Anatomy of Cobalt Strike's DLL Stager

blog.nviso.eu/2021/04/26/anatomy-of-cobalt-strike-dll-stagers/

April 26, 2021

NVISO recently monitored a targeted campaign against one of its customers in the financial sector. The attempt was spotted at its earliest stage following an employee's report concerning a suspicious email. While no harm was done, we commonly identify any related indicators to ensure additional monitoring of the actor.

The reported email was an application for one of the company's public job offers and attempted to deliver a malicious document. What caught our attention, besides leveraging an actual job offer, was the presence of execution-guardrails in the malicious document. Analysis of the document uncovered the intention to persist a Cobalt Strike stager through Component Object Model Hijacking.

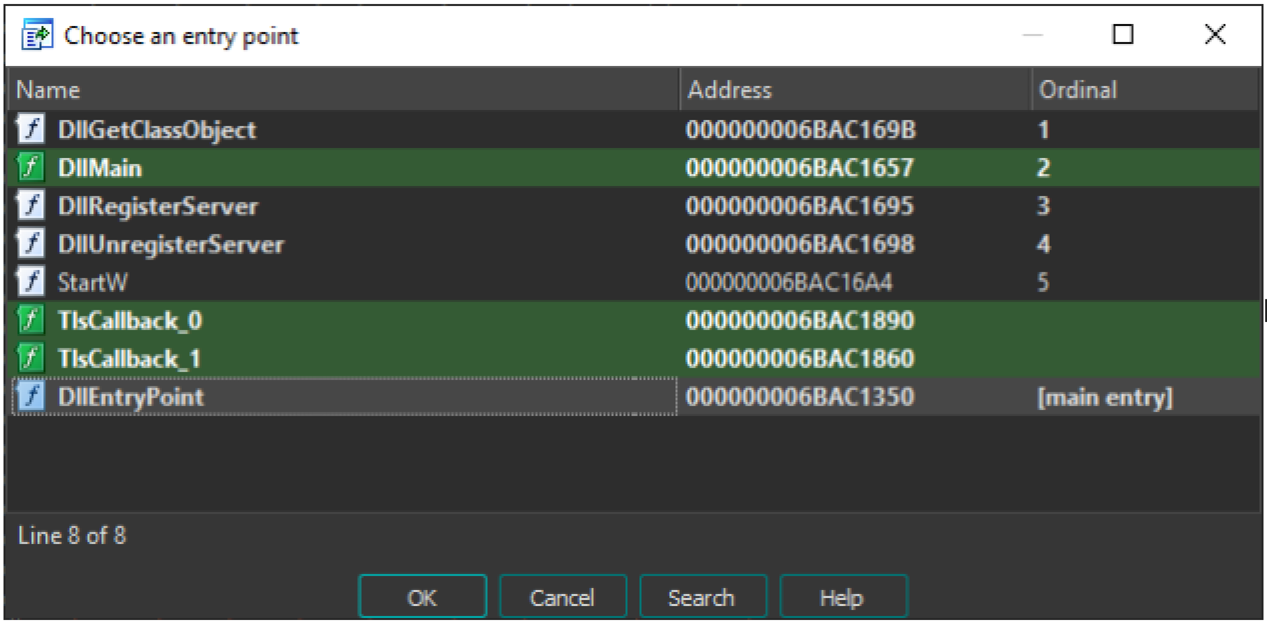
During my free time I enjoy analyzing samples NVISO spots in-the-wild, and hence further dissected the Cobalt Strike DLL payload. This blog post will cover the payload's anatomy, design choices and highlight ways to reduce both log footprint and time-to-shellcode.

Execution Flow Analysis

To understand how the malicious code works we have to analyze its behavior from start to end. In this section, we will cover the following flows:

1. The initial execution through `DllMain`.
2. The sending of encrypted shellcode into a named pipe by `WriteBufferToPipe`.
3. The pipe reading, shellcode decryption and execution through `PipeDecryptExec`.

As previously mentioned, the malicious document's DLL payload was intended to be used as a COM in-process server. With this knowledge, we can already expect some known entry points to be exposed by the DLL.



List of available entry points as displayed in IDA.

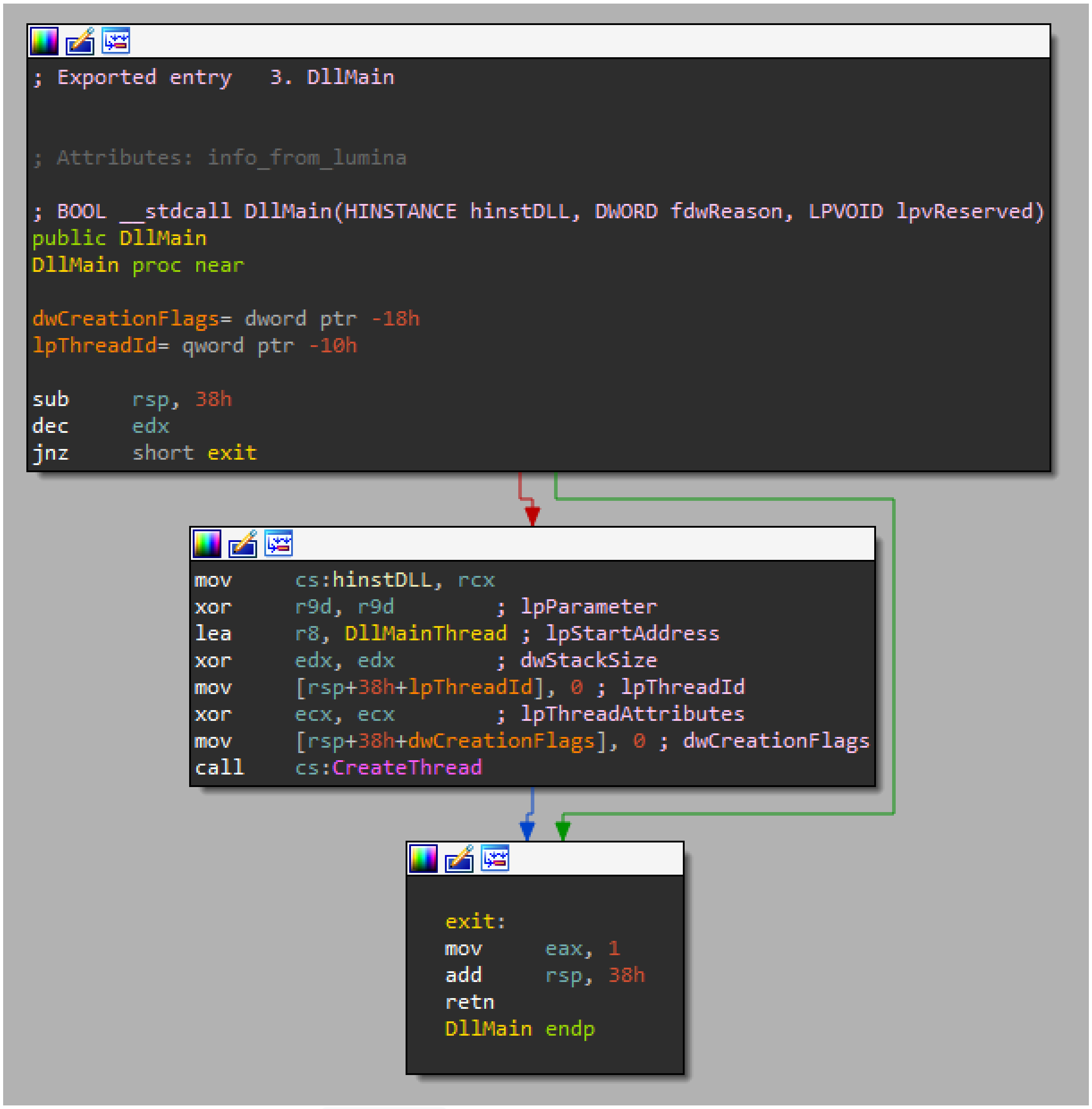
While technically the malicious execution can occur in any of the 8 functions, malicious code commonly resides in the `DllMain` function given, besides TLS callbacks, it is the function most likely to execute.

> `DllMain`: An optional entry point into a dynamic-link library (DLL). When the system starts or terminates a process or thread, it calls the entry-point function for each loaded DLL using the first thread of the process. The system also calls the entry-point function for a DLL when it is loaded or unloaded using the `LoadLibrary` and `FreeLibrary` functions.
>
> *docs.microsoft.com/en-us/windows/win32/dlls/dllmain*

Throughout the following analysis functions and variables have been renamed to reflect their usage and improve clarity.

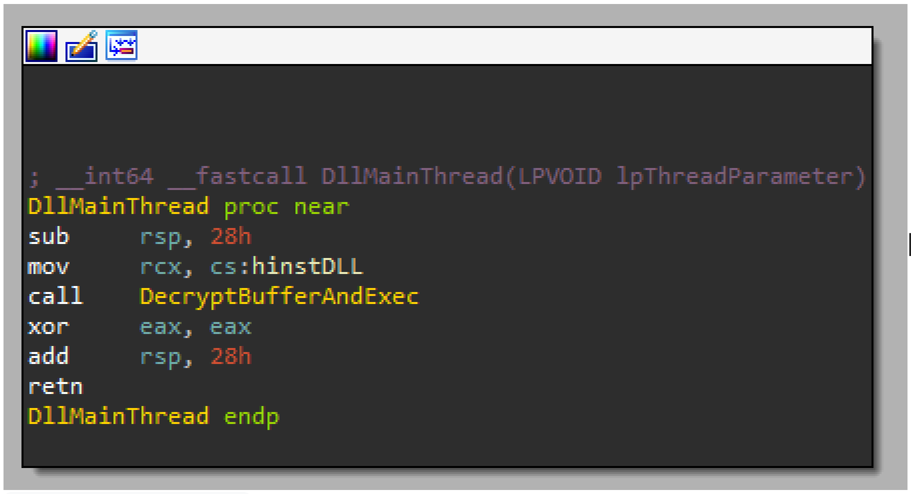
The `DllMain` Entry Point

As can be seen in the following capture, the `DllMain` function simply executes another function by creating a new thread. This threaded function we named `DllMainThread` is executed without any additional arguments being provided to it.



Graphed disassembly of `DllMain` .

Analyzing the `DllMainThread` function uncovers it is an additional wrapper towards what we will discover is the malicious payload's decryption and execution function (called `DecryptBufferAndExec` in the capture).



Disassembly of `DllMainThread`.

