

Bachelor's Thesis Computing Science

Antivirus and EDR bypasses for initial access

Jorn Heibrink
s1040183

January 17, 2023

Company supervisor:
Stefan Vlems

First supervisor/assessor:
Bart Mennink

Second assessor:
Ileana Buhan

Radboud University



Abstract

For Red Team operations to accurately test the security of their target, it is important to implement Antivirus and Endpoint Detection and Response (EDR) bypasses. Antivirus and EDR are getting better over time, and so should the bypassing methods. In this research we dive into these bypassing methods and implement them in Cobalt Strike, a popular threat emulation software.

We take a look at the features of Antivirus and EDR, and implemented bypassing methods focused on these features. The used bypassing methods can be categorized into four categories: Signature evasion, sandbox evasion, API hook evasion, and network detection evasion.

Our results show that all the implemented bypassing methods decrease the detection rate, but fully bypassing Antivirus and EDR requires effort and a good balance between methods.

Table of Contents

| | | |
|----------|-------------------------------------------|-----------|
| 1 | Introduction | 3 |
| 1.1 | The Toolset | 3 |
| 1.2 | Outcomes | 4 |
| 1.3 | Outline | 5 |
| 2 | Preliminaries | 6 |
| 2.1 | Antivirus | 6 |
| 2.2 | Endpoint Detection and Response | 7 |
| 2.2.1 | Functionality | 7 |
| 2.3 | Red Team | 8 |
| 2.3.1 | Vulnerability Scan & Assessment | 8 |
| 2.3.2 | Pentesting | 8 |
| 2.3.3 | Red Teaming | 8 |
| 3 | Research | 10 |
| 3.1 | Setting up Cobalt Strike | 10 |
| 3.2 | Testing Environment | 11 |
| 3.3 | Expanding the Artifact Kit | 12 |
| 3.3.1 | Signature evasion | 13 |
| 3.3.2 | Sandbox evasion | 15 |
| 3.3.3 | Named pipes | 15 |
| 3.3.4 | Sandbox detection | 18 |
| 3.4 | Implementing Direct Syscalls | 20 |
| 3.4.1 | Bypassing the hooks | 21 |
| 3.4.2 | Moving to SysWhispers | 23 |
| 3.4.3 | Implementing the functions | 26 |
| 3.5 | Malleable Profiles | 30 |
| 3.5.1 | Profile structure | 30 |
| 3.5.2 | Situational profile | 32 |
| 3.5.3 | Universal profile | 35 |
| 4 | Related Work | 39 |
| 5 | Conclusions | 41 |

| | | |
|----------|-----------------------------------------------|-----------|
| A | Request and Response pairs | 46 |
| A.1 | Request and response read.aspx page | 46 |
| A.1.1 | Request | 46 |
| A.1.2 | Response | 47 |

Chapter 1

Introduction

There are many methods to assess the security of a company, one of the methods Secura provides is Red Team operations [2.3]. With Red Team operations the goal is to emulate a real world attack as realistic as possible.

One key part of Red Team operations is to evade the defending Blue Team. For successful evasion it is important to bypass all the detection methods that are deployed on a system. Two of these methods are Antivirus (AV) and Endpoint Detection and Response (EDR). To really test the limits of the security, a Red Team has to try to bypass the AV and EDR.

AV and EDR software keep improving to increase the security. The Red Team also has to update bypassing methods to keep up with the increased security. The Red Teams at Secura also have to update the bypassing methods. The goal of this thesis is to research bypassing methods and implement these methods in the toolset Secura uses.

1.1 The Toolset

The main tool Secura uses is Cobalt Strike. Cobalt Strike is threat emulation software ideal for Red Team exercises. Cobalt Strike can replicate the tactics and techniques of a real life threat actor in a network. Cobalt Strike makes use of so called Beacons as post exploitation payload.

A Beacon lets the attacker interact with the system it is deployed on. Some examples of interactions are: Executing Powershell scripts, log keystrokes, take screenshots, and spawn other beacons. A bea-

con uses asynchronous communication over HTTP, HTTPs or DNS. The Beacon is developed to be as flexible as possible. Beacons can be modified specifically for certain targets or can be used to emulate real threat actors. Our goal was to research bypassing methods and implement these methods in the Beacons.

1.2 Outcomes

There are many methods AV and EDR deploy to detect malicious activity. For our research we focused on a subset of these detection methods. The methods can roughly be divided in four subsets: Signature, Sandbox, API hooks and Behavior. For each of these four parts we researched concepts for bypassing them. Afterwards we implemented these concepts in the Beacons to make sure that the concepts are actually viable.

For signature detection we made a list of concepts for bypassing. We kept track of this list while implementing the rest of our methods to make sure signature detection will not trigger.

For Sandbox detection we found two bypassing methods. The first method is to use functions that are not supported by the sandbox. The second method is to detect if the code is run in a sandbox. When a sandbox is detected the Beacon will not execute the malicious sections of the code.

For API hooks we implemented methods for direct system calls, which results in not having to use the hooked APIs at all. We can not trigger hooks on an API when we do not use the API. The functions that would normally be called through the API were implemented directly in the code.

The final part focuses on the behavior of the Beacons. We researched how we could change the behavior of our Beacons using the Malleable Profiles of Cobalt Strike. For our implementation we focused specifically on the network traffic section of the Malleable Profiles. We implemented methods to hide Beacon traffic in normal traffic that would be present in a target environment.

All the implemented methods showed improvement in detection rate.

1.3 Outline

In the next chapters we will first create a better understanding of the concepts of AV in Section 2.1, EDR in Section 2.2, and Red Teaming in Section 2.3. After the preliminaries we will start the research by setting up Cobalt Strike in Section 3.1 and the testing environment in Section 3.2. When the environment is set up correctly we will visit each of the four detection methods. First we take a look at the Cobalt Strike Artifact Kit in Section 3.3, and combine this with our signature and sandbox bypassing methods in Section 3.3.1 and Section 3.3.2. Next we focus on bypassing the API hooks in Section 3.4, where we will implement direct syscalls. The last part of the research will be about the behavior for which we used the Malleable Profiles in Section 3.5. After the research we compare our findings with findings of other research in Chapter 4. Finally we will talk about the conclusions of our research in Chapter 5.

Chapter 2

Preliminaries

In this chapter we will explain how AV (Section 2.1) and EDR (Section 2.2) software works. We will also take a look at what Red Teaming is, and how Red Teaming differs from other methods to test security in Section 2.3.

2.1 Antivirus

AV is host-based security solution software with the goal to detect and prevent malware attacks on the host machine [26]. The AV runs in the background of a system to monitor activity. AV software can have different functionalities, with different detection techniques. In general we can combine these techniques in two categories: signature detection and behavior detection [6].

Signature detection scans the system for signatures of programs that indicate malicious software. AV software compares the signatures of software on the system with a dictionary of known indicators of malicious software, also called indicators of compromise (IOC). The dictionary is populated by analysing previously detected malicious software. An IOC can be different parts of a program: file hashes, byte sequences, malicious domains, and many more [7].

Behavior detection monitors the behavior of installed software. When software shows suspicious activity, the behavior detection spots this activity and alerts the user. Suspicious activity can be: modifying protected files, installation of certificates, process injection, and many more [12]. Behavior detection adds to signature detection by being able to detect unknown malware.

For this thesis we will focus on the two general techniques, as these

techniques are present in all AV products. For more vendor specific information on advanced features, the website of the vendor can be referenced. For an overview of AV products, comparison websites like [AV-TEST](#) can be used.

2.2 Endpoint Detection and Response

Endpoint Detection and Response is an endpoint security solution that continuously monitors end-user devices to detect and respond to cyber threats [4]. EDR records and stores behaviour on the endpoint system, uses data analytic techniques to detect suspicious system behaviour, provides contextual information, blocks malicious activity, and provides remediation suggestions to restore the affected systems.

