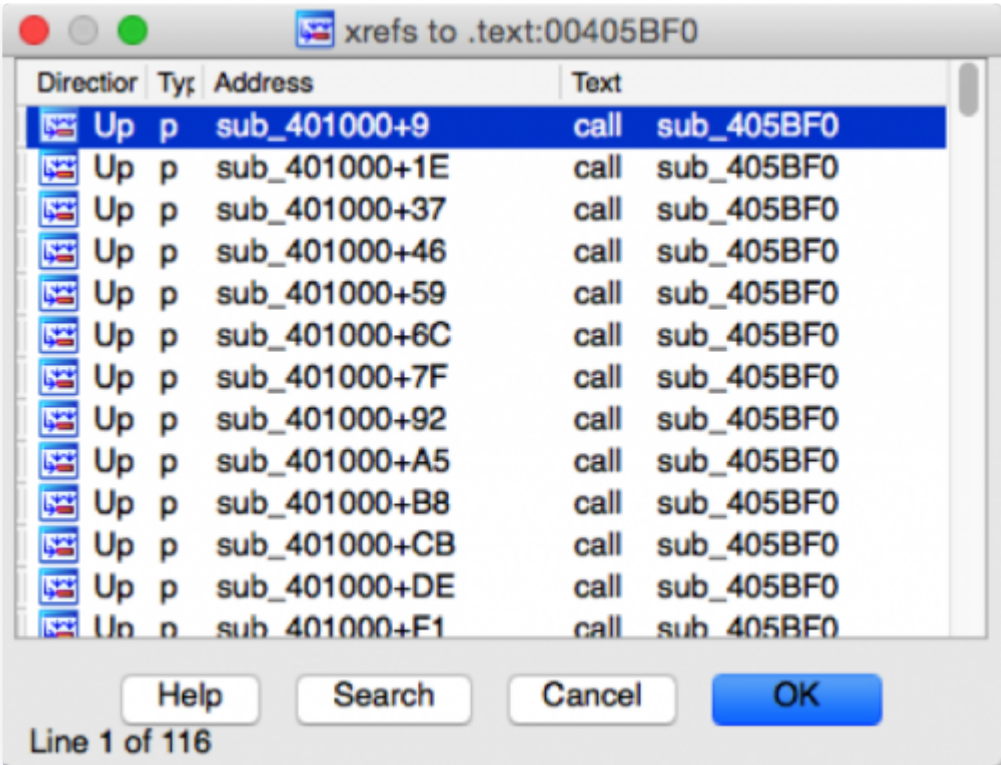


Figure 2 High number of references to suspect function

As we can see in figure 2, there are 116 instances where this particular function is called. In each instance where this function is called, a blob of data is being supplied as an argument to this function via the ESI register.

