Analyzing a Magnitude EK Appx Package Dropping Magniber

forensicitguy.github.io/analyzing-magnitude-magniber-appx/

January 2, 2022

By *Tony Lambert*

Posted *2022-01-02* Updated *2022-03-28* *10 min* read

In this post I'll work through analyzing an AppX package from Magnitude Exploit Kit that drops Magniber. This adventure comes courtesy of a tweet from @JAMESWT_MHT:

> Some #Magniber sampleshttps://t.co/6XaMq0X2QH https://t.co/wWef0eSk2o
>
> — JAMESWT (@JAMESWT_MHT) January 1, 2022

This caught my interest because AppX packages have gotten some mileage as droppers lately courtesy of Bazar and Emotet.

- https://news.sophos.com/en-us/2021/11/11/bazarloader-call-me-back-attack-abuses-windows-10-apps-mechanism/
- https://redcanary.com/blog/intelligence-insights-december-2021/

If you want to play along from home, the file I'm analyzing is here: https://bazaar.abuse.ch/sample/da1729efaaa590d66f46d388680ed5b1b956246ababd277e7cdd14f90fbf60fa/

Analyzing the AppX Package

To start off, let's get a handle on what kind of file an AppX package is. We can do this using `file`.

```
remnux@remnux:~/cases/magnitude/update$ file
edge_update.appx
edge_update.appx: Zip archive data, at least v4.5 to
extract
```

The `file` command says the magic bytes for the file correspond to a zip archive. This is common with application or package archives like AppX, JARs, and more. If we want more confirmation we can always look at the first few bytes with `hexdump` and `head`.

```
remnux@remnux:~/cases/magnitude/update$ hexdump -C edge_update.appx |
head
00000000  50 4b 03 04 2d 00 08 00  00 00 f8 6e 9d 53 00 00
|PK..-......n.S..|
00000010  00 00 00 00 00 00 00 00  00 00 26 00 00 00 49 6d
|..........&...Im|
00000020  61 67 65 73 2f 53 71 75  61 72 65 31 35 30 78 31
|ages/Square150x1|
00000030  35 30 4c 6f 67 6f 2e 73  63 61 6c 65 2d 31 35 30
|50Logo.scale-150|
00000040  2e 70 6e 67 89 50 4e 47  0d 0a 1a 0a 00 00 00 0d
|.png.PNG........|
00000050  49 48 44 52 00 00 00 e1  00 00 00 e1 08 06 00 00
|IHDR............|
00000060  00 3e b3 d2 7a 00 00 00  09 70 48 59 73 00 00 0e
|.>..z....pHYs...|
00000070  c3 00 00 0e c3 01 c7 6f  a8 64 00 00 71 fc 49 44
|.......o.d..q.ID|
00000080  41 54 78 9c ec bd 77 90  25 c7 79 27 f8 65 99 e7
|ATx...w.%.y'.e..|
00000090  db 9b e9 ee f1 33 98 19  0c 06 84 77 04 41 18 92
|.....3.....w.A..|
```

Yup, looks like a zip file based on `50 4b 03 04`! That means we can unpack the archive using `unzip`.

```
remnux@remnux:~/cases/magnitude/update$ unzip edge_update.appx
Archive:  edge_update.appx
 extracting: Images/Square150x150Logo.scale-150.png
 extracting: Images/Wide310x150Logo.scale-150.png
 extracting: Images/SmallTile.scale-150.png
 extracting: Images/LargeTile.scale-150.png
 extracting: Images/BadgeLogo.scale-150.png
 extracting: Images/SplashScreen.scale-150.png
 extracting: Images/StoreLogo.scale-150.png
 extracting: Images/Square44x44Logo.targetsize-32.png
 extracting: Images/Square44x44Logo.altform-unplated_targetsize-32.png
 extracting: Images/Square44x44Logo.scale-150.png
 extracting: Images/Square44x44Logo.altform-lightunplated_targetsize-
32.png
  inflating: eediwjus/eediwjus.exe
  inflating: eediwjus/eediwjus.dll
  inflating: resources.pri
  inflating: AppxManifest.xml
  inflating: AppxBlockMap.xml
  inflating: [Content_Types].xml
  inflating: AppxMetadata/CodeIntegrity.cat
  inflating: AppxSignature.p7x
```