2.2.1 Functionality

The functionality of EDR products can be divided in three sections: Endpoint behavioral sensors, cloud security analytics, and threat intelligence [13].

Endpoint behavioral sensors functionality is nearly the same as the behavior detection of AV products (Section 2.1). The only difference is that the EDR sends the behavioral data to a cloud instance for review.

The cloud security analytics processes the behavioral data using a collective database and machine learning. The behavioral signals are used for insights in activity, malware detection, and recommendations for actions that have to be taken.

All the systems using EDR, together with security experts, collectively create threat intelligence. The threat intelligence is used to identify techniques and procedures. The intelligence is also used for threat detection and recommendation for actions that have to be taken.

The EDR also records all relevant activity to catch incidents that evaded the automatic detection methods. The user can access everything that is recorded on their endpoint through the portal provided by the EDR vendor. The activity gives security teams everything they need for manual investigations.

For vendor specific features and an overview of all these vendors, comparison websites like [Gartner](#) can be used.

2.3 Red Team

There exist many ways to assess the security of a target. We will discuss three of these assessment methods that Secura and many other security companies provide: Vulnerability Scan & Assessment, Penetration Testing or Pentesting, and Red Teaming. For clarity we will explain the goal of each of the methods, because these three methods often get confused.

2.3.1 Vulnerability Scan & Assessment

A vulnerability scan & assessment gives a complete overview of the vulnerabilities of the target scope [25]. It uses automated scanning tools, extended with validation by the team that runs the scan. After the scan the team identifies the weak spots and gives an advice for improvement. The results will be summarized in a report which the client organization can use to reduce their security risks.

2.3.2 Pentesting

During a pentest the target scope will also be scanned, but a pentest extends on a vulnerability scan & assessment by actually trying to exploit the vulnerabilities [23]. By exploiting the vulnerabilities the consequences of an issue will become a lot clearer. A pentest will show the seriousness of issues, so the client will become aware of the potential dangers. The results will also be shown in a report, including a risk analysis and recommendations for the client.

The difference between a vulnerability assessment and a pentest is that the first will find more vulnerabilities, while the second will find less, but show more depth what the potential risk can be. Both methods can also be combined to create a complete assessment.

2.3.3 Red Teaming

Finally we have Red Teaming. The key difference for Red Teaming is simulation, to simulate a full-spectrum cyber-attack [24]. The Red Team will emulate attacks in such a way a real malicious actor would. Simulation gives a realistic insight in the security while also testing the capabilities of the defending team (the Blue Team).

The Red Team has to avoid detection by evading the Blue Team and their security measures.

The Red Team process can be split into three phases:

- **Phase 1: Planning and Preparation**
A dedicated project manager, the observers and the Red Team lead will create the schedule and the rules of engagement.
- **Phase 2: The Attack**
The Red Team will start their attack and try to access so-called “crown-jewels”. Trying to access can be done by any means necessary within the scope.
- **Phase 3: Clean Closure**
After the attack it is time to clean up all the digital remains and discuss the results with the Blue Team.

The attack phase can again be split up in four parts, resulting in a total overview called the “kill-chain” (Figure 2.1).

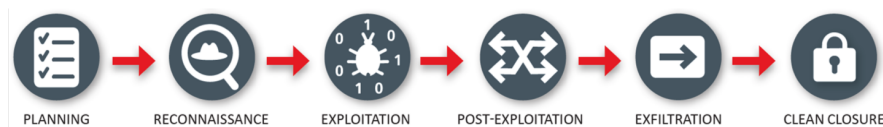


Figure 2.1: Red Team Kill-Chain

For this research the important part is the exploitation phase. During the exploitation phase the Red Team will try to break-in and get the initial access.

Initial access can be achieved by compromising a system, or exploiting target staff using social engineering (phishing, bypass physical control). The goal is to get a backdoor on a system of the target environment. To create a backdoor it is crucial for the payload to avoid detection.

Chapter 3

Research

In this chapter we will discuss the bypassing methods that we will implement, and test these methods using our testing environment. First we will set up Cobalt Strike in Section 3.1 and the test environment in Section 3.2. When the environment is set up correctly we will start with signature evasion in Section 3.3.1, followed by sandbox evasion in Section 3.3.2. We will extend on these evasion techniques by implementing direct system calls in Section 3.4. Finally we will change the network behavior by creating malleable profiles in Section 3.5.

3.1 Setting up Cobalt Strike

Cobalt Strike uses a client-server structure, so to interact with the Beacon clients we have to setup a Team Server. For our infrastructure we used two servers, one as back-end and one as relay. The back-end is the Team Server itself, the controller for the Beacon payload. The Team Server also stores the data that is collected by Cobalt Strike and manages all the logging. The relay serves as an extra layer between the Team Server and the target. In real world red team operations, we have to protect the Team Server, so the defending Blue Team can not trace back to the server. The relay makes sure the Team Server is not exposed directly.

The Infrastructure Deployment Figure 3.1 shows an overview of the infrastructure.

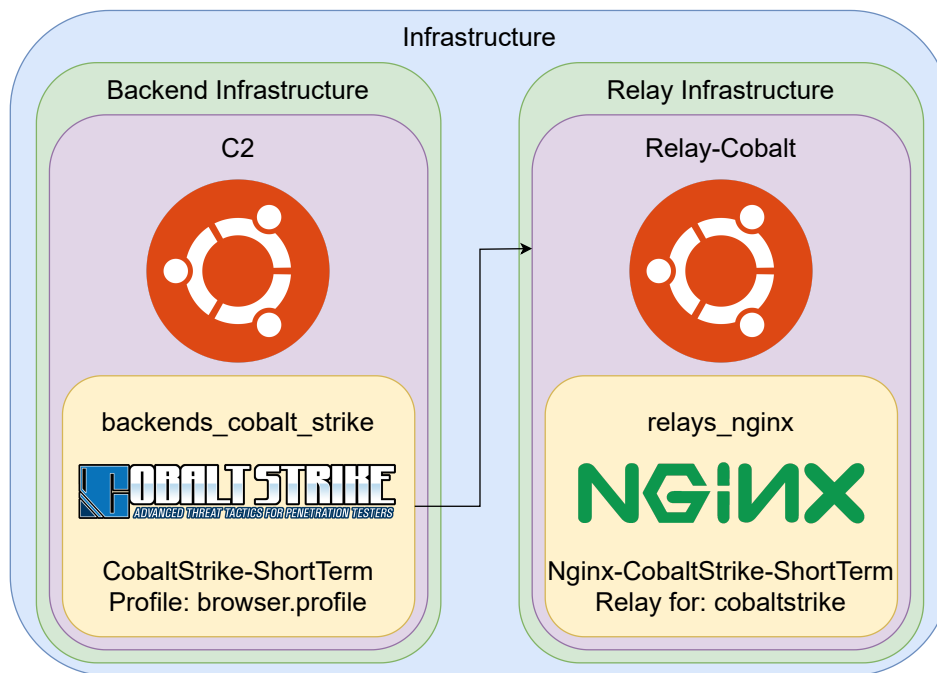


Figure 3.1: Infrastructure Deployment

3.2 Testing Environment

To test our Beacons, we need a testing environment that is as close as possible to what the Secura Red Teams will encounter in the real world. We used a virtual machine with the newest Windows version at the time of installation, this version was Windows 11 Enterprise version 10.0.22000 Build 22000. In addition we used the site [AntiScan.me](https://www.antiscan.me). AntiScan scans the supplied file with 26 of the most popular antivirus products. We used AntiScan instead of well known scanners like VirusTotal, because VirusTotal distributes the results, and AntiScan does not. Distributing our Beacons will make them unusable and exposes our Cobalt Strike environment.

As our EDR solution we chose Microsoft Defender for Endpoint, because Defender for Endpoint integrates best with our Windows based test environment. Defender for Endpoint is a part of the Microsoft Defender 365 enterprise defense suite, this suite can be used for free for 3 months by signing up with a company account. When we use Defender for Endpoint, we have to disable distribution of the results, otherwise we will run into the same problem as VirusTotal.