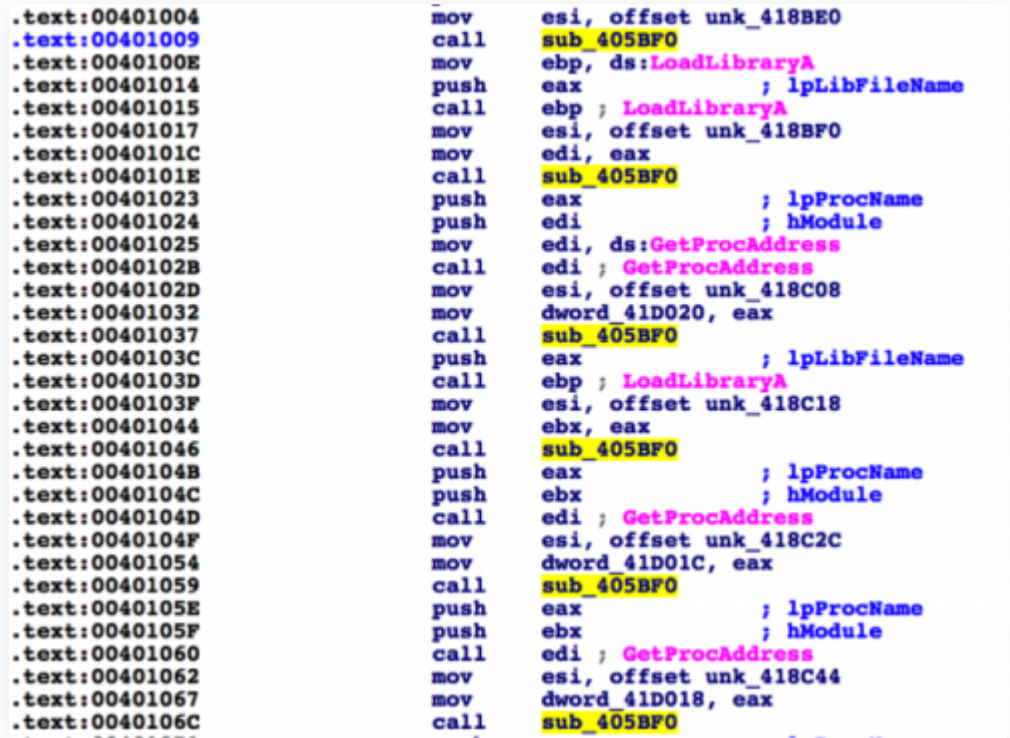


Figure 3 Instances where the suspect function (405BF0) is called

At this point I am confident that this function is being used by the malware to decrypt strings during runtime. When faced with this type of situation, I typically have a few choices:

1. I can manually decrypt and rename these obfuscated strings
2. I can dynamically run this sample and rename the strings as I encounter them
3. I can write a script that will both decrypt these strings and rename them for me

If this were a situation where the malware was only decrypting a few strings overall, I might take the first or second approach. However, as we've identified previously, this function is being used 116 times, so the scripting approach will make a lot more sense.

Scripting in IDAPython

The first step in defeating this string obfuscation is to identify and replicate the decryption function. Fortunately for us, this particular decryption function is quite simple. The function is simply taking the first character of the blob and using it as a single-byte XOR key for the remaining data.

E4 91 96 88 89 8B 8A CA 80 88 88

In the above example, we would take the 0xE4 byte and XOR it against the remaining data. Doing so results in the string of 'urlmon.dll'. In Python, we can replicate this decryption as such:

```
5 while c < length:
6     o += chr(ord(data[0]) ^ ord(data[c]))
7     c += 1
8 return o
```

In testing this code, we get the expected result.

```
1 >>> from binascii import *
2 >>> d = unhexlify("E4 91 96 88 89 8B 8A CA 80 88 88".replace(" ", ''))
3 >>> decrypt(d)
4 'urlmon.dll'
```

The next step for us would be to identify what code is referencing the decryption function, and extracting the data being supplied as an argument. Identifying references to a function in IDA proves to be quite simple, as the XrefsTo() API function does exactly this. For this script, I'm going to hardcode the address of the decryption script. The following code can be used to identify the addresses of the references to the decryption function. As a test, I'm simply going to print out the addresses in hexadecimal.

```
1 for addr in XrefsTo(0x00405BF0, flags=0):
2     print hex(addr.frm)
3
4 Result:
5 0x401009L
6 0x40101eL
7 0x401037L
8 0x401046L
9 0x401059L
10 0x40106cL
11 0x40107fL
12 <truncated>
```

Getting the supplied argument to these cross-references and extracting the raw data proves to be slightly more tricky, but certainly not impossible. The first thing we'll want to do is get the offset address provided in the 'mov esi, offset unk_??' instruction that proceeds the call to the string decryption function. To do this, we're going to step backward one instruction at a time for each reference to the string decryption function and look for a 'mov esi, offset [addr]' instruction. To get the actual address of the offset address, we can use the GetOperandValue() API function.