With the archive unzipped, we can focus on significant files within the package. These are:

- AppxManifest.xml (list of properties and components used by the AppX package)
- AppxSignature.p7x (AppX Signature Object, contains code signatures for AppX Package)

- eediwjus/eediwjus.exe (non-default content that is likely executable)
- eediwjus/eediwjus.dll (non-default content that is likely executable)

First, we can look at the AppxManifest.xml file. I’ve included the points of interest below.

```
<?xml version="1.0" encoding="utf-8"?>
<Package xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10"
xmlns:uap="http://schemas.microsoft.com/appx/manifest/uap/windows10" xmlns:re
scap="http://schemas.microsoft.com/appx/manifest/foundation/windows10/restrictedcapabili
ties" IgnorableNamespaces="uap rescap build" xmlns:build="http://schem
as.microsoft.com/developer/appx/2015/build">
  <Identity Name="3669e262-ec02-4e9d-bcb4-3d008b4afac9" Publisher="CN=Foresee Consulting
Inc., O=Foresee Consulting Inc., L=North York, S=Ontario, C=CA, SERIA
LNUMBER=1004913-1, OID.1.3.6.1.4.1.311.60.2.1.3=CA, OID.2.5.4.15=Private Organization"
Version="96.0.1072.0" ProcessorArchitecture="neutral" />
  <Properties>
    <DisplayName>Edge Update</DisplayName>
    <PublisherDisplayName>Microsoft Inc</PublisherDisplayName>
    <Logo>Images\StoreLogo.png</Logo>
  </Properties>

...

  <Applications>
    <Application Id="App" Executable="eediwjus\eediwjus.exe"
EntryPoint="Windows.FullTrustApplication">
      ...
    </Application>
  </Applications>
  <Capabilities>
    <Capability Name="internetClient" />
    <rescap:Capability Name="runFullTrust" />
  </Capabilities>
</Package>
```

First, let's take a look at the Identity and Properties sections. Identity contains code signature information that should theoretically be included within the AppxSignature.p7x file. The Properties section contains metadata the Windows Store/Universal Windows App interface uses to identify the app. From the name `Edge Update` and publisher name `Microsoft Inc`, it appears the malware wants to masquerade as a Microsoft Edge browser update. Note how there is no link or control between the publisher display name and the actual signing identity. This is a major problem for victims trying to be sure of themselves.

The Application section identifies the EXE that will execute when the package is installed and run. In this sample, the EXE is `eediwjus.exe`. In the package content there is also a DLL, but that isn't mentioned in the manifest. A possibility to explore might be that the EXE uses content from the DLL for execution.

Finally, the Capabilities section shows the app will execute with `internetClient` and `runFullTrust` capabilities. Documented by Microsoft, these capabilities just mean the app can download stuff from the Internet. Now we can jump into the executable content, the EXE file.

Analyzing the Application Executable

The EXE has these hashes:

```
filepath:                          eediwjus.exe
md5:                               3439bbe95df314d390cc4862cdad94fd
sha1:                              92429885d54a05ed87a5c14d34aa504c28ea8b54
sha256:
ad4f74c0c3ac37e6f1cf600a96ae203c38341d263dbac0741e602686794c4f5a
ssdeep:                            48:6/yaz1YKkikwFJSDq6tPRqBHwOul2a3iq:yz1fkigtJkGYK
imphash:                           f34d5f2d4577ed6d9ceec516c1f5a744
```

Note the import table hash starting with `f34d`. That specific import table hash commonly appears with .NET binaries, so if you pivot on it in VT or other tools, you'll find a lot of .NET. Using `Detect It Easy` in REMnux, we can confirm the executable is a .NET binary.

```
remnux@remnux:~/cases/magnitude/update/eediwjus$ diec
eediwjus.exe
filetype: PE32
arch: I386
mode: 32-bit
endianess: LE
type: GUI
  library: .NET(v4.0.30319)[-]
  linker: Microsoft Linker(11.0)[GUI32]
```