By going one level deeper, we can see the start of the malicious logic. Analysts experienced with Cobalt Strike will recognize the well-known `MSSE-%d-server` pattern.

```
; __int64 DecryptBufferAndExec()
DecryptBufferAndExec proc near

dot= dword ptr -48h
BackSlash1= qword ptr -40h
p1= dword ptr -38h
i= dword ptr -30h
p2= dword ptr -28h
e= dword ptr -20h
BackSlash2= dword ptr -18h
TickCount= dword ptr -10h

sub     rsp, 68h
call    cs:GetTickCount
mov     ecx, 26AAh
xor     edx, edx
mov     r9d, '\'
div     ecx
mov     [rsp+68h+BackSlash2], '\'
mov     [rsp+68h+e], 'e'
mov     [rsp+68h+p2], 'p'
mov     [rsp+68h+i], 'i'
mov     [rsp+68h+p1], 'p'
mov     dword ptr [rsp+68h+BackSlash1], '\'
mov     [rsp+68h+dot], '.'
mov     r8d, '\'
lea     rcx, PipeName   ; Buffer
mov     [rsp+68h+TickCount], edx
lea     rdx, PipeFormat ; "%c%c%c%c%c%c%c%c%cMSSE-%d-server"
call    sprintf_10
xor     ecx, ecx        ; lpThreadAttributes
xor     r9d, r9d        ; lpParameter
xor     edx, edx        ; dwStackSize
mov     [rsp+68h+BackSlash1], 0 ; lpThreadId
lea     r8, WriteBufferToPipeThread ; lpStartAddress
mov     [rsp+68h+dot], 0 ; dwCreationFlags
call    cs:CreateThread
xor     ecx, ecx
add     rsp, 68h
jmp     PipeDecryptExec
DecryptBufferAndExec endp
```

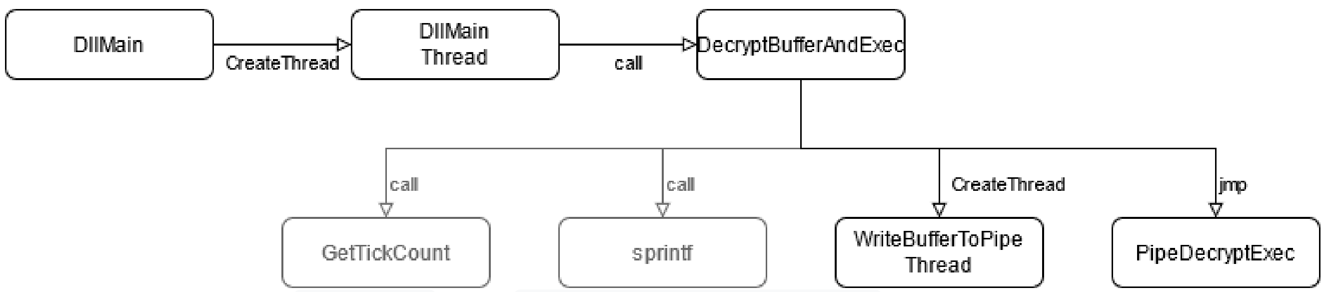
Disassembly of `DecryptBufferAndExec`.

A couple of things occur in the above code:

1. The sample starts by retrieving the tick count through `GetTickCount` and then divides it by `0x26AA`. While obtaining a tick count is often a time measurement, the next operation solely uses the divided tick as a random number.

2. The sample then proceeds to call a wrapper around an implementation of the `sprintf` function. Its role is to format a string into the `PipeName` buffer. As can be observed, the formatted string will be `\\.\pipe\MSSE-%d-server` where `%d` will be the result computed in the previous division (e.g.: `\\.\pipe\MSSE-1234-server`). This pipe's format is a well-documented Cobalt Strike indicator of compromise.
3. With the pipe's name defined in a global variable, the malicious code creates a new thread to run `WriteBufferToPipeThread` . This function will be the next one we will analyze.
4. Finally, while the new thread is running, the code jumps to the `PipeDecryptExec` routine.

So far, we had a linear execution from our `DllMain` entry point until the `DecryptBufferAndExec` function. We could graph the flow as follows:

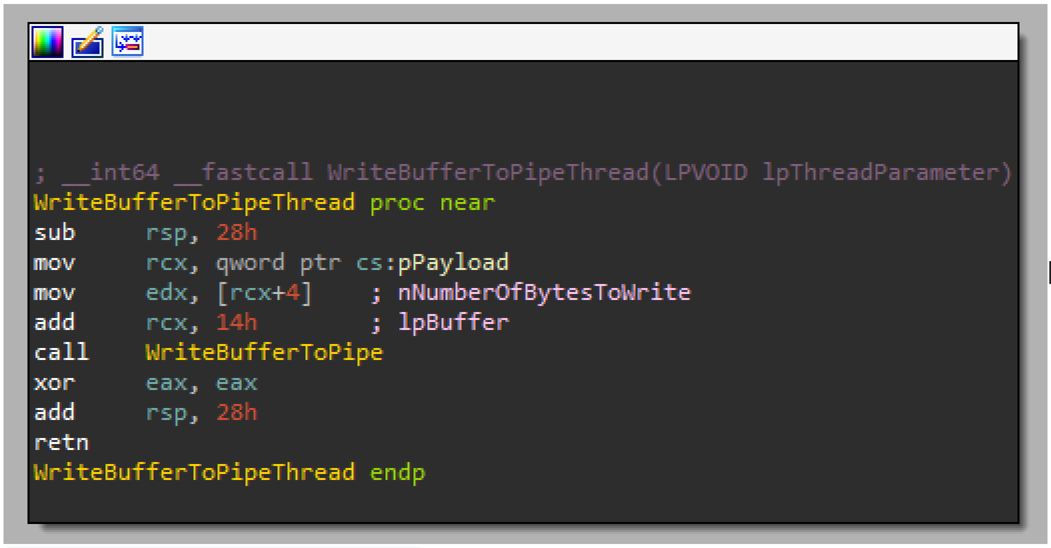


Execution flow from `DllMain` until `DecryptBufferAndExec` .
As we can see, two threads are now going to run concurrently. Let's focus ourselves on the one writing into the pipe (`WriteBufferToPipeThread`) followed by its reading counterpart (`PipeDecryptExec`) afterwards.

The `WriteBufferToPipe` Thread

The thread writing into the generated pipe is launched from `DecryptBufferAndExec` without any additional arguments. By entering into the `WriteBufferToPipeThread` function, we can observe it is a simple wrapper to `WriteBufferToPipe` except it furthermore passes the following arguments recovered from a global `Payload` variable (pointed to by the `pPayload` pointer):

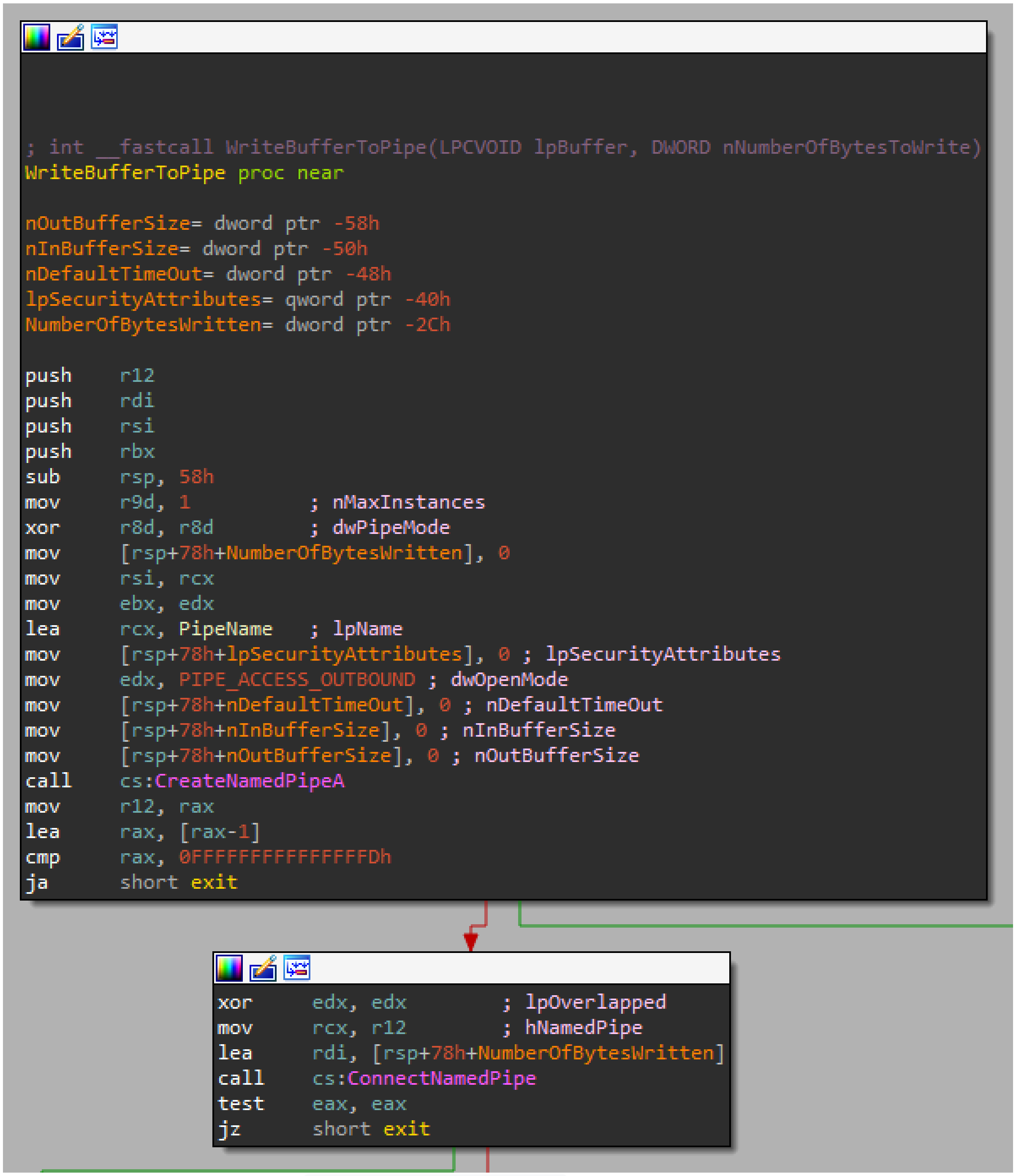
1. The size of the shellcode, stored at offset `0x4` .
2. A pointer to a buffer containing the encrypted shellcode, stored at offset `0x14` .



```
; __int64 __fastcall WriteBufferToPipeThread(LPVOID lpThreadParameter)
WriteBufferToPipeThread proc near
sub     rsp, 28h
mov     rcx, qword ptr cs:pPayload
mov     edx, [rcx+4]     ; nNumberOfBytesToWrite
add     rcx, 14h         ; lpBuffer
call    WriteBufferToPipe
xor     eax, eax
add     rsp, 28h
retn
WriteBufferToPipeThread endp
```