The following code allows us to accomplish this:

```
1 def find_function_arg(addr):
2     while True:
3         addr = idc.PrevHead(addr)
4         if GetMnem(addr) == "mov" and "esi" in GetOpnd(addr, 0):
5             print "We found it at 0x%x" % GetOperandValue(addr, 1)
6             break
7
8 Example Results:
9 Python>find_function_arg(0x00401009)
10 We found it at 0x418be0
```

Now we simply need to extract the string from the offset address. Normally we would use the GetString() API function, however, since the strings in question are raw binary data, this function will not work as expected. Instead, we're going to iterate byte-by-byte until we reach a null terminator. The following code can be used to accomplish this:

```
1 def get_string(addr):
2     out = ""
3     while True:
4         if Byte(addr) != 0:
5             out += chr(Byte(addr))
6         else:
7             break
8         addr += 1
9     return out
```

At this point, it's simply a matter of taking everything we've created thus far and putting it together.

```
1 def find_function_arg(addr):
2     while True:
3         addr = idc.PrevHead(addr)
4         if GetMnem(addr) == "mov" and "esi" in GetOpnd(addr, 0):
5             return GetOperandValue(addr, 1)
6     return ""
7
8 def get_string(addr):
9     out = ""
10    while True:
11        if Byte(addr) != 0:
12            out += chr(Byte(addr))
13        else:
14            break
15        addr += 1
16    return out
17
18 def decrypt(data):
19     length = len(data)
20     c = 1
21     o = ""
22     while c < length:
23         o += chr(ord(data[0]) ^ ord(data[c]))
24         c += 1
25     return o
```



```
30 string = get_string(ref)
31 dec = decrypt(string)
32 print "Ref Addr: 0x%x | Decrypted: %s" % (x.frm, dec)
33
34 Results:
35 [*] Attempting to decrypt strings in malware
36 Ref Addr: 0x401009 | Decrypted: urlmon.dll
37 Ref Addr: 0x40101e | Decrypted: URLDownloadToFileA
38 Ref Addr: 0x401037 | Decrypted: wininet.dll
39 Ref Addr: 0x401046 | Decrypted: InternetOpenA
40 Ref Addr: 0x401059 | Decrypted: InternetOpenUrlA
41 Ref Addr: 0x40106c | Decrypted: InternetReadFile
42 <truncated>
```

We can see all of the decrypted strings within the malware. While we can stop at this point, if we take the next step of providing a comment of the decrypted string at both the string decryption reference address and the position of the encrypted data, we can easily see what data is being provided. To do this, we'll make use of the `MakeComm()` API function. Adding the following two lines of code after our last print statement will add the necessary comments:

```
1 MakeComm(x.frm, dec)
2 MakeComm(ref, dec)
```

Adding this extra step cleans up the cross-reference view nicely, as we can see below. Now we can easily identify where particular strings are being referenced.