So let’s take a peek with `floss` from Mandiant to see if there are signs of obfuscation. There aren’t any signs of obfuscation like randomized, high-entropy strings, but we do get some interesting strings.

```
mscorlib
System
Object
mhjpfzvitta
Main
.ctor
lpBuffer
args
System.Runtime.Versioning
TargetFrameworkAttribute
System.Security.Permission
s
SecurityPermissionAttribut
e
SecurityAction
System.Runtime.CompilerSer
vices
CompilationRelaxationsAttr
ibute
RuntimeCompatibilityAttrib
ute
System.Runtime.InteropServ
ices
DllImportAttribute
eediwjus.dll
```

Sure enough, the EXE references the DLL in the same folder, and it includes the string `DllImportAttribute`. This is a good sign that the EXE will load an unmanaged DLL and call an export from it. Unobfuscated .NET code is usually pretty easy to decompile from bytecode form into source, so we can give that a shot with `ilspycmd`. If you're on Windows you can also use `ILSpy` or `DNSpy`. The result is a pretty brief source file:

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;
using System.Runtime.Versioning;
using System.Security;
using System.Security.Permissions;

[assembly: TargetFramework(".NETFramework,Version=v4.5", FrameworkDisplayName = ".NET
Framework 4.5")]
[assembly: CompilationRelaxations(8)]
[assembly: RuntimeCompatibility(WrapNonExceptionThrows = true)]
[assembly: SecurityPermission(8, SkipVerification = true)]
[assembly: AssemblyVersion("0.0.0.0")]
[module: UnverifiableCode]
namespace eediwjus
{
	public class eediwjus
	{
		[DllImport("eediwjus.dll")]
		private static extern void mhjpfzvitta(uint lpBuffer);

		private static void Main(string[] args)
		{
			uint lpBuffer = 5604u;
			mhjpfzvitta(lpBuffer);
		}
	}
}
```

The entry point for the program is the `Main` function inside the `eediwjus` class. The `DllImport` code imports the function `mhjpfzvitta()` from the DLL and calls it with the argument `lpBuffer`. That argument contains an unsigned integer value of 5604. `lpBuffer` appears loads of times in Microsoft documentation around Windows calls like `VirtualAlloc` and others that need a buffer of memory for operation. It stands to reason that `lpBuffer` here might correspond to some form of a memory management call.

Analyzing the Magniber DLL

The DLL has these hashes:

```
filepath:                          eediwjus.dll
md5:                               e7e4878847d31c4de301d3edf7378ecb
sha1:                              a93d0f59b3374c6d3669a5872d44515f056e9dbf
sha256:
f423bd6daae6c8002acf5c203267e015f7beb4c52ed54a78789dd86ab35e46c6
ssdeep:
96:qUG6xykl2J6lc5irN3qjNu47Ru/8IAgecgKDD:qsQMl0u3qjA47RuZAhk
```

Our `pehash` command didn't find an import table hash, so that's interesting. There may not be an import table in this binary or it might be mangled. We can take a look using the Python `pefile` library.

```
remnux@remnux:~/cases/magnitude/update/eediwjus$ python3
Python 3.8.10 (default, Nov 26 2021, 20:14:08)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> import pefile
>>> bin = pefile.PE('eediwjus.dll')
>>> bin.get_imphash()
''
>>> bin.get_rich_header_hash()
''
```

Sure enough, the binary doesn’t seem to have an import table hash or rich header hash. Maybe those parts don’t exist? We can confirm with `pefile` again.

```
>>> for directory in
bin.OPTIONAL_HEADER.DATA_DIRECTORY:
...     print(directory)
...
[IMAGE_DIRECTORY_ENTRY_EXPORT]
0x148      0x0   VirtualAddress:
0x2000
0x14C      0x4   Size:                          0x4B
[IMAGE_DIRECTORY_ENTRY_IMPORT]
0x150      0x0   VirtualAddress:                0x0
0x154      0x4   Size:                          0x0

...