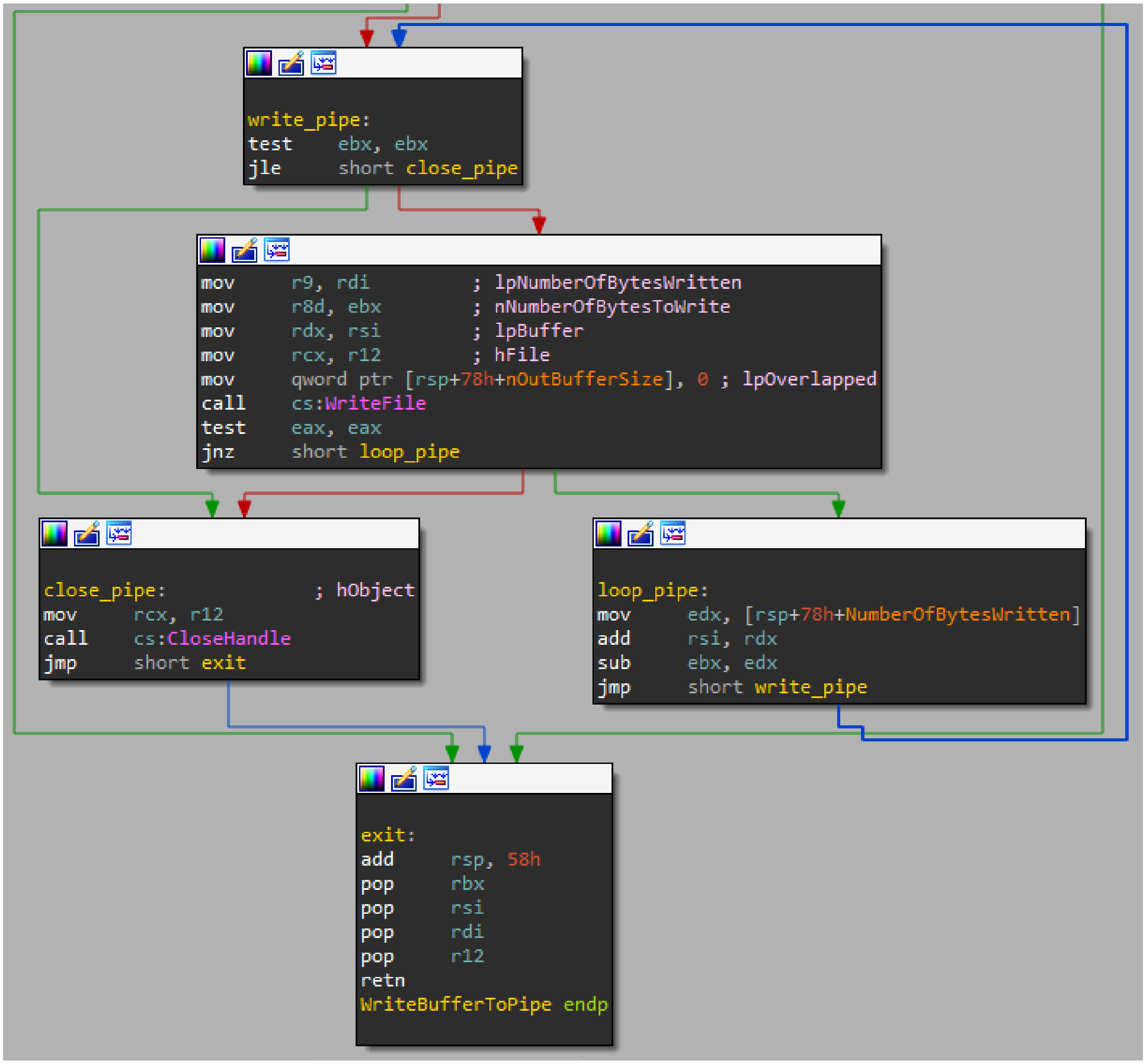
Disassembly of `WriteBufferToPipeThread`.

Within the `WriteBufferToPipe` function we can notice the code starts by creating a new pipe. The pipe's name is recovered from the `PipeName` global variable which, if you remember, was previously populated by the `sprintf` function. The code creates a single instance, outbound pipe (`PIPE_ACCESS_OUTBOUND`) by calling `CreateNamedPipeA` and then connects to it using the `ConnectNamedPipe` call.



Graphed disassembly of `WriteBufferToPipe` ‘s named pipe creation.

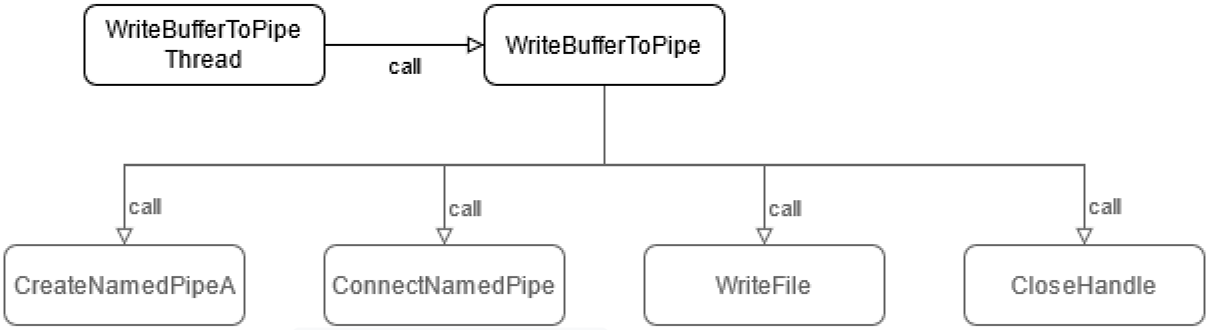
If the connection was successful, the `WriteBufferToPipe` function proceeds to loop the `WriteFile` call as long as there are bytes of the shellcode to be written into the pipe.



Graphed disassembly of `WriteBufferToPipe` writing to the pipe.

One important detail worth noting is that once the shellcode is written into the pipe, the previously opened handle to the pipe is closed through `CloseHandle`. This indicates that the pipe's sole purpose was to transfer the encrypted shellcode.

Once the `WriteBufferToPipe` function is completed, the thread terminates. Overall the execution flow was quite simple and can be graphed as follows:



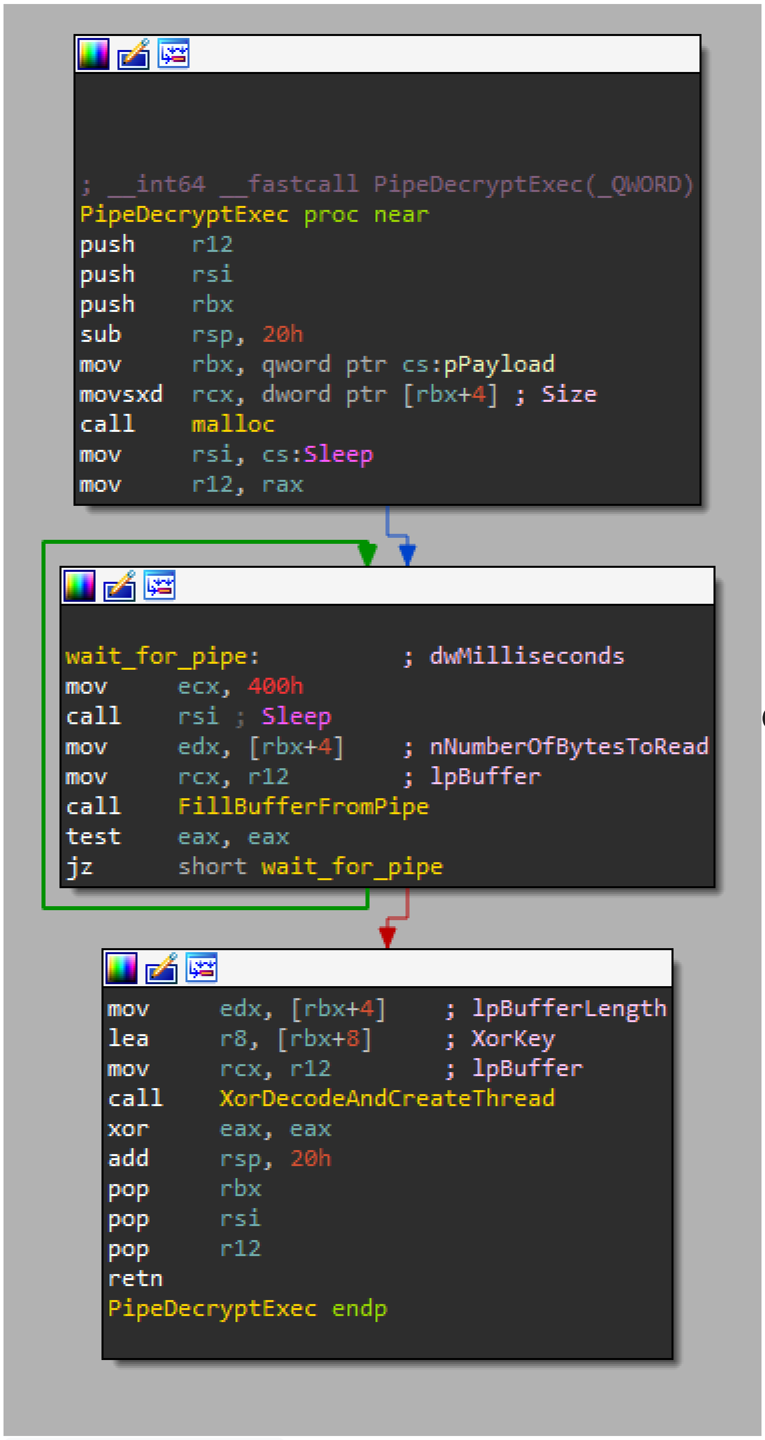
Execution flow from `WriteBufferToPipe`.

The `PipeDecryptExec` Flow

As a quick refresher, the `PipeDecryptExec` flow was executed immediately after the creation of the `WriteBufferToPipe` thread. The first task performed by `PipeDecryptExec` is to allocate a memory region to receive shellcode to be transmitted through the named pipe. To do so, a call to `malloc` is performed with as argument the shellcode size stored at offset `0x4` of the global `Payload` variable.

Once the buffer allocation is completed, the code sleeps for 1024 milliseconds (`0x400`) and calls `FillBufferFromPipe` with both buffer location and buffer size as argument. Should the `FillBufferFromPipe` call fail by returning `FALSE` (`0`), the code loops again to the `Sleep` call and attempts the operation again until it succeeds. These `Sleep` calls and loops are required as the multi-threaded sample has to wait for the shellcode being written into the pipe.

Once the shellcode is written to the allocated buffer, `PipeDecryptExec` will finally launch the decryption and execution through `XorDecodeAndCreateThread`.



Graphed disassembly of `PipeDecryptExec`.

To transfer the encrypted shellcode from the pipe into the allocated buffer, `FillBufferFromPipe` opens the pipe in read-only mode (`GENERIC_READ`) using `CreateFileA`. As was done for the pipe's creation, the name is retrieved from the global `PipeName` variable. If accessing the pipe fails, the function proceeds to return `FALSE` (`0`), resulting in the above described `Sleep` and retry loop.