The distribution can be disabled in the [Advanced Features] section of the endpoint settings of the Defender 365 portal. Onboarding the test system can be done using the provided onboarding script on the Defender 365 portal. When the system is onboarded correctly, the system will show up in the [Endpoints] section.

The test system is now ready to be used for testing with our Cobalt Strike payloads.

3.3 Expanding the Artifact Kit

Cobalt Strike uses a so called Artifact Kit to generate its payloads, these payloads are also called artifacts. The artifacts can be an executable or a DLL. The Artifact Kit is a source code framework to build artifacts, which can evade signature checking and sandboxing. The Artifact kit is a part of the larger Arsenal Kit which can be downloaded from the Cobalt Strike [website](#) using a license key.

To use the Artifact Kit we first have to generate the configuration file by running the included `build_arsenal_kit.sh`. The script generates a configuration file with the supplied settings. Afterwards we can load the `arsenal_kit.cna` configuration file in the scripts view of Cobalt Strike:

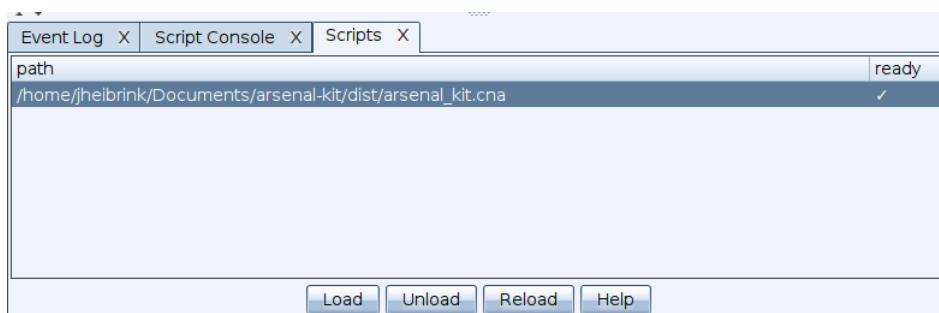


Figure 3.2: Loading scripts

Now we can generate our payloads by navigating to:

[Payload] -> [Windows Executable (Stageless)]

The menu presents options to select our listener and generate the payload.

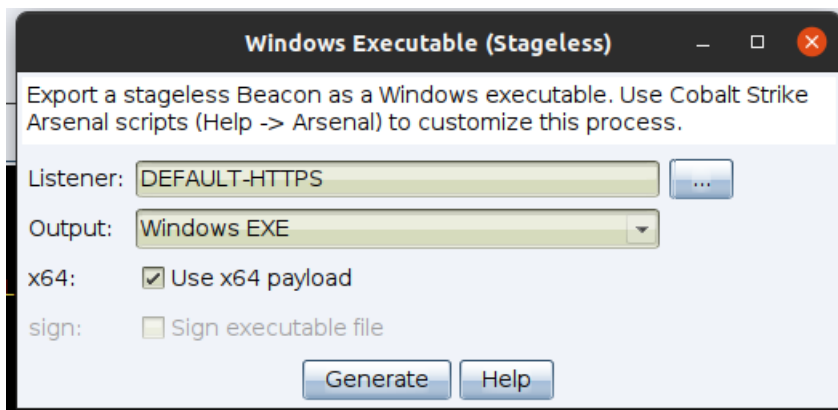


Figure 3.3: Generating a payload

After generating, we can check the script console to see that the Artifact Kit was loaded successfully:

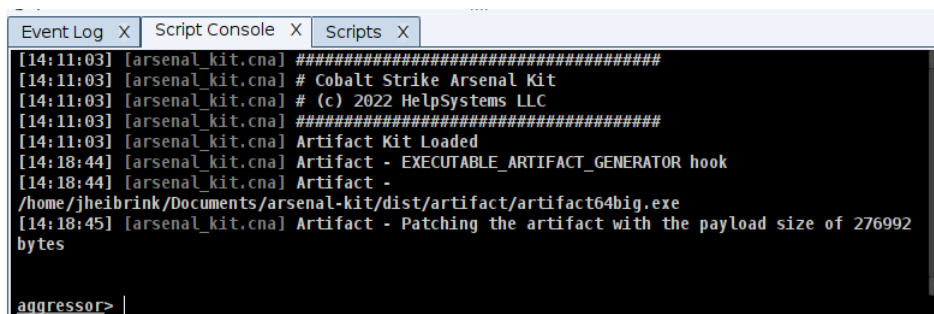


Figure 3.4: Script View

The payload should be ready and can be moved to the target system.

3.3.1 Signature evasion

Now we know how to generate payloads, we can start expanding on the Artifact Kit. As mentioned in the preliminaries the AV can recognize malicious software by using signatures. Signature detection can be bypassed using obfuscation methods. Obfuscation is the process of modifying the source code in such a way that it is more difficult to understand for the reader. The reader in our case is the AV software.

There are multiple ways to obfuscate our code. In the following list we will briefly explain some of these obfuscation methods:

- **Change the code order**
Change the execution order of the code without changing the behaviour of the program.
- **Inserting dead code**
Add dead code to the program, this will not change the behaviour of the program, but makes it a lot harder for static methods to distinguish the executed code from the dead code.
- **Code replacement**
Replace the original code with other instructions that result in the same behaviour.
- **Rename identifiers**
Rename the identifiers of variables, functions, and registers without changing the behaviour of the program.
- **Packing the code**
A packer compresses and encrypts the code into different forms while maintaining the same behaviour. The packer includes code into the program to unpack the compressed code. There are several ways to pack code, some popular methods of which are:
 - **Encryption**
Encrypt the code using an encryption key. Different versions of the same malware can be made using the same encryption method but different keys.
 - **Oligomorphic**
Using Oligomorphic encryption, the decryptor is built from different parts. The parts are randomly selected from a predefined set, which results in a different program for every decryption.
 - **Polymorphic**
Polymorphic uses the same principle as Oligomorphic, but now the packer can produce infinitely many different versions of the malware, instead of a limited set.
 - **Metamorphic**
A metamorphic packer does not need encryption. The code is changed dynamically by applying a series of transformations. The transformations result in malware being very hard to detect.

We will use some of these obfuscation methods together with other methods that we will discuss in the rest of the research.

3.3.2 Sandbox evasion

AV software not only uses signature detection, another detection method is behavior detection. Using a sandbox is part of the behavior detection. The AV software will create a virtual environment where the software can be safely executed. Using this sandbox the AV can determine if the software is malicious or not without exposure of the host system. For each step in the execution the AV checks the process space for malicious indicators. If indicators show up the executable gets flagged.

The limitation of emulation in AV software is that the engine can never support all existing operations. The AV will give up on the execution of the unsupported section of the program. The limitation enables bypassing the sandbox by “hiding” the malicious code in an operation or set of operations that is not supported by the AV.

Another limitation is that malware can detect when it runs in a sandbox. When a sandbox is detected, the malware can switch to normal behavior that would not be flagged as malicious.

3.3.3 Named pipes

One method we will use to bypass sandboxes is passing the payload through a named pipe before execution. A named pipe is a one-way or duplex pipe for communication between the pipe server and a pipe client. More information on named pipes can be found in the Win32 IPC documentation [15]. Named pipes is a technique that not all sandboxes are able to emulate correctly. Passing the payload through a named pipe will make the payload invisible to AV products that do not support named pipes.

Named pipes is a technique already present in the Cobalt Strike Artifact Kit. The code for the bypass can be found in the file `src-common/-bypass-pipe.c` of the Artifact Kit folder. Because we will be modifying the bypass, we create a new bypass file in the `src-common` folder called `bypass-secura.c`. As a starting point, the code from the default `bypass-pipe.c` can be copied.

