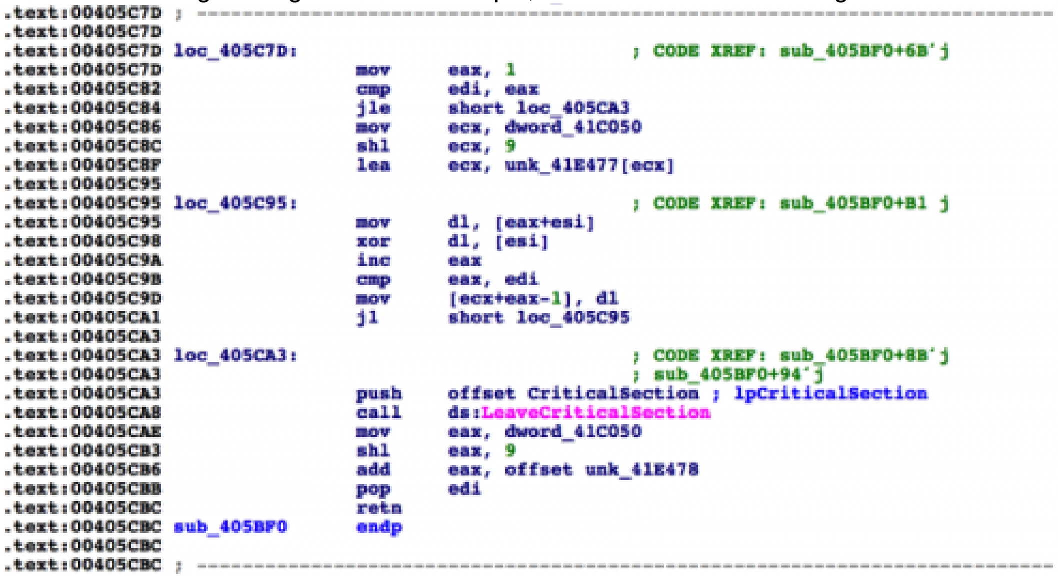
Using IDAPython to Make Your Life Easier: Part 1

posted by: Josh Grunzweig on December 29, 2015 3:00 PM
filed in: Unit 42
tagged: 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