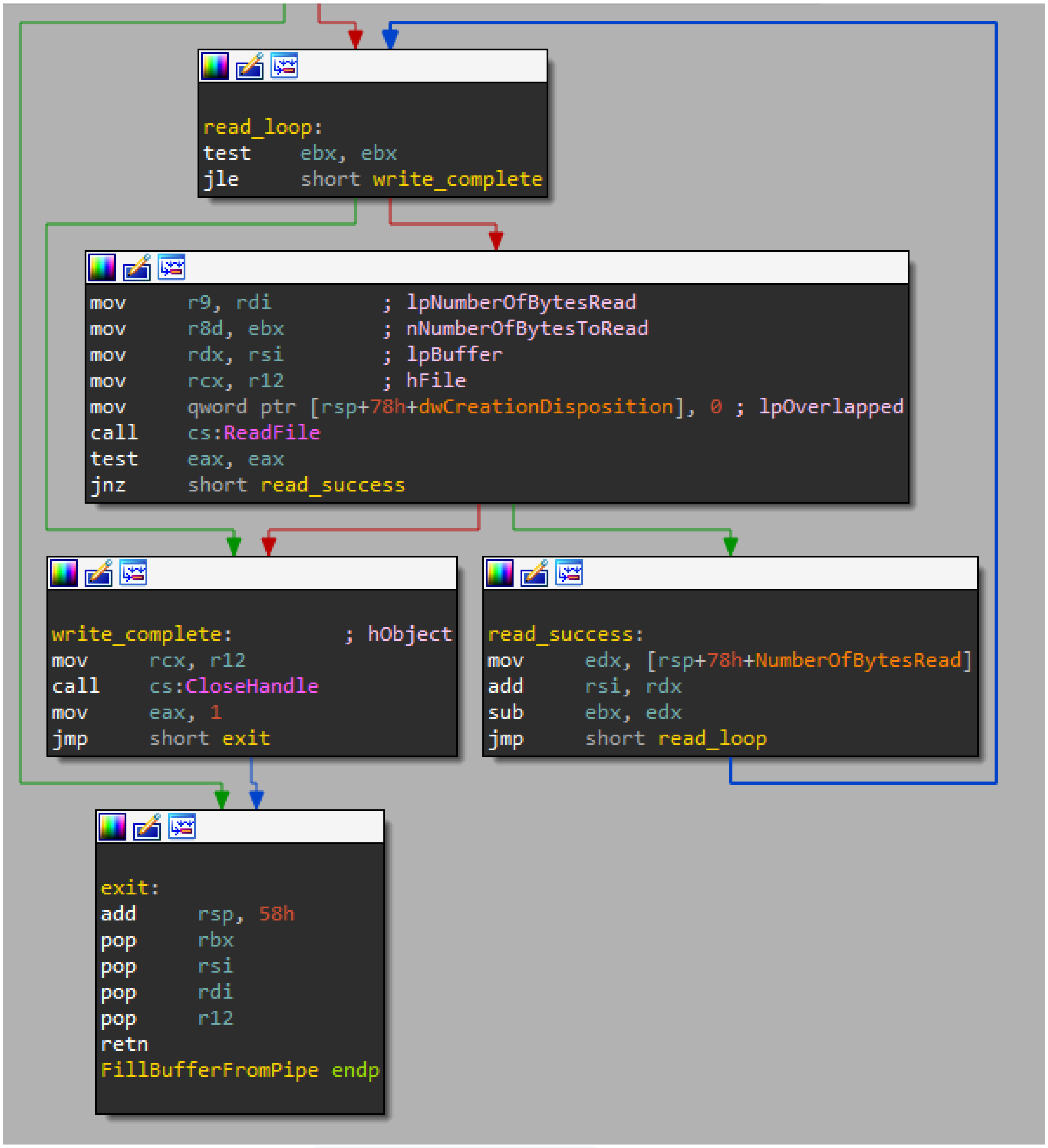
```
; __int64 __fastcall FillBufferFromPipe(LPVOID lpBuffer, DWORD nNumberOfBytesToRead)
FillBufferFromPipe proc near

dwCreationDisposition= dword ptr -58h
dwFlagsAndAttributes= dword ptr -50h
hTemplateFile= qword ptr -48h
NumberOfBytesRead= dword ptr -2Ch

push    r12
push    rdi
push    rsi
push    rbx
sub     rsp, 58h
xor     r9d, r9d        ; lpSecurityAttributes
mov     r8d, 3          ; dwShareMode
mov     [rsp+78h+NumberOfBytesRead], 0
mov     rsi, rcx
mov     ebx, edx
lea     rcx, PipeName   ; lpFileName
mov     [rsp+78h+hTemplateFile], 0 ; hTemplateFile
mov     edx, GENERIC_READ ; dwDesiredAccess
lea     rdi, [rsp+78h+NumberOfBytesRead]
mov     [rsp+78h+dwFlagsAndAttributes], FILE_ATTRIBUTE_NORMAL ; dwFlagsAndAttributes
mov     [rsp+78h+dwCreationDisposition], 3 ; dwCreationDisposition
call    cs:CreateFileA
mov     r12, rax
xor     eax, eax
cmp     r12, 0FFFFFFFFFFFFFFFFh
jz      short exit
```

Disassembly of `FillBufferFromPipe` ‘s pipe access.

Once the pipe opened in read-only mode, the `FillBufferFromPipe` function proceeds to copy over the shellcode until the allocated buffer is filled using [`ReadFile`](). Once the buffer filled, the handle to the named pipe is closed through `CloseHandle` and `FillBufferFromPipe` returns `TRUE` (`1`).



Graphed disassembly of `FillBufferFromPipe` copying data.

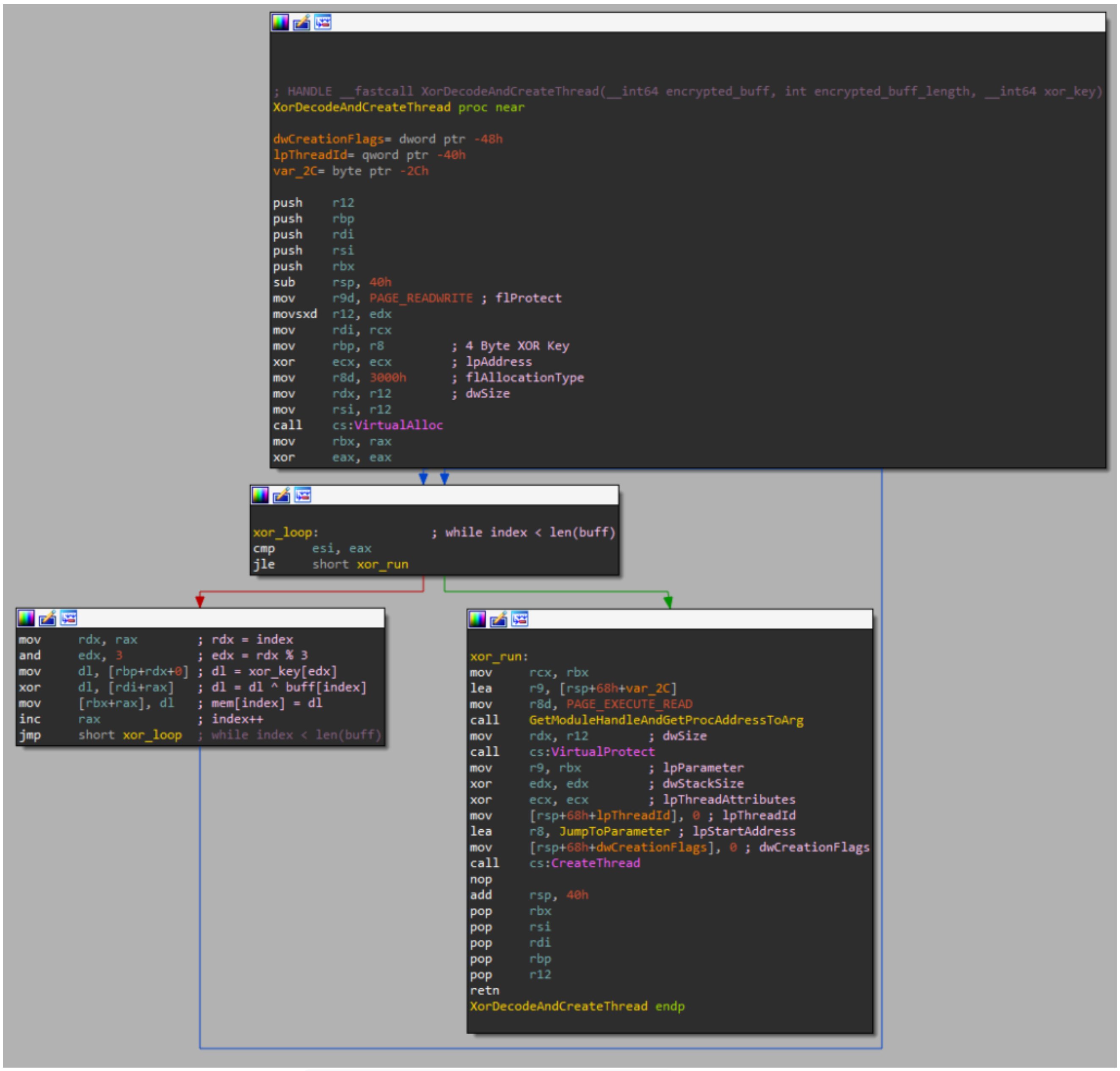
Once `FillBufferFromPipe` has successfully completed, the named pipe has completed its task and the encrypted shellcode has been moved from one memory region to another.

Back in the caller `PipeDecryptExec` function, once the `FillBufferFromPipe` call returns `TRUE` the `XorDecodeAndCreateThread` function gets called with the following parameters:

1. The buffer containing the copied shellcode.
2. The length of the shellcode, stored at the global `Payload` variable’s offset `0x4`.
3. The symmetric XOR decryption key, stored at the global `Payload` variable’s offset `0x8`.