As mentioned, the pipe uses a server and a client, so first a server has to be started. The server is defined in the `void server(char *`

`data, int length)` function. The `data` parameter passes the payload data and the `length` parameter passes the payload length. In the server function a pipe is created, followed by the data being written to the pipe using `WriteFile()`.

The client is defined in the `BOOL client(char * buffer, int length)` function. The `buffer` parameter passes the data taken from the pipe and the `length` parameter is the length of the payload. In the client function `ReadFile()` is used to read the data from the pipe.

After setting the name for the pipe, the server is executed in a designated thread using the `CreateThread()` function. After a small delay the client connects to the named pipe to take the payload data and spawn the payload.

As a general rule we never want to use default values provided by templates, because these values have a higher chance of triggering signature detection. We also want to change the default values of the named pipe bypass. We can run the `strings` command on a generated payload to check what values can possibly trigger the signature detection. The output of `strings` shows a lot of information, this information also includes the string:

```
%c%c%c%c%c%c%c%c%c\netsvc\\%d
```

In the code for the bypass we can see that this string is used for assigning the pipe name. Assigning the pipe name is done with the following function:

```
1 sprintf(pipename, "%c%c%c%c%c%c%c%c%c\netsvc\\%d", 92, 92, 46, 92,  
↪ 112, 105, 112, 101, 92, (int)(GetTickCount() % 9898));
```

On execution this is translated to the string `\\.pipe\netsvc`, followed by a number generated using the tick count. The tick count is used to add some randomness to the string. The string can trigger signature detection, therefore we should replace the name with something else.

Finding a valid name can be done by looking at pipes that are already present. On a Windows system we can view all open pipes by running the following command in Powershell:

```
(get-childitem \\.pipe\).FullName
```

Having a look at the output on our test system in the open pipes, Figure 3.5 shows that many names have the same kind of structure. The structure that occurs the most is `mojo` followed by three numbers.

Figure 3.5: Open Pipes

The code on line 33 to 41 of the source code for the named pipes used by Chromium [10] reveals the structure for the names:

We can see the pipes start with the `mojo` name, which is followed by a process id, thread id, and a random integer.

Implementing the same structure for the named pipes as Chromium in our bypass results in the following code:

```
1 HANDLE hProc = GetCurrentProcess();
2 srand((int)(GetTickCount()));
3 sprintf(pipename, "%c%c%c%c%c%c%c%cLOCAL\\mojo.%d.%d.%d", 92, 92,
   ↪ 46, 92, 112, 105, 112, 101, 92, (int)(GetProcessId(hProc)),
   ↪ (int)(GetCurrentThreadId()), rand());
```

The process and thread id's are retrieved using `GetProcessId()` and `GetCurrentThreadId()`. The random number is generated using `rand()`. The seed for the random number generator is set using the tick count to increase randomness.

3.3.4 Sandbox detection

The second method we will use to bypass sandboxes combines two methods to detect if the program is executed in a sandbox. We will execute a function that can be emulated incorrectly by the sandbox and combine this function with the limited resources available to the sandbox. The blogpost of Michael Schierl about AV evasion with Metasploit [22] was used as inspiration for this bypass.

First we will post a random thread message for the current thread id using `PostThreadMessageA()`. The message will be retrieved using `PeekMessageA()`. After the message is retrieved we will check if the returned message is the same as the posted message. When thread messages are handled incorrectly, an incorrect message is returned, which indicates that the program is not executed in the intended environment. When the message check fails, execution of the payload is skipped.

As an extra safeguard we combined the message technique with a timing check. Between posting and retrieving the message we add a sleep period. Before the sleep starts we will check the tick count using `GetTickCount()`. After the sleep period is passed we will check the tick count again. The tick count should be correct relative to the provided sleep time.

Using the timing check, the sandbox has to execute the entire sleep period, otherwise the execution of the payload is skipped. Because

of the limited resources available to the sandbox, the AV may not have the time to wait for this sleep period. The AV can skip the sleep period, but skipping is detected with the tick count check.

The two methods combined result in the following code:

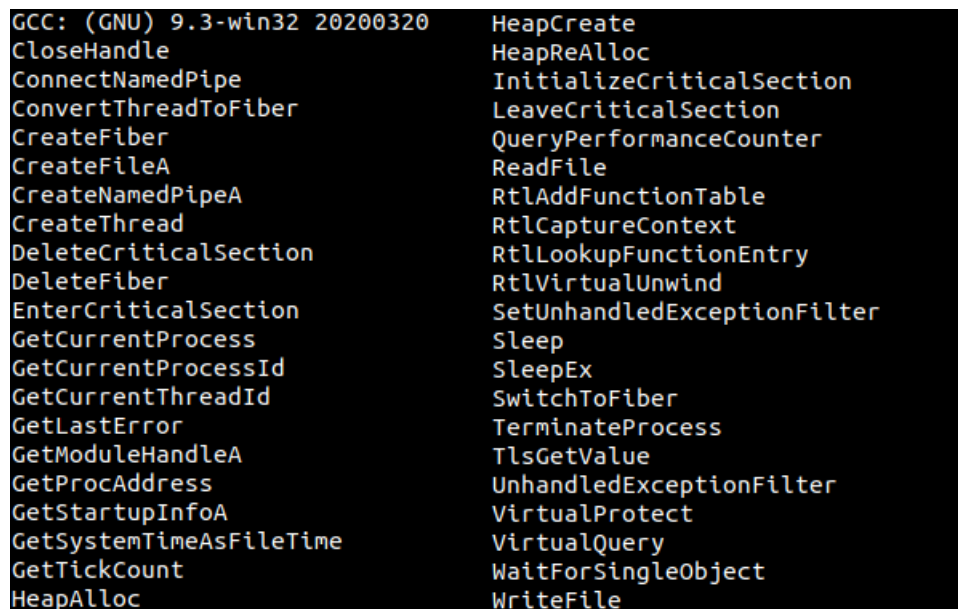
```
1 srand((int)(GetTickCount()));
2 UINT msg_in = rand();
3 MSG msg_out;
4 DWORD tc;
5
6 // Post thread message
7 PostThreadMessageA(GetCurrentThreadId(), msg_in, 23, 42);
8
9 // Check AV sandbox timing (sleeps for 65 seconds)
10 tc = GetTickCount();
11 Sleep(65000);
12 if (((GetTickCount() - tc) / 30000) != 2)
13     return;
14
15 // Check if a message can be retrieved and compare the content
16 if (!PeekMessageA(&msg_out, (HWND)-1, 0, 0, 0))
17     return;
18
19 if (msg_out.message != msg_in || msg_out.wParam != 23 ||
    ↪ msg_out.lParam != 42)
20     return;
```

Future Perspective

For future research the sandbox detection could be expanded further. Other detection methods could be used, or the timing and sleep combination could be expanded with a more complex program flow. Intertwined threads with more complicated timings would be even harder to emulate by a sandbox. Other functions that are hard to emulate or are unsupported by the sandbox could be researched. These unsupported functions could be implemented on their own or combined with other methods.

3.4 Implementing Direct Syscalls

To bypass the signature detection we also want to clean up the import table. We can take a look at our import table using the `strings` command, the results of which can be seen in the base import table Figure 3.6.



```
GCC: (GNU) 9.3-win32 20200320
CloseHandle
ConnectNamedPipe
ConvertThreadToFiber
CreateFiber
CreateFileA
CreateNamedPipeA
CreateThread
DeleteCriticalSection
DeleteFiber
EnterCriticalSection
GetCurrentProcess
GetCurrentProcessId
GetCurrentThreadId
GetLastError
GetModuleHandleA
GetProcAddress
GetStartupInfoA
GetSystemTimeAsFileTime
GetTickCount
HeapAlloc
HeapCreate
HeapReAlloc
InitializeCriticalSection
LeaveCriticalSection
QueryPerformanceCounter
ReadFile
RtlAddFunctionTable
RtlCaptureContext
RtlLookupFunctionEntry
RtlVirtualUnwind
SetUnhandledExceptionFilter
Sleep
SleepEx
SwitchToFiber
TerminateProcess
TlsGetValue
UnhandledExceptionFilter
VirtualProtect
VirtualQuery
WaitForSingleObject
WriteFile
```

Figure 3.6: Base import table using strings

The import table contains some imports that could indicate malicious activity: `HeapAlloc`, `VirtualProtect` and `CreateThread`. The three possibly malicious functions are part of the Win32 API. Information on the API and the functions can be found in the programming reference [16]. `HeapAlloc` is used to allocate memory from a heap, `VirtualProtect` is used to change the protection on a region of memory, and `CreateThread` is used to create a thread in the designated address space.