>>> bin.RICH_HEADER
>>>
```

Sure enough, the import table is apparently empty and no rich header exists for the binary. This is slightly unusual, so let’s see if we can run some more commands to find capabilities before jumping further into analysis.

The Mandiant tools `floss` and `capa` yield nothing significant.

```
+-----------+------------------------------------------------------------
----+
| md5       | e7e4878847d31c4de301d3edf7378ecb
|
| sha1      | a93d0f59b3374c6d3669a5872d44515f056e9dbf
|
| sha256    |
f423bd6daae6c8002acf5c203267e015f7beb4c52ed54a78789dd86ab35e46c6 |
| path      | eediwjus.dll
|
+-----------+------------------------------------------------------------
----+

no capabilities found
```

Yara tells us more of what we already know.

```
remnux@remnux:~/cases/magnitude/update/eediwjus$ yara-rules
eediwjus.dll
IsPE64 eediwjus.dll
IsDLL eediwjus.dll
IsWindowsGUI eediwjus.dll
ImportTableIsBad eediwjus.dll
HasModified_DOS_Message eediwjus.dll
```

A `pedump` command gets us some export info. You could also get this with `pefile` in Python, I just like this output better.

```
=== EXPORTS ===

# module "eediwjus.dll"
# flags=0x0  ts="2021-12-29 10:55:45"  version=0.0
ord_base=1
# nFuncs=1  nNames=1

  ORD ENTRY_VA  NAME
    1     1f74  mhjpfzvitta
```

The export `mhjpfzvitta()` jives with what we expect coming from the EXE previously seen. This is probably our best entry point to examine the DLL.

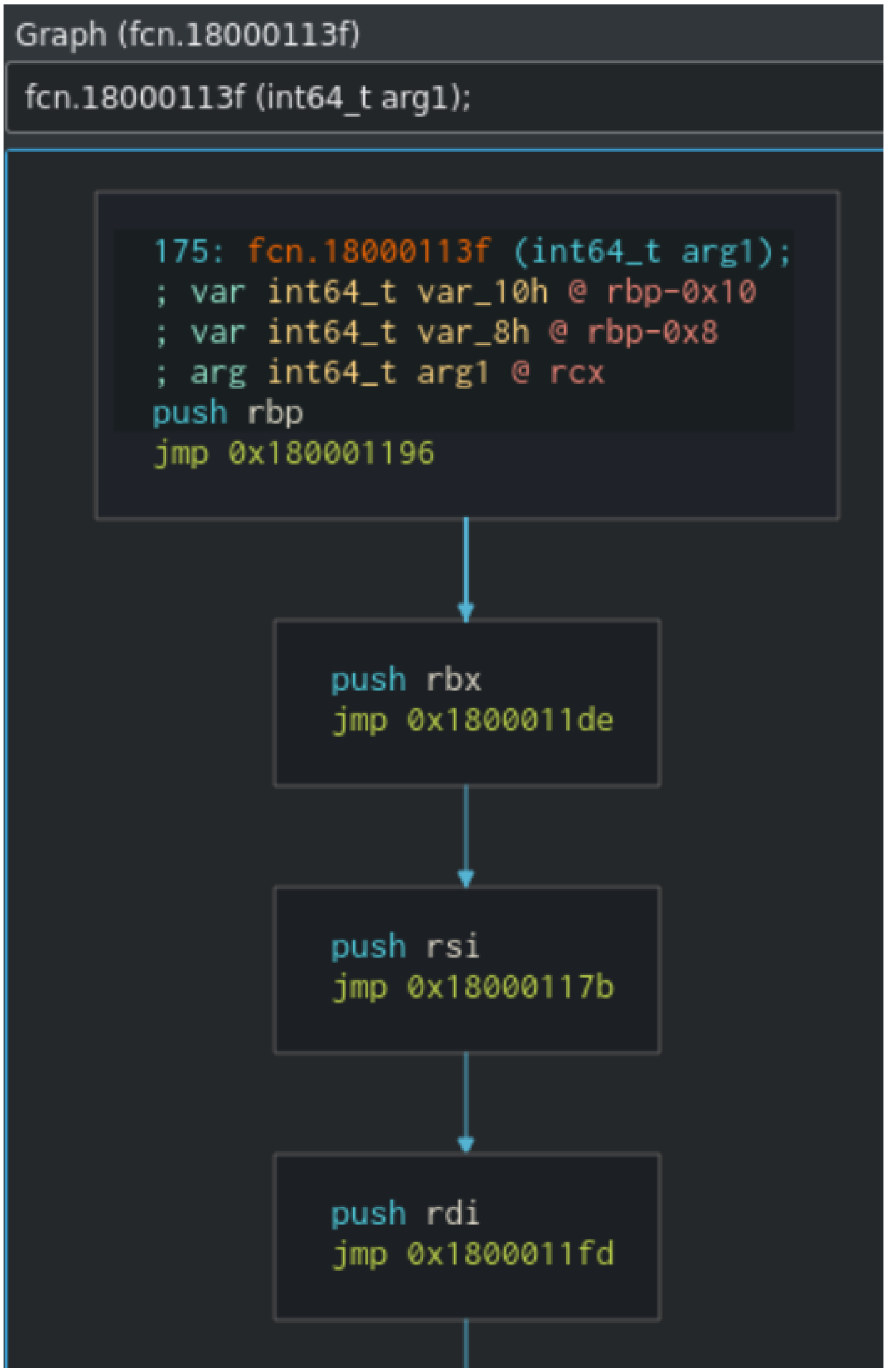
Getting Dirty In Assembly

I usually work with Ghidra, but Cutter seemed to have a better representation of the assembly for this binary.

The entry point export `mhjpfzvitta()` is fairly brief.