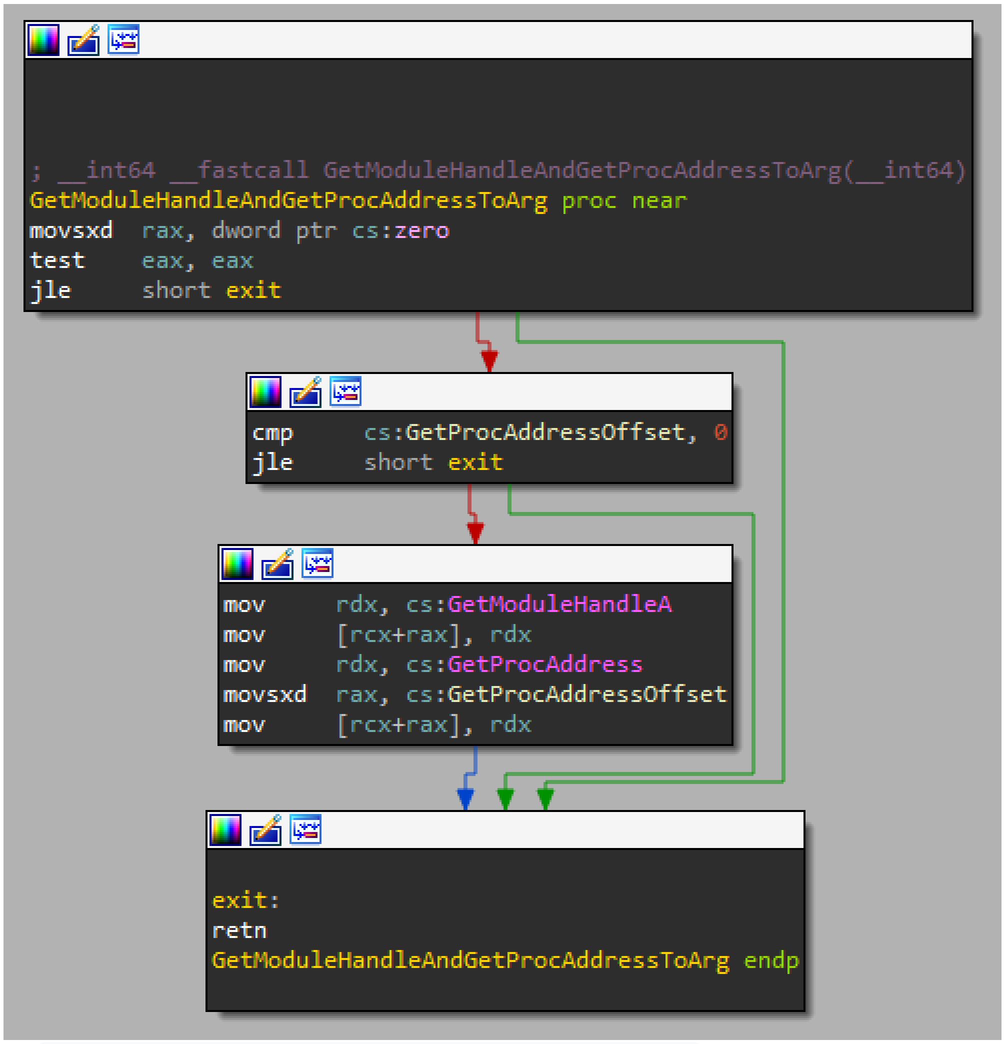
Once invoked, the `XorDecodeAndCreateThread` function starts by allocating yet another memory region using `VirtualAlloc`. The allocated region has read/write permissions (`PAGE_READWRITE`) but is not executable. By not making a region writable and executable at the same time, the sample possibly attempts to evade security solutions which only look for `PAGE_EXECUTE_READWRITE` regions.

Once the region is allocated, the function loops over the shellcode buffer and decrypts each byte using a simple `xor` operation into the newly allocated region.



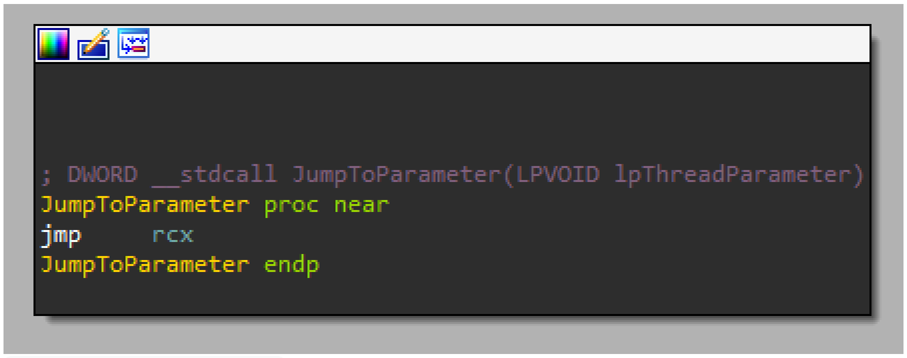
Graphed disassembly of `XorDecodeAndCreateThread`.

When the decryption is complete, the `GetModuleHandleAndGetProcAddressToArg` function is called. Its role is to place pointers to two valuable functions into memory: `GetModuleHandleA` and `GetProcAddress`. These functions should enable the shellcode to further resolve additional procedures without relying on them being imported. Before storing these pointers, the `GetModuleHandleAndGetProcAddressToArg` function first ensures a specific value is not `FALSE` (`0`). Surprisingly enough, this value stored in a global variable (here called `zero`) is always `FALSE` , resulting in the pointers never being stored.



Graphed disassembly of `GetModuleHandleAndGetProcAddressToArg` .

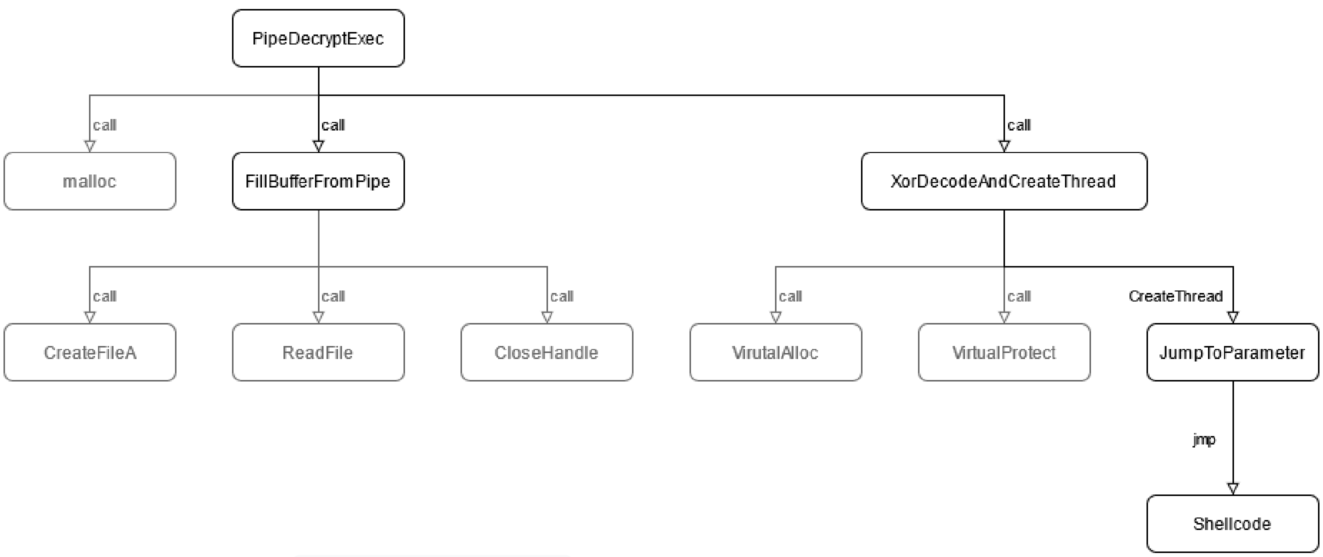
Back in the caller function, `XorDecodeAndCreateThread` changes the shellcode's memory region to be executable (`PAGE_EXECUTE_READ`) using `VirtualProtect` and finally creates a new thread. This final thread starts at the `JumpToParameter` function which acts as a simple wrapper to the shellcode, provided as argument.



Disassembly of `JumpToParameter`.

From here, the previously encrypted Cobalt Strike shellcode stager executes to resolve WinINet procedures, download the final beacon and execute it. We will not cover the shellcode's analysis in this post as it would deserve a post of its own.

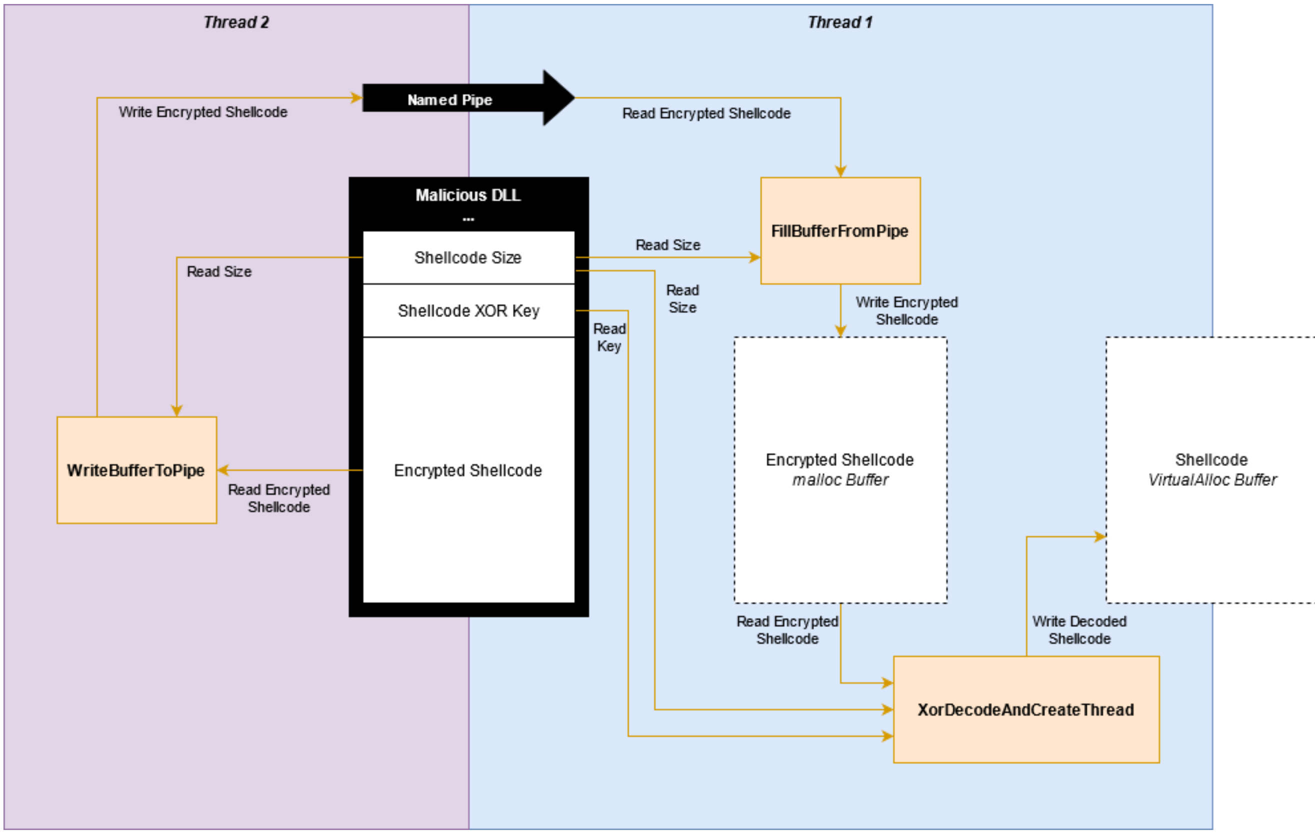
While this last flow contained more branches and logic, the overall graph remains quite simple:



Execution flow from `PipeDecryptExec` until the shellcode.