The combination of memory allocation, changing memory protection and creating a thread can indicate that shellcode is loaded. Not only can the combination trigger the signature checking, but the combination can also trigger the hooks that most AV or EDR products have on the Win32 API. Replacing or removing the usage of this combination of functions can help us bypass both AV and EDR.

3.4.1 Bypassing the hooks

First we have to take a look at how the Win32 API interacts with applications and the operating system, Figure 3.7 shows an overview of the Windows Programming Interface and the position of the Win32 API.

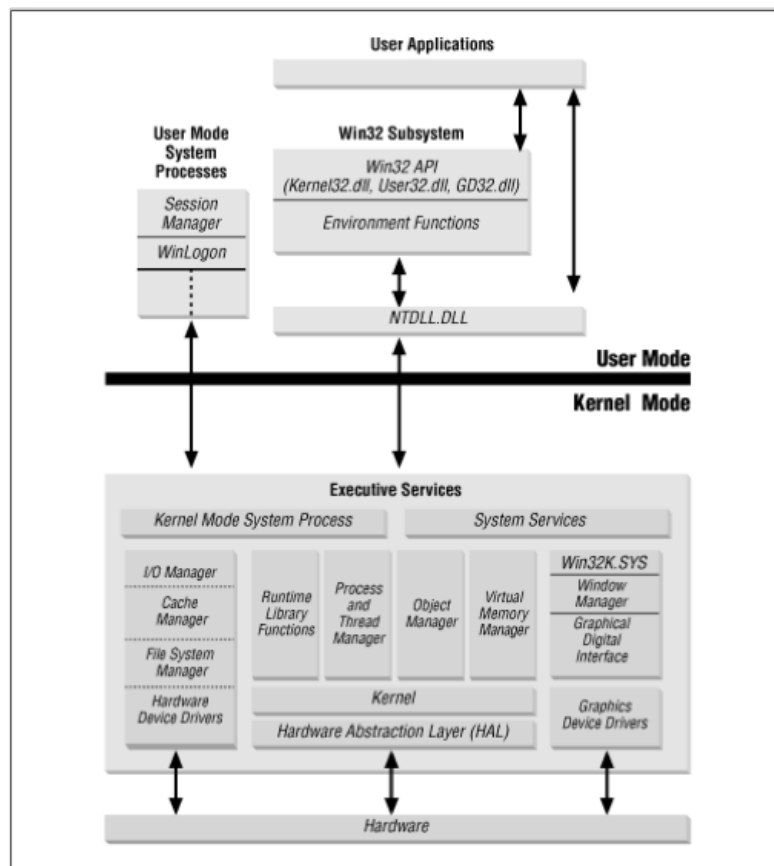


Figure 3.7: Windows Programming Interface

We can see that the interface is split into two security levels, User Mode and Kernel Mode. Normally applications are always run in User Mode, so for the application to interact with the operating system a developer would normally use the Win32 API. This API interacts with the Native API, which is also called NTDLL. NTDLL can be

seen as the gateway between applications and the operating system. The Win32 API is well documented, but there is no official documentation for NTDLL, so for normal development the Win32 API should always be used.

Because NTDLL functions as a gateway between the user and kernel level, most AV and EDR products have hooks on NTDLL. To bypass the hooks we have to find a way around the usage of the Win32 API and the underlying NTDLL. One way was described by Outflank in one of their Red Team Tactics blog posts [20]. The blog post was the inspiration for our implementation.

If we want to find a way around the usage of the Win32 API, we have to take a look at the functionality, specifically when we call our suspicious functions `HeapAlloc`, `VirtualProtect` and `CreateThread`. First we will run our payload in a debugger. We will use the WinDbg program on our test system. We start debugging by opening our payload using the launch executable option. To quickly find the three functions we will set breakpoints on the usage of the functions and run the program.

When we reach our breakpoints we can see the NTDLL functions that are used: `HeapAlloc` calls `NtAllocateVirtualMemory`, `CreateThread` calls `NtCreateThreadEx`, and `VirtualProtect` calls `NtProtectVirtualMemory`. Now we know the underlying Nt functions we can determine the functionality of these Nt functions. To determine the functionality, we have to step into the previously found Nt functions. In Figure 3.8 we can see the disassembled `NtCreateThreadEx` instruction as an example.

| ntdll!NtCreateThreadEx: | | | |
|-------------------------|------------------------------------|---------|--------------------------------------------|
| 1 - | 00007ffb`c68c5400 4c8bd1 | mov | r10, rcx |
| 2 - | 00007ffb`c68c5403 b8c5000000 | mov | eax, 0C5h |
| 3 - | 00007ffb`c68c5408 f604250803fe7f01 | test | byte ptr [7FFE0308h], 1 |
| 4 - | 00007ffb`c68c5410 7503 | jne | ntdll!NtCreateThreadEx+0x15 (7ffbc68c5415) |
| 5 - | 00007ffb`c68c5412 0f05 | syscall | |
| 6 - | 00007ffb`c68c5414 c3 | ret | |
| 7 - | 00007ffb`c68c5415 cd2e | int | 2Eh |
| 8 - | 00007ffb`c68c5417 c3 | ret | |
| 9 - | 00007ffb`c68c5418 0f1f840000000000 | nop | dword ptr [rax+rax] |

Figure 3.8: Disassembled `NtCreateThreadEx` function

First the function moves the arguments on the stack (line 1), then the function puts the system call number in the `eax` register (line 2), followed by checking if the correct function name is called (lines 3 and 4). Finally the function executes the `syscall` (line 5). After the

syscall the CPU will enter kernel mode, finds the corresponding API call to the system call number and executes this API call. When the call is finished the function will return back to user mode including the return values of the API call.

Now we have found the syscall number for each of the three suspicious functions. The only problem is that these syscall numbers change for different versions of Windows. Instead of running a debugger each time, we can find these numbers a lot easier by loading `ntdll.dll` in a disassembler program like `IDA`. In `IDA` we can quickly search for each function, but searching requires which `NTDLL` function corresponds to which `Win32` API function. In Figure 3.9 we can see the disassembled `NtProtectVirtualMemory` in `IDA` as an example.

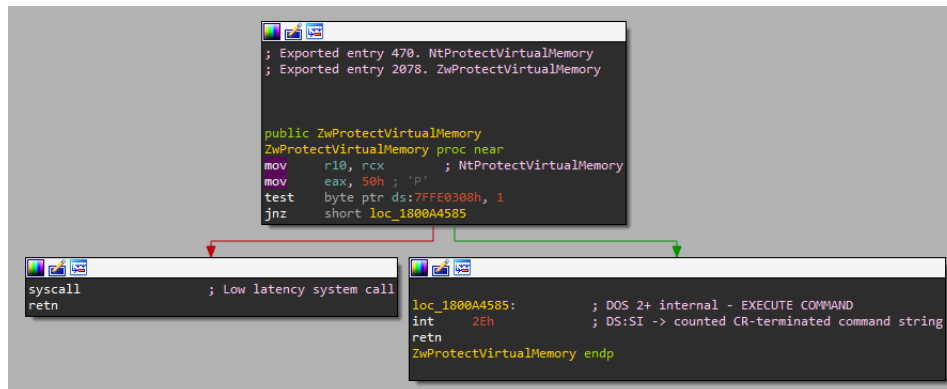


Figure 3.9: Disassembled `NtProtectVirtualMemory` in `IDA`

3.4.2 Moving to SysWhispers

Even if we can find the syscall numbers easier, implementing the numbers for every version takes a lot of time. To counter having to implement the numbers every time, the `SysWhispers` project was created. `SysWhispers` is a tool which generates header and assembly pairs to make it easier to use direct syscalls.