```
6: mhjpfzvitta (int64_t arg1);
; arg int64_t arg1 @ rcx
0x180001f74      call fcn.18000113f
0x180001f79      ret
```

The entry point immediately calls a function at offset `18000113f` and returns. Once we go to look at the assembly for that function, we see quite a wild execution graph.



```
fcn.18000113f (int64_t arg1);
```

```
175: fcn.18000113f (int64_t arg1);
; var int64_t var_10h @ rbp-0x10
; var int64_t var_8h @ rbp-0x8
; arg int64_t arg1 @ rcx
push rbp
jmp 0x180001196
```

```
push rbx
jmp 0x1800011de
```

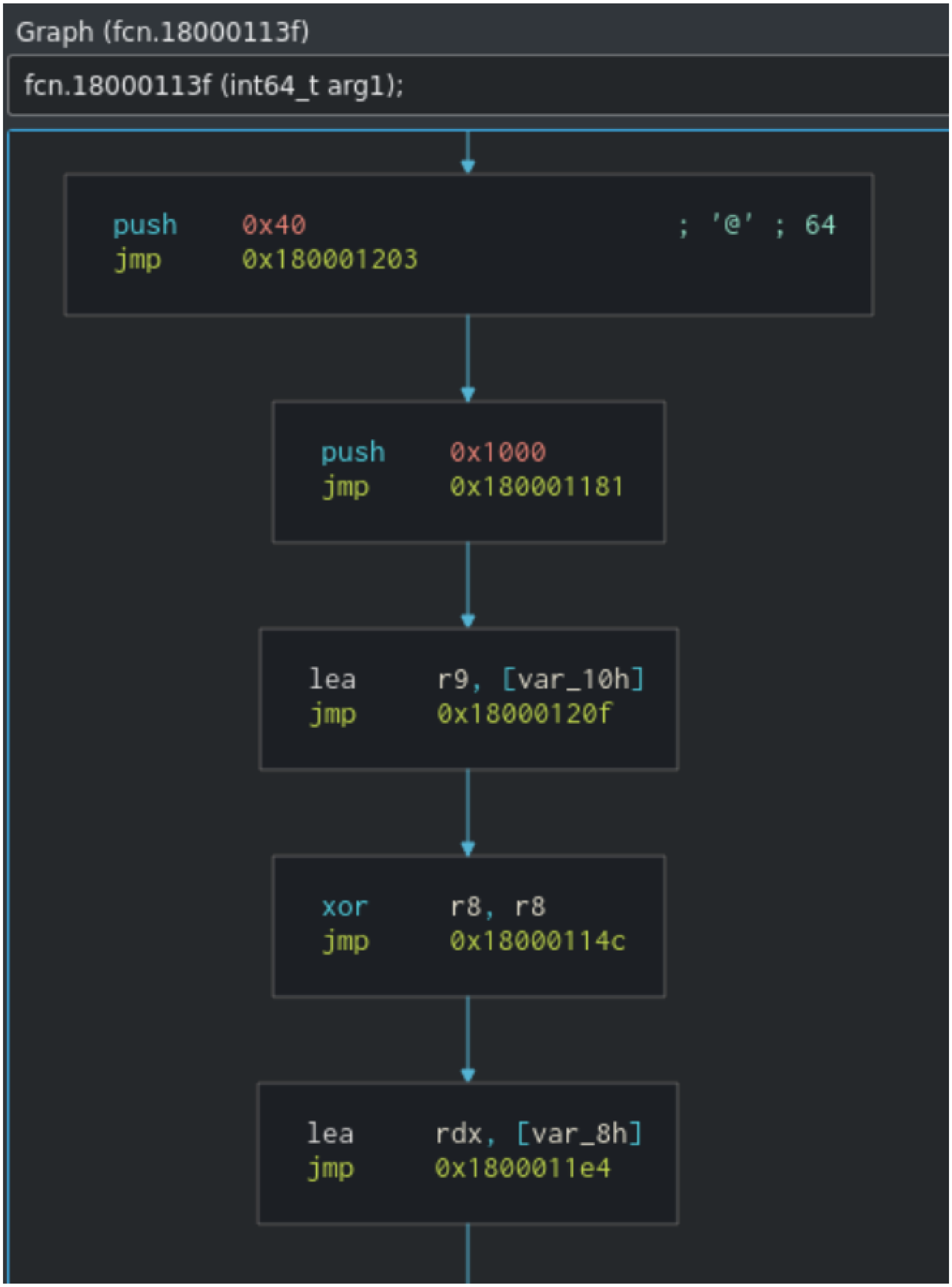
```
push rsi
jmp 0x18000117b
```

```
push rdi
jmp 0x1800011fd
```

Once entering the function, the sample contains loads of `jmp` instructions that cause execution to bounce around to various points of the binary. This makes it hard for analysts to follow execution, and eventually we see some more evidence of suspicious activity in decompiled code.

```
undefined8 fcn.180001f8e(int64_t
arg1)
{
    syscall();
    return 0x18;
}
```

Since the sample doesn’t have an import table, it’s relying on manual syscall calls like one to `0x18` for `NtAllocateVirtualMemory`. Avast saw this with Magniber in the past, alongside the `jmp` obfuscation.



```
fcn.18000113f (int64_t arg1);

push    0x40                      ; '@' ; 64
jmp     0x180001203

push    0x1000
jmp     0x180001181

lea     r9, [var_10h]
jmp     0x18000120f

xor     r8, r8
jmp     0x18000114c

lea     rdx, [var_8h]
jmp     0x1800011e4
```

While I'm not yet skilled enough to tear much more out of the binary through static analysis, my eye was caught by one section of code that pushes `0x40` and `0x1000` to registers. These two values sometimes pop up when malware calls `VirtualAlloc`. 0x40 refers to

`PAGE_EXECUTE_READWRITE` protection and 0x1000 refers to `MEM_COMMIT`. Since these values popped up in the sample, we can hypothesize that the sample may inject or unpack material into a memory space.

How do we know it’s Magniber?

I didn’t have luck getting Yara rules for Magniber to match this sample, so the best references I have right now are the tweet from @@JAMESWT_MHT and the blog post from Avast showing similar `jmp` obfuscation and syscall references.

Thanks for reading!