Memory Flow Analysis

What was the most surprising throughout the above analysis was the presence of a well-known named pipe. Pipes can be used as a defense evasion mechanism by decrypting the shellcode at pipe exit or for inter-process communications; but in our case it merely acted as a `memcpy` to move encrypted shellcode from the DLL into another buffer.

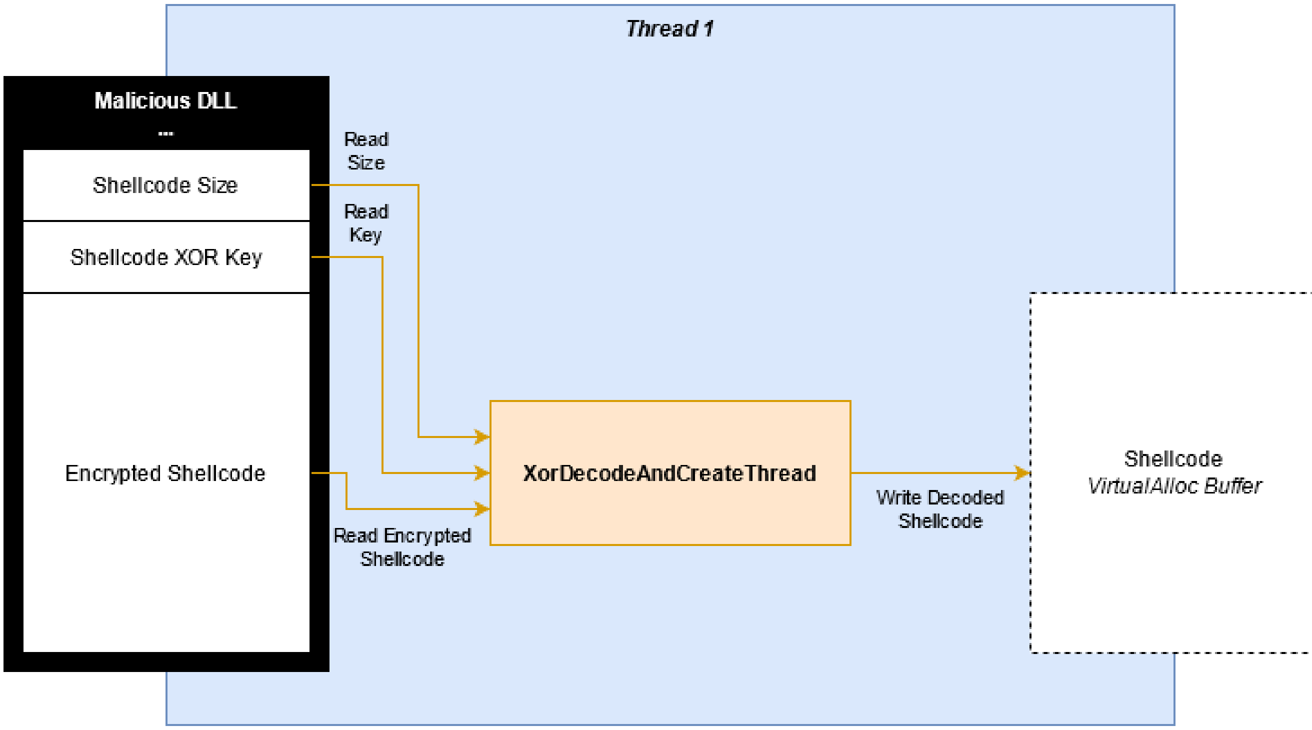


Memory flow from encrypted shellcode until decryption.

So why would this overhead be implemented? As pointed out by another colleague, the answer lays in the Artifact Kit, a Cobalt Strike dependency:

> Cobalt Strike uses the Artifact Kit to generate its executables and DLLs. The Artifact Kit is a source code framework to build executables and DLLs that evade some anti-virus products. […] One of the techniques [see: `src-common/bypass-pipe.c` in the Artifact Kit] generates executables and DLLs that serve shellcode to themselves over a named pipe. If an anti-virus sandbox does not emulate named pipes, it will not find the known bad shellcode.
>
> *cobaltstrike.com/help-artifact-kit*

As we can see in the above diagram, the staging of the encrypted shellcode in the `malloc` buffer generates a lot of overhead supposedly for evasion. These operations could be avoided should `XorDecodeAndCreateThread` instead directly read from the initial encrypted shellcode as outlined in the next diagram. Avoiding the usage of named pipes will furthermore remove the need for looped `Sleep` calls as the data would be readily available.

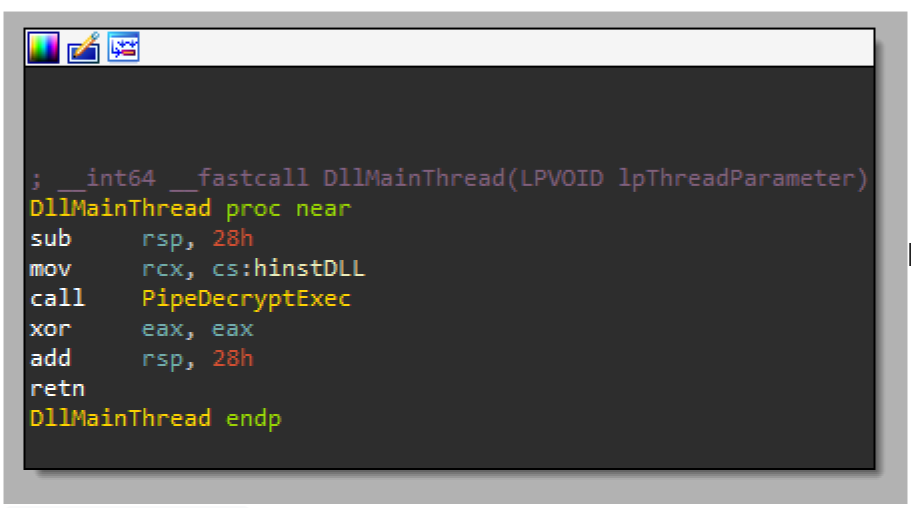


Improved memory flow from encrypted shellcode until decryption.

It seems we found a way to reduce the time-to-shellcode; but do popular anti-virus solutions actually get tricked by the named pipe?

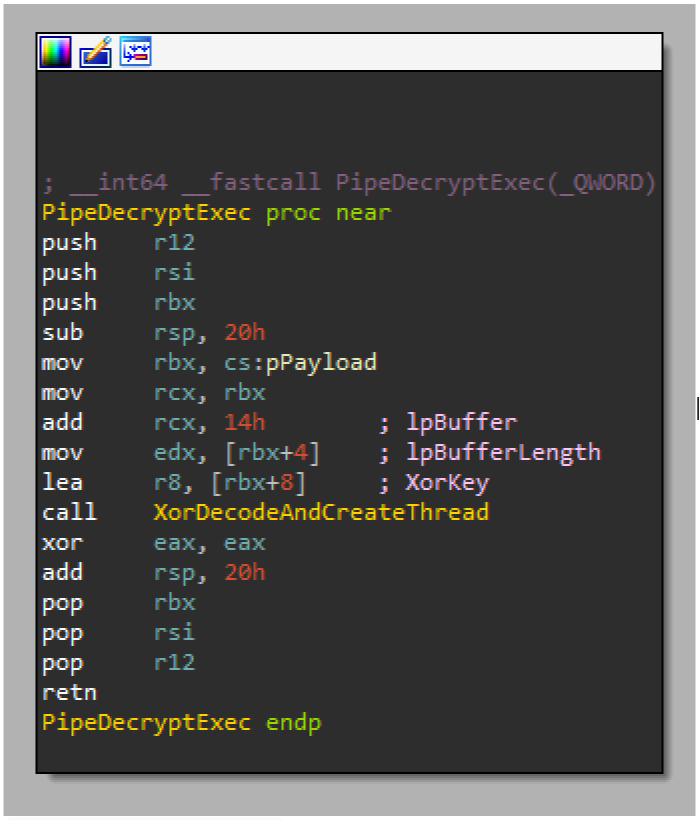
Patching the Execution Flow

To test that theory, let's improve the malicious execution flow. For starters we could skip the useless pipe-related calls and have the `DllMainThread` function call `PipeDecryptExec` directly, bypassing pipe creation and writing. How the assembly-level patching is performed is beyond this blog post's scope as we are just interested in the flow's abstraction.



Disassembly of the patched `DllMainThread`.

The `PipeDecryptExec` function will also require patching to skip `malloc` allocation, pipe reading and ensure it provides `XorDecodeAndCreateThread` with the DLL's encrypted shellcode instead of the now-nonexistent duplicated region.

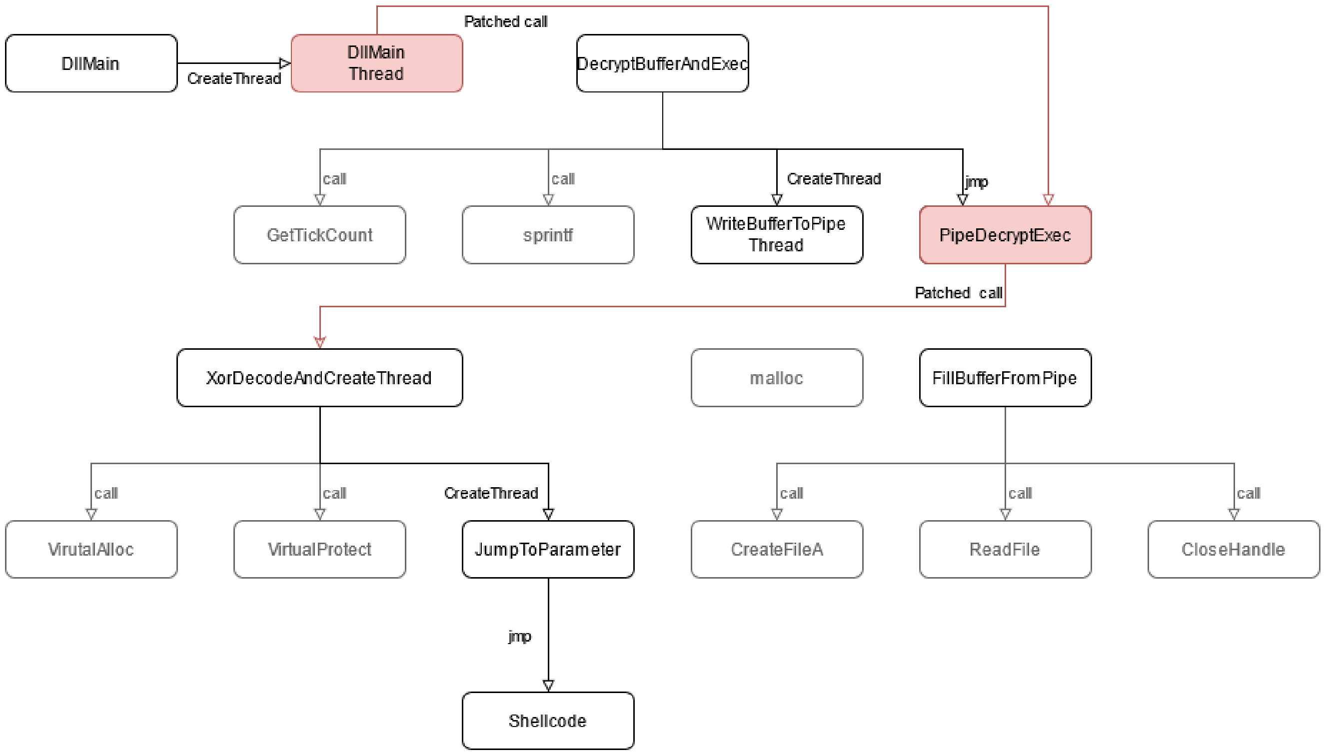


```
; __int64 __fastcall PipeDecryptExec(_QWORD)
PipeDecryptExec proc near
push    r12
push    rsi
push    rbx
sub     rsp, 20h
mov     rbx, cs:pPayload
mov     rcx, rbx
add     rcx, 14h        ; lpBuffer
mov     edx, [rbx+4]    ; lpBufferLength
lea     r8, [rbx+8]     ; XorKey
call    XorDecodeAndCreateThread
xor     eax, eax
add     rsp, 20h
pop     rbx
pop     rsi
pop     r12
retn
PipeDecryptExec endp
```

Disassembly of the patched `PipeDecryptExec`.