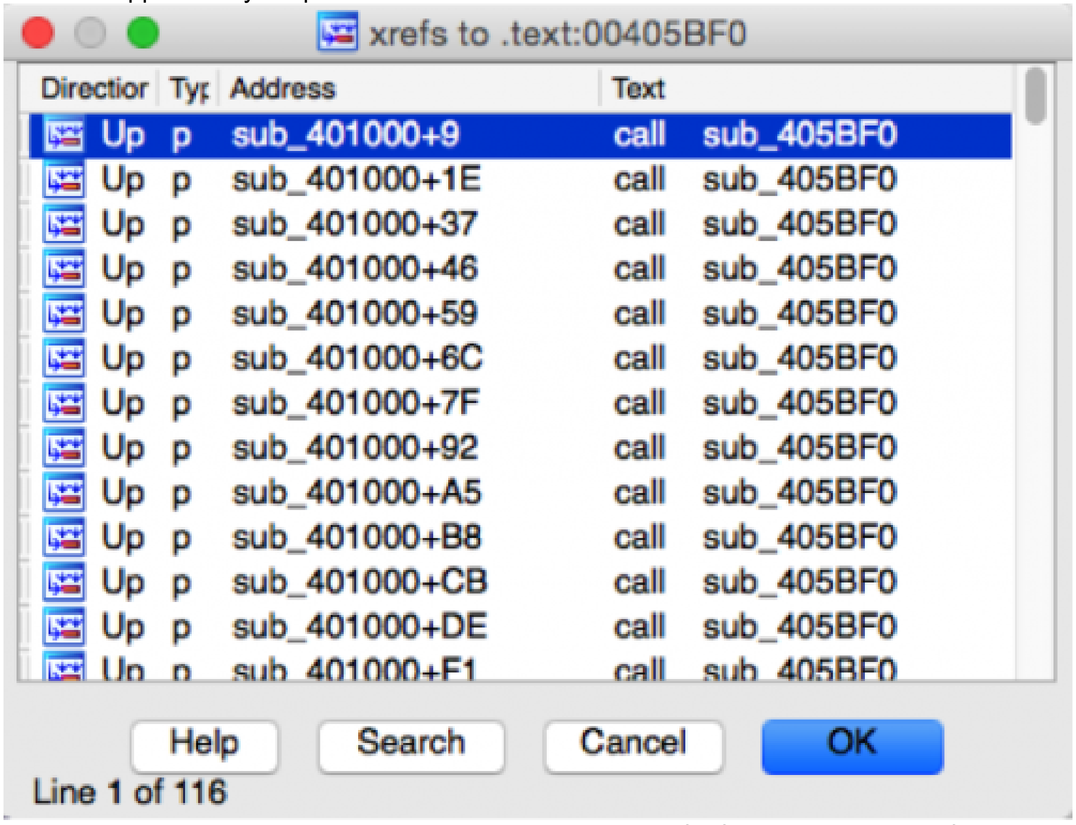
While reverse-engineering a malicious sample, I encountered the following function:



```
.text:00405C7D ; ---------------------------------------------------------------------------
.text:00405C7D
.text:00405C7D loc_405C7D:                             ; CODE XREF: sub_405BF0+6B↑j
.text:00405C7D                 mov     eax, 1
.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
.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
.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:00405CBC                 retn
.text:00405CBC sub_405BF0      endp
.text:00405CBC
.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.


xrefs to .text:00405BF0

Directior	Typ	Address	Text
Up	p	sub_401000+9	call sub_405BF0
Up	p	sub_401000+1E	call sub_405BF0
Up	p	sub_401000+37	call sub_405BF0
Up	p	sub_401000+46	call sub_405BF0
Up	p	sub_401000+59	call sub_405BF0
Up	p	sub_401000+6C	call sub_405BF0
Up	p	sub_401000+7F	call sub_405BF0
Up	p	sub_401000+92	call sub_405BF0
Up	p	sub_401000+A5	call sub_405BF0
Up	p	sub_401000+B8	call sub_405BF0
Up	p	sub_401000+CB	call sub_405BF0
Up	p	sub_401000+DE	call sub_405BF0
Up	p	sub_401000+F1	call sub_405BF0

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.

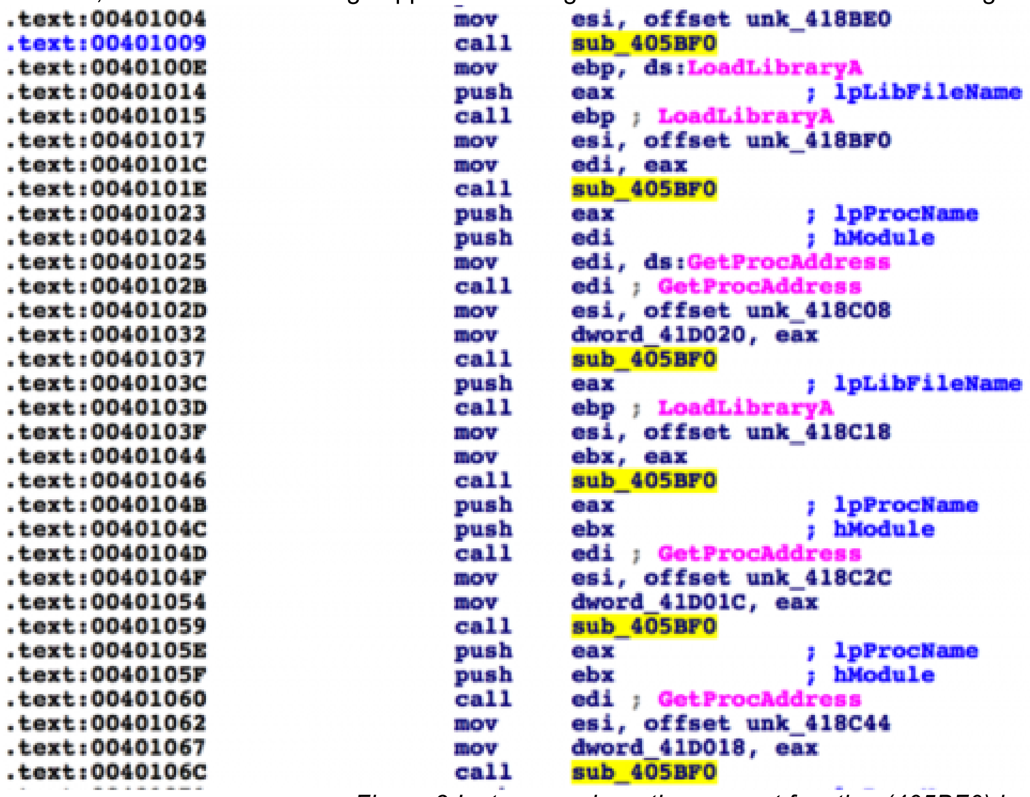