Instead of having to look up the syscall numbers manually, the code will find the `DLLBase` address of `ntdll.dll` on runtime by looping through the full list of imported dlls. Looping through the full list is necessary, because `NTDLL` is not always in the same position. When `NTDLL` is found, the code loads all included syscall entries in a list. The list can later be accessed to find the correct syscall number.

Loading such a list on runtime is compatible with all windows versions without having to alter the code each time.

For our implementation we used the newest version of Syswhispers, which is `SysWhispers3`. `SysWhispers3` can be installed using the instructions on the GitHub page. Using `SysWhispers` we generate our three functions. The generation can be done using the following command:

```
python3 ./syswhispers.py --functions
↳ NtProtectVirtualMemory,NtWriteVirtualMemory,NtCreateThreadEx
↳ -o syscalls
```

This command generates three files: `syscalls.c`, `syscalls.h` and `syscalls.asm`. When we have a look at the code we can see that the Nt functions are defined like the NtInternals NTAPI Undocumented Functions [19], to make sure the functions behave the same as their NTDLL counterparts.

As an example we will have a look at the `NtProtectVirtualMemory` function. In Table 3.1 we can see the `SysWhispers3` version side by side with the NTDLL version.

| SysWhispers3 | NTDLL |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Definition NtProtectVirtualMemory(IN HANDLE ProcessHandle, IN OUT PVOID *BaseAddress, IN OUT PSIZE_T RegionSize, IN ULONG NewProtect, OUT PULONG OldProtect) | Definition NtProtectVirtualMemory(IN HANDLE ProcessHandle, IN OUT PVOID *BaseAddress, IN OUT PULONG ↪ NumberOfBytesToProtect, IN ULONG ↪ NewAccessProtection, OUT PULONG ↪ OldAccessProtection) |
| Assembly mov [rsp+8], rcx mov [rsp+16], rdx mov [rsp+24], r8 mov [rsp+32], r9 sub rsp, 28h mov ecx, 00E93061Ch call SW3_GetSyscallNumber add rsp, 28h mov rcx, [rsp+8] mov rdx, [rsp+16] mov r8, [rsp+24] mov r9, [rsp+32] mov r10, rcx syscall ret | Assembly mov r10, rcx mov eax, 50h test byte ptr ds:7FFE0308h jnz short loc_1800A4585 syscall retn |

Table 3.1: Comparison of SysWhispers3 and NTDLL

As we can see in the table, the definition is the same, but the assembly instructions are a bit different. The NTDLL version checks if the loaded function name is correct, which is an extra error check. The SysWhispers version first saves the used registers to restore them afterwards, this restoration is included to not disrupt the normal program flow.

Instead of loading the syscall number, the SysWhispers version calls the function SW3_GetSyscallNumber. This is the function that looks for the correct syscall number based on the function name, and stores the number in the `eax` register. Looking for the syscall numbers in the list is the part which makes the code compatible with all Windows versions. We can see that the assembly syntax is different, but, as shown, the semantics are the same.

3.4.3 Implementing the functions

Knowing the SysWhispers functions should work correctly compared to their NTDLL version, the functions can be implemented in the Artifact Kit. To use the generated SysWhispers3 functions, the files `syscalls.c` and `syscalls.h` have to be included in the build config. To support the 64 bits assembly instructions the `-masm=intel` flag also has to be used. This flag tells the compiler to generate the assembler output file using Intel syntax. Now the functions can be used in the Artifact Kit code by including the `syscalls.h` header.

The next step is to replace all the occurrences of the functions with their Nt counterpart. The definitions of the old functions can be found in the Win32 API documentation [16], and the definitions of their replacements can be found in the `syscalls.h` header or on the NtInternals site [19].

First we will be replacing all the instances of `CreateThread`. In the code the instances are used in `patch.c` and the previously created `bypass-secura.c`. We start with translating the `CreateThread` used in `bypass-secura.c`. The following codeblock shows the two different versions:

```
1 // Original
2 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)&server_thread,
   ↪ (LPVOID) NULL, 0, NULL);
3
4 // Using direct syscall
5 HANDLE hProc = GetCurrentProcess();
6 HANDLE thandle = NULL;
7 NtCreateThreadEx(&thandle, GENERIC_EXECUTE, NULL, hProc,
   ↪ (LPTHREAD_START_ROUTINE)&server_thread, NULL, FALSE, 0, 0, 0,
   ↪ NULL);
```

Before using `NtCreateThreadEx` the current process handle has to be obtained and a thread handle has to be initialized. We can use the same process handle that was earlier defined for the named pipes Section 3.3.3.

The translation in `patch.c` is a bit different, because in this translation we have to pass an argument to the function that is run in the thread. The following code block shows the two different versions:

```

1 // Original
2 CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)&run, ptr, 0, NULL);
3
4 // Using direct syscall
5 HANDLE hProc = GetCurrentProcess();
6 HANDLE thandle = NULL;
7 NtCreateThreadEx(&thandle, GENERIC_EXECUTE, NULL, hProc,
  ↪ (LPTHREAD_START_ROUTINE)&run, ptr, FALSE, 0, 0, 0, NULL);

```

Now we move on to the next function, this function is VirtualProtect. VirtualProtect is used twice in spoof.c, which also includes an if-statement. Replacing VirtualProtect in the if-statement should be no problem, as the returned NTSTATUS value by NtProtectVirtualMemory can be used in the same way as the returned boolean of VirtualProtect. The following code block shows how both functions are replaced:

```

1 // First original function
2 if (!VirtualProtect(addressToHook, dwSize, PAGE_EXECUTE_READWRITE,
  ↪ &oldProt)) {
3     return 0;
4 }
5
6 // First replacement
7 HANDLE hProc = GetCurrentProcess();
8 if (!NtProtectVirtualMemory(hProc, (PVOID)&addressToHook,
  ↪ (PSIZE_T)&dwSize, PAGE_EXECUTE_READWRITE, &oldProt)) {
9     return 0;
10 }
11
12 // Second original function
13 VirtualProtect(addressToHook, dwSize, oldProt, &oldProt);
14
15 // Second replacement
16 NtProtectVirtualMemory(hProc, (PVOID)&addressToHook,
  ↪ (PSIZE_T)&dwSize, oldProt, &oldProt);

```

In the replacement a process handle has to be used. We can use the same process handle we used previously. Some typecasting is required to make the original values compatible with the Nt function.

The final function will be HeapAlloc, which is only used in patch.c. As we have seen before, HeapAlloc eventually calls NtAllocateVirtualMemory. Instead of rewriting every part of the heap allocation pro-

cess, we step away from using heap allocation, and use the Nt function `NtAllocateVirtualMemory` directly.