With our execution flow patched, we can furthermore zero-out any unused instructions should these be used by security solutions as a detection base.

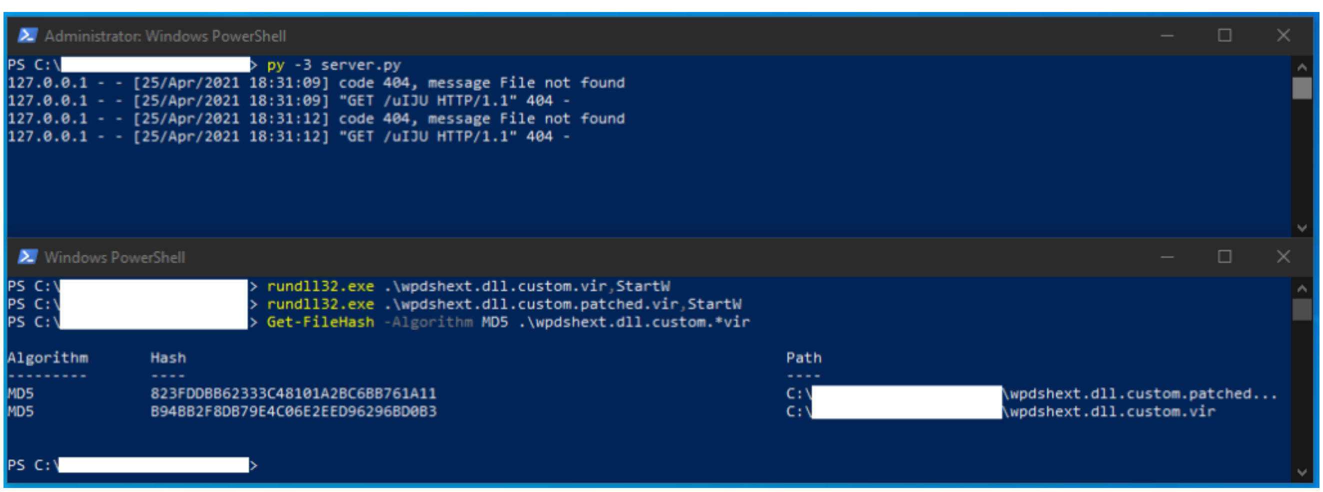
When the patches are applied, we end up with a linear and shorter path until shellcode execution. The following graph focuses on this patched path and does not include the leaves beneath `WriteBufferToPipeThread.`



Outline of the patched (red) execution flow and functions.

As we also figured out how the shellcode is encrypted (we have the `xor` key), we modified both samples to redact the actual C2 as it can be used to identify our targeted customer.

To ensure the shellcode did not rely on any bypassed calls, we spun up a quick Python HTTPS server and made sure the redacted domain resolved to `127.0.0.1`. We then can invoke both the original and patched DLL through `rundll32.exe` and observe how the shellcode still attempts to retrieve the Cobalt Strike beacon, proving our patches did not affect the shellcode. The exported `StartW` function we invoke is a simple wrapper around the `Sleep` call.



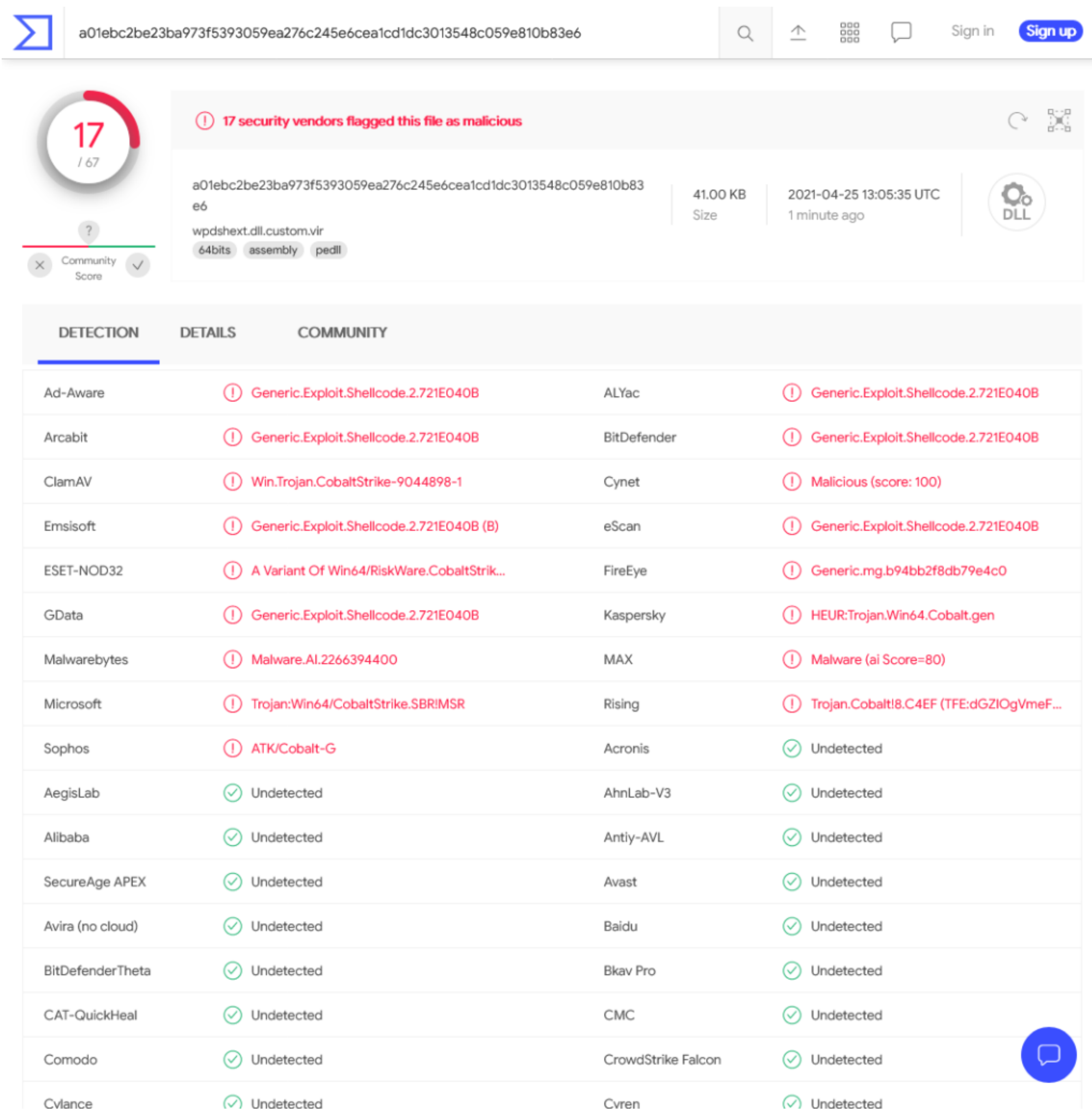
Capture of both the original and patched DLL attempting to fetch the Cobalt Strike beacon.

Anti-Virus Review

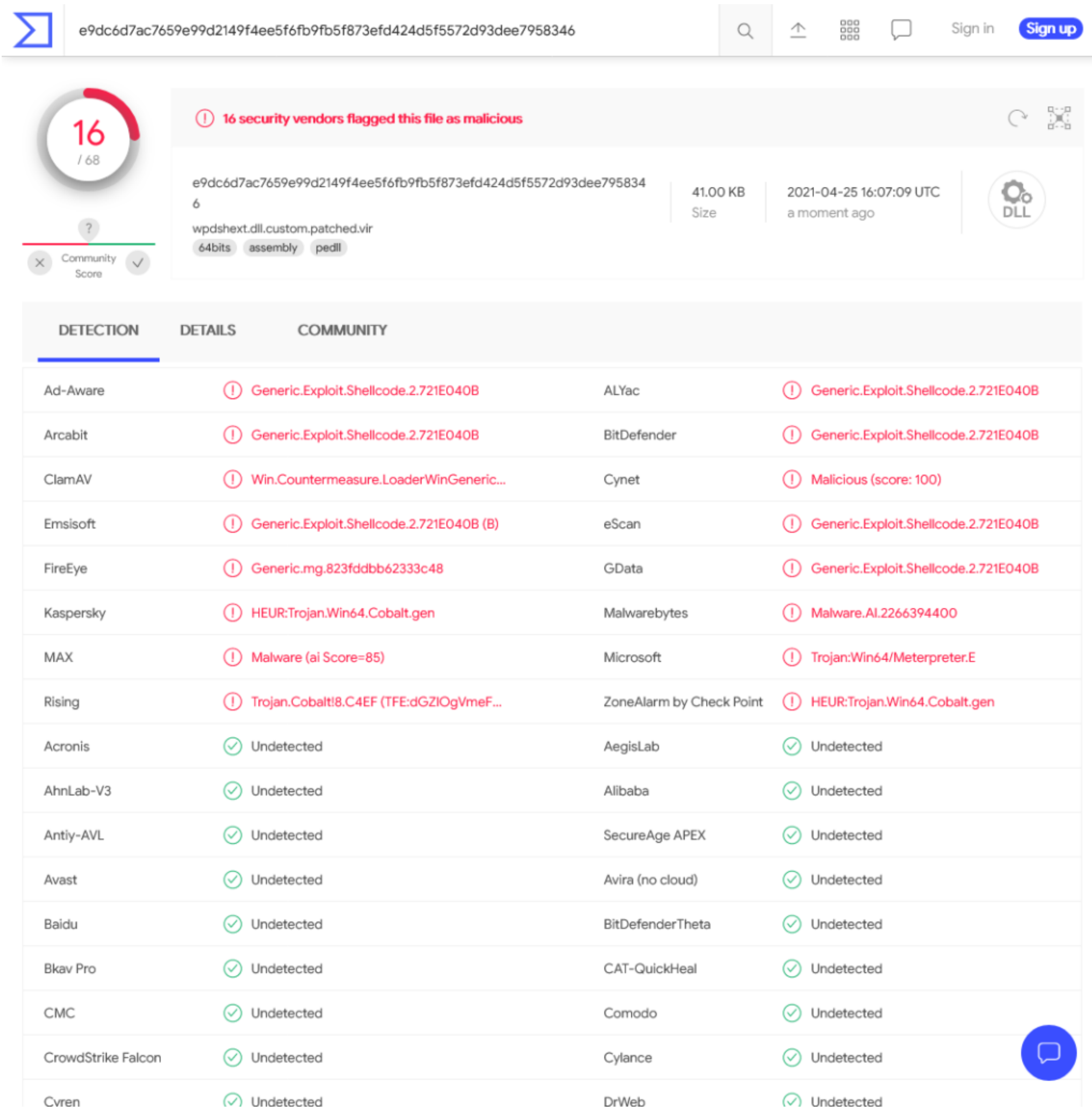
So do named pipes actually work as a defense evasion mechanism? While there are efficient ways to measure our patches' impact (e.g.: comparing across multiple sandbox solutions), VirusTotal does offer a quick primary assessment. As such, we submitted the following versions with redacted C2 to VirusTotal:

- `wpdshext.dll.custom.vir` which is the redacted Cobalt Strike DLL.
- `wpdshext.dll.custom.patched.vir` which is our patched and redacted Cobalt Strike DLL without named pipes.

As the original Cobalt Strike contains identifiable patterns (the named pipe), we would expect the patched version to have a lower detection ratio, although the Artifact Kit would disagree.



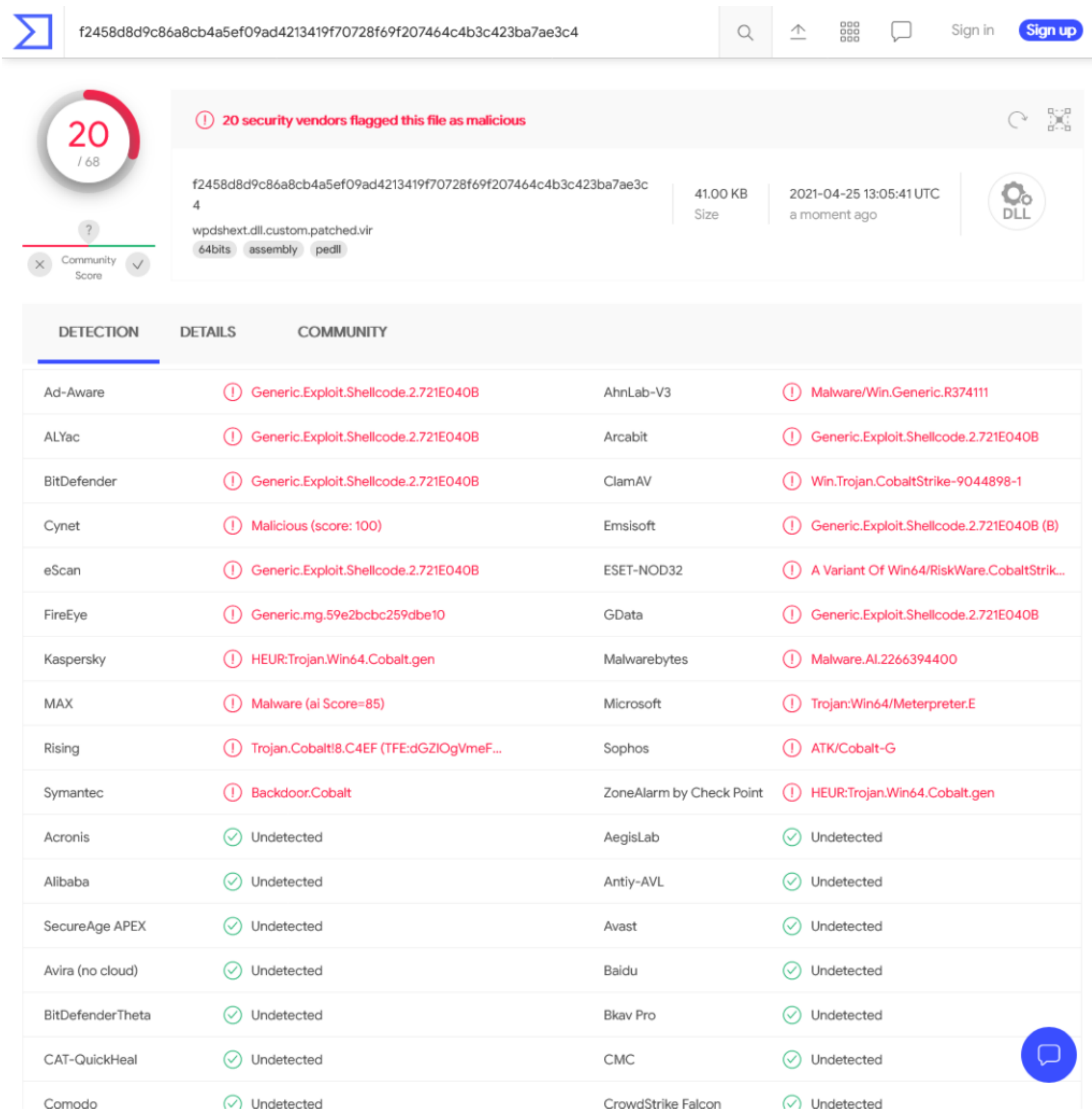
Capture of the original Cobalt Strike's detection ratio on VirusTotal.



Capture of the patched Cobalt Strike's detection ratio on VirusTotal.

As we expected, the named-pipe overhead leveraged by Cobalt Strike actually turned out to act as a detection base. As can be seen in the above captures, while the original version (left) obtained only 17 detections, the patched version (right) obtained one less for a total of 16 detections. Among the thrown-off solutions we noticed ESET and Sophos did not manage to detect the pipe-less version, whereas ZoneAlarm couldn't identify the original version.

One notable observation is that an intermediary patch where the flow is adapted but unused code is not zeroed-out turned out to be the most detected version with a total of 20 hits. This higher detection rate occurs as this patch allows pipe-unaware anti-virus vendors to also locate the shellcode while pipe-related operation signatures are still applicable.



f2458d8d9c86a8cb4a5ef09ad4213419f70728f69f207464c4b3c423ba7ae3c4

Sign in Sign up

20 / 68

20 security vendors flagged this file as malicious

f2458d8d9c86a8cb4a5ef09ad4213419f70728f69f207464c4b3c423ba7ae3c4

wpdshext.dll.custom.patched.vir

64bits assembly pedll

41.00 KB Size

2021-04-25 13:05:41 UTC a moment ago

DLL

Community Score

DETECTION DETAILS COMMUNITY

Ad-Aware	Generic.Exploit.Shellcode.2.721E040B	AhnLab-V3	Malware/Win.Generic.R374111
ALYac	Generic.Exploit.Shellcode.2.721E040B	Arcabit	Generic.Exploit.Shellcode.2.721E040B
BitDefender	Generic.Exploit.Shellcode.2.721E040B	ClamAV	Win.Trojan.CobaltStrike-9044898-1
Cynet	Malicious (score: 100)	Emsisoft	Generic.Exploit.Shellcode.2.721E040B (B)
eScan	Generic.Exploit.Shellcode.2.721E040B	ESET-NOD32	A Variant Of Win64/RiskWare.CobaltStrik...
FireEye	Generic.mg.59e2bcbc259dbe10	GData	Generic.Exploit.Shellcode.2.721E040B
Kaspersky	HEUR:Trojan.Win64.Cobalt.gen	Malwarebytes	Malware.AI.2266394400
MAX	Malware (ai Score=85)	Microsoft	Trojan:Win64/Meterpreter.E
Rising	Trojan.Cobalt!8.C4EF (TFE:dGZlOgVmeF...	Sophos	ATK/Cobalt-G
Symantec	Backdoor.Cobalt	ZoneAlarm by Check Point	HEUR:Trojan.Win64.Cobalt.gen
Acronis	Undetected	AegisLab	Undetected
Alibaba	Undetected	Antiy-AVL	Undetected
SecureAge APEX	Undetected	Avast	Undetected
Avira (no cloud)	Undetected	Baidu	Undetected
BitDefenderTheta	Undetected	Bkav Pro	Undetected
CAT-QuickHeal	Undetected	CMC	Undetected
Comodo	Undetected	CrowdStrike Falcon	Undetected

Capture of the intermediary patched Cobalt Strike's detection ratio on VirusTotal.

While these tests focused on the default Cobalt Strike behavior against the absence of named pipes, one might argue that a customized named pipe pattern would have had the best results. Although we did not think of this variant during the initial tests, we submitted a version with altered pipe names (`NVISO-RULES-%d` instead of `MSSE-%d-server`) the day after and obtained 18 detections. As a comparison, our two other samples had their detection rate increase to 30+ over night. We however have to consider the possibility that these 18 detections are influenced by the initial shellcode being burned.

Conclusion

Reversing the malicious Cobalt Strike DLL turned out to be more interesting than expected. Overall, we noticed the presence of noisy operations whose usage weren't a functional requirement and even turn out to act as a detection base. To confirm our hypothesis, we patched the execution flow and observed how our simplified version still reaches out to the C2 server with a lowered (almost unaltered) detection rate.

So why does it matter?

The Blue

First and foremost, this payload analysis highlights a common Cobalt Strike DLL pattern allowing us to further fine-tune detection rules. While this stager was the first DLL analyzed, we did take a look at other Cobalt Strike formats such as default beacons and those leveraging a malleable C2, both as Dynamic Link Libraries and Portable Executables. Surprisingly enough, all formats shared this commonly documented `MSSE-%d-server` pipe name and a quick search for open-source detection rules showed how little it is being hunted for.

The Red

Besides being helpful for NVISO's defensive operations, this research further comforts our offensive team in their choice of leveraging custom-built delivery mechanisms; even more so following the design choices we documented. The usage of named pipes in operations targeting mature environments is more likely to raise red flags and so far does not seem to provide any evasive advantage without alteration in the generation pattern at least.

To the next actor targeting our customers: I am looking forward to modifying your samples and test the effectiveness of altered pipe names.

Maxime Thiebaut

Maxime Thiebaut is a GCFA-certified intrusion analyst in NVISO's Managed Detection & Response team. He spends most of his time investigating incidents and improving detection capabilities. Previously, Maxime worked on the SANS SEC699 course. Besides his coding capabilities, Maxime enjoys reverse engineering samples observed in the wild.

Twitter