```
.text:00401004                 mov     esi, offset unk_418BE0
.text:00401009                 call    sub_405BF0
.text:0040100E                 mov     ebp, ds:LoadLibraryA
.text:00401014                 push    eax             ; lpLibFileName
.text:00401015                 call    ebp ; LoadLibraryA
.text:00401017                 mov     esi, offset unk_418BF0
.text:0040101C                 mov     edi, eax
.text:0040101E                 call    sub_405BF0
.text:00401023                 push    eax             ; lpProcName
.text:00401024                 push    edi             ; hModule
.text:00401025                 mov     edi, ds:GetProcAddress
.text:0040102B                 call    edi ; GetProcAddress
.text:0040102D                 mov     esi, offset unk_418C08
.text:00401032                 mov     dword_41D020, eax
.text:00401037                 call    sub_405BF0
.text:0040103C                 push    eax             ; lpLibFileName
.text:0040103D                 call    ebp ; LoadLibraryA
.text:0040103F                 mov     esi, offset unk_418C18
.text:00401044                 mov     ebx, eax
.text:00401046                 call    sub_405BF0
.text:0040104B                 push    eax             ; lpProcName
.text:0040104C                 push    ebx             ; hModule
.text:0040104D                 call    edi ; GetProcAddress
.text:0040104F                 mov     esi, offset unk_418C2C
.text:00401054                 mov     dword_41D01C, eax
.text:00401059                 call    sub_405BF0
.text:0040105E                 push    eax             ; lpProcName
.text:0040105F                 push    ebx             ; hModule
.text:00401060                 call    edi ; GetProcAddress
.text:00401062                 mov     esi, offset unk_418C44
.text:00401067                 mov     dword_41D018, eax
.text:0040106C                 call    sub_405BF0
```

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:

I can manually decrypt and rename these obfuscated strings

I can dynamically run this sample and rename the strings as I encounter them

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:

```
def decrypt(data):
  length = len(data)
  c = 1
  o = ""
  while c < length:
    o += chr(ord(data[0]) ^ ord(data[c]))
    c += 1
  return o
```

In testing this code, we get the expected result.

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

```
for addr in XrefsTo(0x00405BF0, flags=0):
  print hex(addr.frm)

Result:
0x401009L
0x40101eL
0x401037L
0x401046L
0x401059L
0x40106cL
0x40107fL
<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:

```
def find_function_arg(addr):
  while True:
    addr = idc.PrevHead(addr)
    if GetMnem(addr) == "mov" and "esi" in GetOpnd(addr, 0):
      print “We found it at 0x%x” % GetOperandValue(addr, 1)
      break

Example Results:
Python>find_function_arg(0x00401009)
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:

```
def get_string(addr):
  out = ""
  while True:
    if Byte(addr) != 0:
      out += chr(Byte(addr))
    else:
      break
    addr += 1
  return out
```

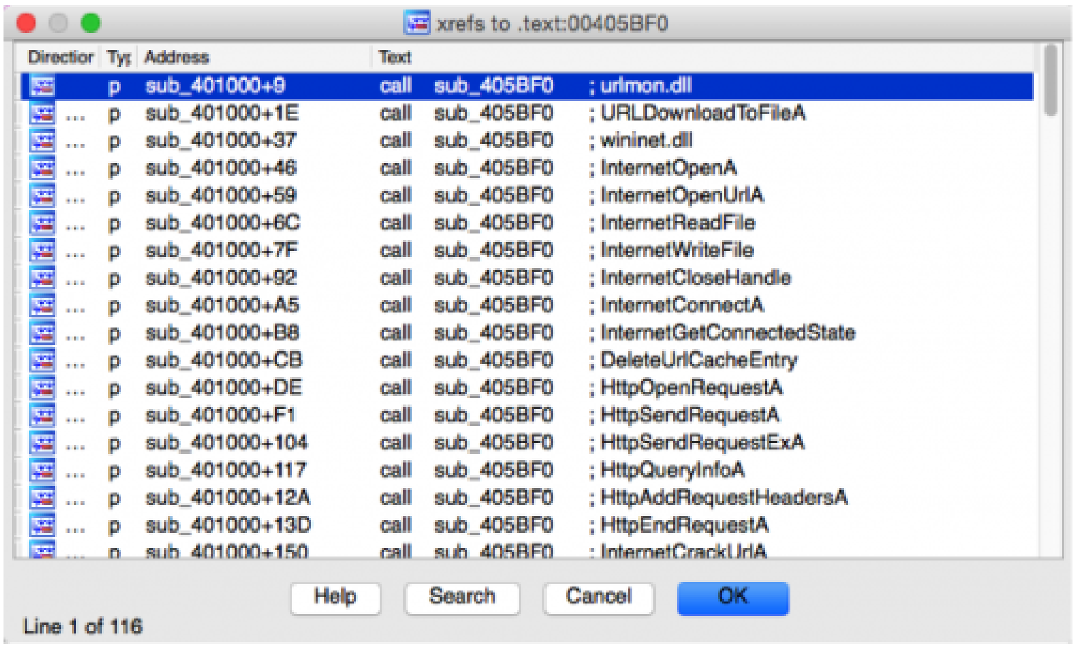
At this point, it’s simply a matter of taking everything we’ve created thus far and putting it together.

```
def find_function_arg(addr):
  while True:
    addr = idc.PrevHead(addr)
    if GetMnem(addr) == "mov" and "esi" in GetOpnd(addr, 0):
      return GetOperandValue(addr, 1)
  return ""

def get_string(addr):
  out = ""
  while True:
    if Byte(addr) != 0:
      out += chr(Byte(addr))
    else:
      break
    addr += 1
  return out

def decrypt(data):
  length = len(data)
  c = 1
  o = ""
  while c < length:
    o += chr(ord(data[0]) ^ ord(data[c]))
    c += 1
  return o

print "[*] Attempting to decrypt strings in malware"
for x in XrefsTo(0x00405BF0, flags=0):
  ref = find_function_arg(x.frm)
  string = get_string(ref)
  dec = decrypt(string)
  print "Ref Addr: 0x%x | Decrypted: %s" % (x.frm, dec)