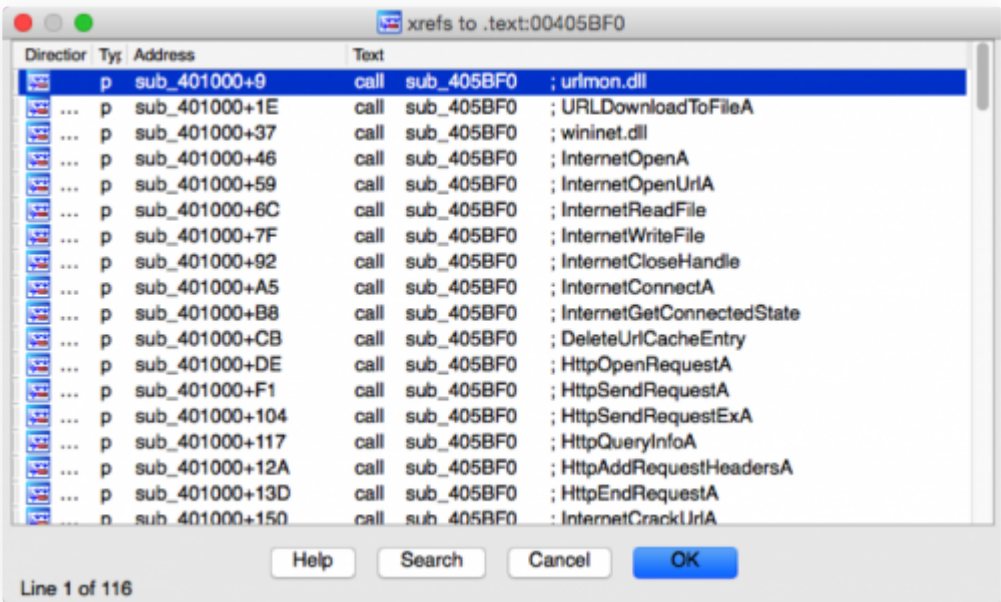


Figure 4 Cross-reference to string decryption after running IDAPython script

Additionally, when navigating the disassembly, we can see the decrypted strings as comments.

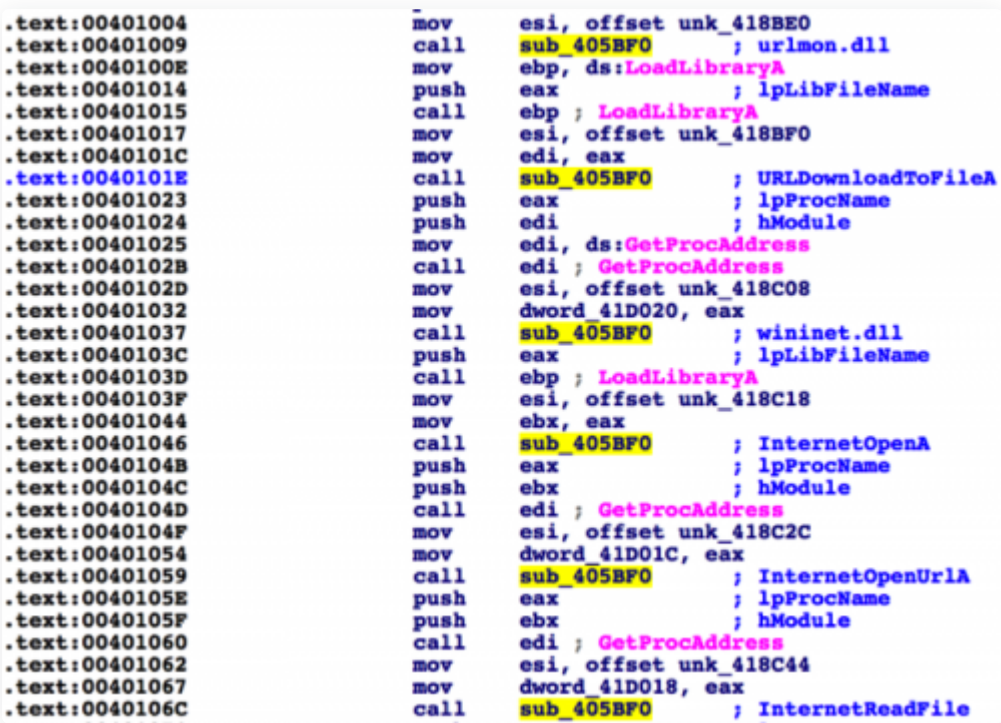


Figure 5 Assembly after IDAPython script is run

Conclusion

Using IDAPython, we were able to take an otherwise difficult task of decrypting 161 instances of encrypted strings in a malicious binary and defeat the binary quite easily. As we've seen, IDAPython can be a powerful tool for a reverse engineer, simplifying various tasks and saving precious time.

Get updates from

This site uses cookies essential to its operation, for analytics, and for personalized content and ads. By continuing to browse this site, you acknowledge the use of cookies. [Privacy statement](#)

[Cookie Settings](#)

Networks!

Sign up to receive the latest news, cyber threat intelligence and research from us

Email address

Subscribe

☐

进行人机身份验证

reCAPTCHA

隐私权 - 使用条款

By submitting this form, you agree to our Terms of Use and acknowledge our Privacy Statement.



Popular Resources

- Resource Center
- Blog
- Communities
- Tech Docs
- Unit 42
- Sitemap

Legal Notices

- Privacy
- Terms of Use
- Documents

Account

- Manage Subscriptions

Report a Vulnerability