When using the Nt function an extra memory protection fix has to be used. The extra protection is done with `NtProtectVirtualMemory`. The heap creation process includes the protection change, but `NtAllocateVirtualMemory` does not. The following codeblock shows how we replaced `HeapAlloc`:

```
1 // Original memory allocation using HeapAlloc
2 HANDLE heap;
3 heap = HeapCreate(HEAP_CREATE_ENABLE_EXECUTE, 0, 0);
4 ptr = HeapAlloc(heap, 0, 10);
5
6 // Replacement using NtAllocateVirtualMemory and
  ↳ NtProtectVirtualMemory
7 NtAllocateVirtualMemory(hProc, (PVOID)&ptr, 0, (PSIZE_T)&length,
  ↳ MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
8
9 DWORD old;
10 NtProtectVirtualMemory(hProc, (PVOID)&ptr, (PSIZE_T)&length,
  ↳ PAGE_EXECUTE_READ, &old);
```

Now all the occurrences of our suspicious functions are replaced. The artifacts have to be built and the updated payloads can be generated. We run the strings command again on the generated payload, Figure 3.10 shows the resulting import table.

| | |
|-------------------------------|-----------------------------|
| GCC: (GNU) 9.3-win32 20200320 | InitializeCriticalSection |
| CloseHandle | LeaveCriticalSection |
| ConnectNamedPipe | QueryPerformanceCounter |
| ConvertThreadToFiber | ReadFile |
| CreateFiber | RtlAddFunctionTable |
| CreateFileA | RtlCaptureContext |
| CreateNamedPipeA | RtlLookupFunctionEntry |
| DeleteCriticalSection | RtlVirtualUnwind |
| DeleteFiber | SetUnhandledExceptionFilter |
| EnterCriticalSection | Sleep |
| GetCurrentProcess | SleepEx |
| GetCurrentProcessId | SwitchToFiber |
| GetCurrentThreadId | TerminateProcess |
| GetLastError | TlsGetValue |
| GetModuleHandleA | UnhandledExceptionFilter |
| GetProcAddress | VirtualProtect |
| GetStartupInfoA | VirtualQuery |
| GetSystemTimeAsFileTime | WaitForSingleObject |
| GetTickCount | WriteFile |

Figure 3.10: Updated import table using strings

As we can see in the figure, two of the suspicious functions are no longer imported, the only one that is still present is `VirtualProtect`. We replaced all our usages of `VirtualProtect`, but `VirtualProtect` is still imported. The function is still imported, because some other included standard functions also use `VirtualProtect` internally. We can never completely remove `VirtualProtect` from the import table, but all usages of `VirtualProtect` that can be considered suspicious are replaced.

The final step is to check if the payload still works as intended. When we moved the payload to the test system and executed the payload, the Beacon checked in as normal in the Cobalt Strike client. Interacting with the beacon also worked as before, so we can assume that our replacements work as intended.

If we take a look at the Microsoft 365 Defender portal we notice that the EDR takes a lot longer before an active instance of Cobalt Strike is spotted. At first an active instance would be reported immediately but now spotting takes a couple of minutes of interaction before the EDR flags the payload as malicious. The Defender 365 portal states that the detection source is the behaviour of the program.

Future Perspective

In future research more Win32 API functions could be implemented using direct syscalls. Another method for bypassing the hooks could also be implemented. For instance a fresh copy of NTDLL without hooks could be loaded on execution.

3.5 Malleable Profiles

Malleable Command and Control (C2) profiles control the behavior of the Beacon on the system and the network. For instance, C2 profiles can be used to simulate normal web traffic or copy the characteristics of a real threat actor. In our case we need to make sure the traffic looks as normal as possible. We have to blend in with normal traffic to not raise any suspicion. What is considered normal traffic can be different for each target environment, therefore we have to find a universal profile or find a way to modify the profile for each target environment.

3.5.1 Profile structure

A C2 profile consist of multiple sections, where each section defines a part of the behavior. We will first take a look at what every section is used for based on the Malleable Command and Control section of the Cobalt Strike User Guide [5].

Profile Name

In this section the name of the profile is defined. The profile name has no influence on the behavior of the Beacon, the name is only used in reports.

Sleep Times

The sleep time settings control the time between check ins of the Beacon. The time is defined in milliseconds. In this section we can also define the jitter. The jitter is a percentage that defines the range around the sleep time in which the beacon can check in. The jitter adds some randomness to the check ins. The Beacon becomes harder to detect when the interval is higher and less predictable.

Server response size jitter

The server response size jitter adds a string of random length to the http-get and http-post server responses. The provided value defines the maximum length of the string.

User-Agent

The User-Agent value defines the User-Agent for the traffic. An agent should be selected that fits in with the target environment. The best agent would be one that is actually present within the target network.

SSL Certificate

This section defines the used SSL certificate for the HTTPS traffic. Best practice would be to use a real and correctly issued SSL certificate for the domain that is used. A trusted certificate makes it harder to detect for the defenders.

SpawnTo Process

This section defines what process will be used by a beacon to spawn to for post-exploitation, that is why we can also call this section the post-exploitation options. The used process can be defined individually for 32 and 64 bits. We can also specify the command line arguments to help blend in even more.

SMB Beacons

The Cobalt Strike SMB Beacons use named pipes to communicate through their parent Beacon. The SMB Beacons can be used for peer-to-peer communication between Beacons on either the network or the same host. The names of the pipes can be defined in this section.

DNS Beacons

The Cobalt Strike DNS Beacons use DNS for their traffic. A DNS Beacon can be best used as a slow backup in case other Beacons get detected. In the DNS Beacon section we can define the DNS requests of the Beacon.

Staging Process

The staging process is used in staged payloads. When the smaller staged payload is executed, the stager has to download the full Beacon. In Cobalt Strike the downloading of the full Beacon is done using a HTTP request. The behavior of this HTTP request is defined in the Staging Process section. Staging can also be disabled entirely.

Memory Indicators

AV and EDR products look at the properties of memory to detect malicious activity in a process. The Memory Indicators section provides options for In-Memory Evasion, by controlling how the Beacon is loaded into memory. The creator of Cobalt Strike, Raphael Mudge, has a series of video's for a more in-depth look at the memory problems. [18]

HTTP GET

The HTTP GET section defines the requests the Beacon sends to the team server to check for new tasks. Here we can define the URI, content, and the headers for the client request and server response.

HTTP POST

The HTTP Post section defines the response of the Beacon to commands that are issued by the server. Despite the name, this can be done as both a HTTP POST and HTTP GET request. The requests can be defined in the same way as done in the HTTP GET Section 3.5.1.

The final syntax of a profile has to be checked for errors. Cobalt Strike includes a tool to check profiles for errors; c2lint. We will use this to check the profiles for any syntax errors. The semantics of the profile can be tested using a network traffic inspector.

For our research we will only focus on the network traffic parts of the profiles. In future research the profiles could be expanded for other parts of the profiles, like the memory indicators (Section 3.5.1) or the post-exploitation (Section 3.5.1).

3.5.2 Situational profile

When we want to hide our traffic, we should have a look at the traffic that is already present in the target network. For our traffic to

not stand out, it should blend in with the rest. We can capture packets in the target network, and look if we can implement the packet in a profile.

As an example we will take a look at the Radboud University network. There are a lot of sites that people on campus can visit often. We will take the website of the [University Library](#) as an example. When a user visits the website of the University Library, the client will always send a POST request to `read.aspx` for JSON data. The POST request is sent for every section on the site, which means that the request is sent a lot when a user visits the site. The request also includes enough data to hide our traffic in.

First we will have to capture the request and response pair to `read`. Capturing this pair can be done using [Burp Suite](#). Burp Suite is an application used for analysing and testing the security of web applications. After we captured the pair using the Burp Suite proxy, we can save the pair on our device. The request and response pair is also included as [Appendix A.1](#) for reference.

Now we have to translate this request to a profile. Translating can be done manually, but an easier way is to use the [Burp2Malleable](#) script, a tool by CodeXTF2 that turns request and response pairs into the network traffic section of C2 profiles. We run the script on our stored request and response using the command:

```
python3 burp2malleable.py request.txt response.txt
```

The script presents the user with options for where to store metadata, id's, tasks and responses of the Beacon. Again we have to make sure the traffic looks as normal as possible, so we want to store the Cobalt Strike data in such a way that the packets look as normal.

We chose to store the metadata in the Cookie header. We stored the Beacon id as extra parameter "id" in the URI. The Beacon responses are stored in the body of the request, replacing the value of `VarIdt`. Finally the Beacon tasks are stored in the body of the response, replacing the value for `id` of the element.