Results:
[*] Attempting to decrypt strings in malware
Ref Addr: 0x401009 | Decrypted: urlmon.dll
Ref Addr: 0x40101e | Decrypted: URLDownloadToFileA
Ref Addr: 0x401037 | Decrypted: wininet.dll
Ref Addr: 0x401046 | Decrypted: InternetOpenA
Ref Addr: 0x401059 | Decrypted: InternetOpenUrlA
Ref Addr: 0x40106c | Decrypted: InternetReadFile
<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:

```
MakeComm(x.frm, dec)
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.

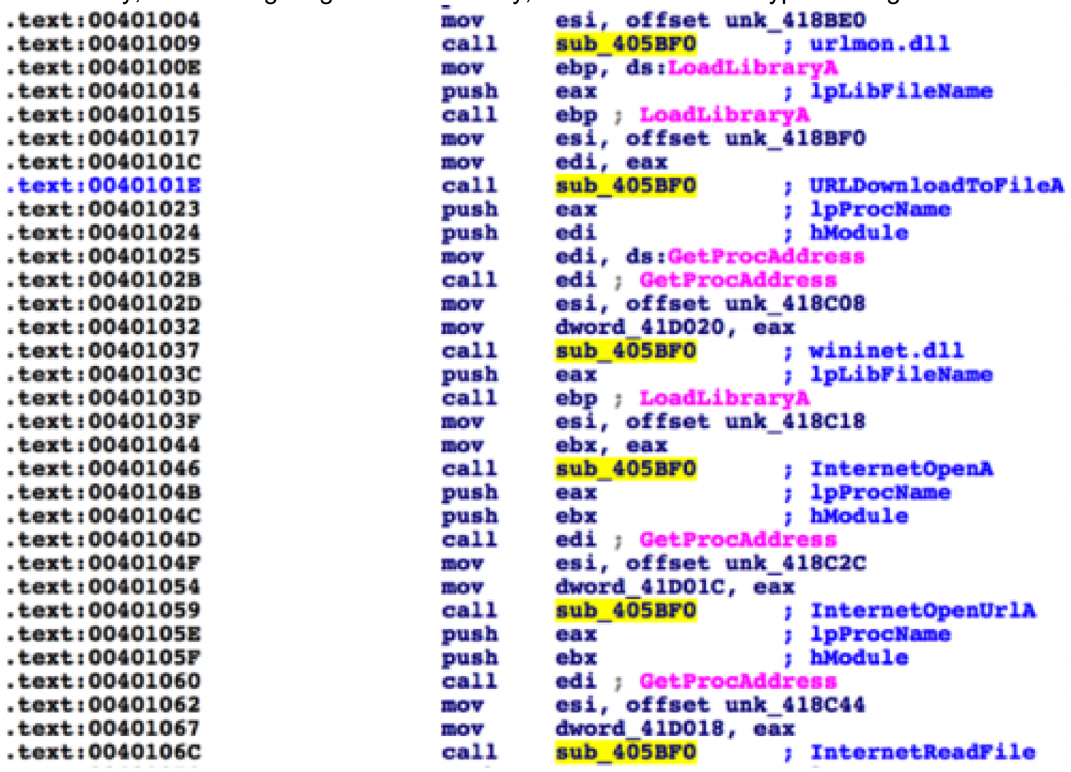


Direction	Typ	Address	Text		
	p	sub_401000+9	call	sub_405BF0	; urlmon.dll
...	p	sub_401000+1E	call	sub_405BF0	; URLDownloadToFileA
...	p	sub_401000+37	call	sub_405BF0	; wininet.dll
...	p	sub_401000+46	call	sub_405BF0	; InternetOpenA
...	p	sub_401000+59	call	sub_405BF0	; InternetOpenUrlA
...	p	sub_401000+6C	call	sub_405BF0	; InternetReadFile
...	p	sub_401000+7F	call	sub_405BF0	; InternetWriteFile
...	p	sub_401000+92	call	sub_405BF0	; InternetCloseHandle
...	p	sub_401000+A5	call	sub_405BF0	; InternetConnectA
...	p	sub_401000+B8	call	sub_405BF0	; InternetGetConnectedState
...	p	sub_401000+CB	call	sub_405BF0	; DeleteUrlCacheEntry
...	p	sub_401000+DE	call	sub_405BF0	; HttpOpenRequestA
...	p	sub_401000+F1	call	sub_405BF0	; HttpSendRequestA
...	p	sub_401000+104	call	sub_405BF0	; HttpSendRequestExA
...	p	sub_401000+117	call	sub_405BF0	; HttpQueryInfoA
...	p	sub_401000+12A	call	sub_405BF0	; HttpAddRequestHeadersA
...	p	sub_401000+13D	call	sub_405BF0	; HttpEndRequestA
...	p	sub_401000+150	call	sub_405BF0	; InternetCrackUrlA

Line 1 of 116

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

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



```
.text:00401004                 mov     esi, offset unk_418BE0
.text:00401009                 call    sub_405BF0      ; urlmon.dll
.text:0040100E                 mov     ebp, ds:LoadLibraryA
.text:00401014                 push    eax             ; lpLibFileName
.text:00401015                 call    ebp ; LoadLibraryA
.text:00401017                 mov     esi, offset unk_418BF0
.text:0040101C                 mov     edi, eax
.text:0040101E                 call    sub_405BF0      ; URLDownloadToFileA
.text:00401023                 push    eax             ; lpProcName
.text:00401024                 push    edi             ; hModule
.text:00401025                 mov     edi, ds:GetProcAddress
.text:0040102B                 call    edi ; GetProcAddress
.text:0040102D                 mov     esi, offset unk_418C08
.text:00401032                 mov     dword_41D020, eax
.text:00401037                 call    sub_405BF0      ; wininet.dll
.text:0040103C                 push    eax             ; lpLibFileName
.text:0040103D                 call    ebp ; LoadLibraryA
.text:0040103F                 mov     esi, offset unk_418C18
.text:00401044                 mov     ebx, eax
.text:00401046                 call    sub_405BF0      ; InternetOpenA
.text:0040104B                 push    eax             ; lpProcName
.text:0040104C                 push    ebx             ; hModule
.text:0040104D                 call    edi ; GetProcAddress
.text:0040104F                 mov     esi, offset unk_418C2C
.text:00401054                 mov     dword_41D01C, eax
.text:00401059                 call    sub_405BF0      ; InternetOpenUrlA
.text:0040105E                 push    eax             ; lpProcName
.text:0040105F                 push    ebx             ; hModule
.text:00401060                 call    edi ; GetProcAddress
.text:00401062                 mov     esi, offset unk_418C44
.text:00401067                 mov     dword_41D018, eax
.text:0040106C                 call    sub_405BF0      ; InternetReadFile
```

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.

Post Your Comment

	Name *
	Email *
	Website

Post Comment

Home
Government
Partners
Unit 42 Threat Intelligence
Technical Documentation
Advanced Endpoint Protection

unit42

Get Updates
Sign up to receive the latest news, cyber threat intelligence and research from Unit 42.

Business Email

Submit

Subscribe to the Research Center Blog

Subscribe

Categories & Archives

Select a Category
Select a Month
More →

Recent Posts

Using IDAPython to Make Your Life Easier: Part 1 posted by Josh Grunzweig on December 29, 2015
2016 Predictions: What's In Store for Cybersecurity? posted by Anna Lough on December 29, 2015
Is the End the Beginning: Where Are We in The Endpoint Journey? posted by Greg Day on December 29, 2015
2016 Prediction #14: Six Cybersecurity Predictions for Asia-Pacific posted by Sean Duca on December 28, 2015
Palo Alto Networks News of the Week – December 26 posted by Anna Lough on December 26, 2015
More →

About Palo Alto Networks

Palo Alto Networks is the network security company. Our innovative platform allows enterprises, service providers, and government entities to secure their networks and safely enable the increasingly complex and rapidly growing number of applications running on their networks.
The core of Palo Alto Networks' platform is our next-generation firewall, which delivers application, user, and content visibility and control integrated within the firewall through its proprietary hardware and software architecture. Palo Alto Networks products and services can address a broad range of network security requirements, from the datacenter to the network perimeter, as well as the distributed enterprise, which includes branch offices and a growing number of mobile devices.

FOLLOW US

Facebook
Twitter
Linked In
You Tube

Learn More

Firewalls
VPN
Malware
Intrusion Prevention System
Intrusion Detection System

Denial of Service Attack
Security Policy
Network Security
Data Center
1.866.320.4788
Privacy Policy
Legal Notices
Site Index
Subscriptions