To check for any errors we run `c2lint` on the resulting profile. The check returned two errors about the header sizes. Both the header for the request and response were too large, which can result in instability. To fix the errors some sections of the headers have to be

moved or removed.

Headers can also be defined in the `http-config` section, but the Burp2Malleable script does not use this section. We manually moved the Date, Server, Content-Length, Connection, and Content-Type headers to the `http-config` section to save space. We can move these headers to the `http-config`, because these headers are either the same for each request, or can be set automatically by the client. Together with removing non-essential headers we saved enough space to fit in the size limit.

The Burp2Malleable script can only be used for the `http-get` and `http-post` sections of the profile. We had to add sections for the sleep time ourselves. The full profile can be seen on [GitHub](#).

After finishing the profile we can load the profile on our Team Server. The behavior of the profile can be checked using any network monitoring tool. We used [Fiddler Everywhere](#), because Fiddler can be easily used to read HTTPS traffic from applications using a tunnel and local proxy. Reading the content of HTTPS traffic is necessary to confirm that our profiles work as intended.

When we run a payload on the test system the beacon checks in correctly. The university library profile [Figure 3.11](#) shows the post request to the domain of our team server with the correct request and response according to the profile. The traffic imitates the request and response of the university library website as intended.

| # | URL | HTTP Version | Status... | Method | Process | Request Time |
|---|-----------------------|--------------|-----------|---------|---------|--------------|
| 1 | http://[redacted]... | HTTP/1.0 | 200 | CONNECT | ub:8264 | 06:48:32.166 |
| 2 | https://[redacted]... | HTTP/1.1 | 200 | POST | ub:8264 | 06:48:32.492 |

Request

https://[redacted] HTTP/1.1 POST

Headers (10) Params (0) Cookies (8) Raw Body

| Key | Value |
|-----------------|----------------------------------------------------------------------------------|
| Accept | application/json, text/javascript, */*; q=0.01 |
| Content-Type | application/x-www-form-urlencoded; charset=UTF-8 |
| Referer | https://www.ru.nl/ |
| Accept-Encoding | gzip, deflate |
| Cookie | cookie-agreed-version=1.0.0; cookie-agreed=2; _fbp=fb.1.1670593551589.1024575... |
| User-Agent | Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv:11.0) like Gecko |
| Host | [redacted].nl |
| Content-Length | 0 |
| Connection | Keep-Alive |
| Cache-Control | no-cache |

Response

CERTIFICATE EXPIRING

BODY: 471 B HTTP/1.1 200

Headers (16) Cookies (0) Raw Preview Body

| Key | Value |
|-----------------------------|----------------------------------------------|
| Content-Type | application/json |
| Content-Length | 364 |
| Connection | keep-alive |
| Cache-Control | public |
| Vary | Host |
| P3P | CP="NOI DSP MON CUR ADM DEV TAI OUR NOR STA" |
| X-Frame-Options | SAMEORIGIN |
| Permissions-Policy | fullscreen=self, geolocation=self |
| Access-Control-Allow-Origin | https://radboudgezichten.ru.nl |
| X-Content-Type-Options | nosniff |
| Referrer-Policy | strict-origin |
| X-Cache | MISS from roesje02.uci.ru.nl |
| X-Cache-Lookup | MISS from roesje02.uci.ru.nl:80 |

Figure 3.11: Traffic for University Library Profile

3.5.3 Universal profile

Making a profile for each environment will increase the effectiveness of the profile, but the level of effort is also increased. The effort would be a lot less if a profile could be used in multiple environments. To create a universal profile we have to know what network traffic can commonly be found on systems in corporate networks. Common traffic can be from applications that are used by employees, like mail clients or collaboration tools. Common traffic can also be web requests that are often sent by websites regardless of the environment.

According to a survey conducted by Stack Overflow [21], the top 3 most used web frameworks and technologies used by Professional Developers are Node.js, React.js, and jQuery. jQuery is a single JavaScript library that has to be loaded to be used in a webpage. The fact that it has to be loaded for each webpage using jQuery makes it interesting compared to Node.js and React.js. Each time a webpage that uses jQuery is loaded, a file called `jQuery-[version].js` is also loaded. Because jQuery is used on many websites, the request for the file is common in many environments.

Because jQuery is this common, a great universal profile could be one that is able to imitate loading the jQuery file. We can start again by making a profile manually or using the Burp2Malleable script as basis. Luckily a lot of time can be saved by using the jQuery profile template provided on the [Threat Express GitHub](#). We edited the template slightly to better fit in with our used payloads and infrastructure. The network traffic section of the final profile can be found on [GitHub](#).

Now we can load the profile on the team server, generate a payload, and run the payload on the test system. Afterwards the beacon checked in correctly. We can confirm the traffic was according to the profile using Fiddler. Using a tunnel through a local proxy we can see the traffic acts as specified in the profile. The stripped traffic is included in Figure 3.12. We can see that the requests and responses imitate jQuery as intended.

| # | URL | HTTP Version | Status... | Method | Process | Request Time |
|---|-----------------------|--------------|-----------|---------|---------|--------------|
| 1 | http://[redacted]... | HTTP/1.0 | 200 | CONNECT | jq:1904 | 07:41:52.391 |
| 2 | https://[redacted]... | HTTP/1.1 | 200 | GET | jq:1904 | 07:41:52.768 |
| 3 | http://[redacted]... | HTTP/1.0 | 200 | CONNECT | jq:1904 | 07:41:52.853 |
| 4 | https://[redacted]... | HTTP/1.1 | 200 | POST | jq:1904 | 07:41:52.887 |

Request

https://[redacted]... HTTP/1.1 POST

Headers (8) Params (1) Cookies (0) Raw Body

| Key | Value |
|-----------------|-----------------------------------------------------------------|
| Accept | text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 |
| Referer | http://code.jquery.com/ |
| Accept-Encoding | gzip, deflate |
| User-Agent | Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv:11.0) like Gecko |
| Host | [redacted].nl |
| Content-Length | 1398 |
| Connection | Keep-Alive |
| Cache-Control | no-cache |

Response

CERTIFICATE EXPIRING BODY: 5.54 KB HTTP/1.1 200

Headers (7) Cookies (0) Raw Preview Body

| Key | Value |
|----------------|---------------------------------------|
| Server | nginx/1.18.0 (Ubuntu) |
| Date | Fri, 16 Dec 2022 15:41:53 GMT |
| Content-Type | application/javascript; charset=utf-8 |
| Content-Length | 5671 |
| Connection | keep-alive |
| Cache-Control | max-age=0, no-cache |
| Pragma | no-cache |

Figure 3.12: Traffic for jQuery

Even with the traffic imitations the payload still gets spotted. Having a look at all the network traffic using the **Microsoft Network Monitor** tool, we can see that directly after the beacon sends a DNS request for the team server, a DNS request for the domain `eu-v20.events.data.microsoft.com` is also sent. The DNS requests can be seen in the DNS Requests Figure 3.13.

| Protocol Name | Description |
|---------------|---------------------------------------------------------------------------------------------------------------------------|
| 1) DNS | DNS:QueryId = 0x2E7A, QUERY (Standard query), Query for [redacted] Team Server Domain .nl of type Host Addr on class I... |
| 2) DNS | DNS:QueryId = 0xCF30, QUERY (Standard query), Query for eu-v20.events.data.microsoft.com of type Host Addr on class... |

Figure 3.13: DNS Requests

According to the Defender for Endpoint URL list published by Microsoft [14], the `eu-v20` domain is used for EDR Cyber Data. The do-

main confirms that the sample is submitted to the EDR. When we have a look at the EDR dashboard we can see that both executables are still flagged as suspicious, but the detection was done using the AV, not the EDR. The detection by the AV shows that payloads generated with a jQuery profile trigger the signature detection again.

Future Perspective

In future research the profiles could be expanded further. Sections for in-memory evasion and post-exploitation could be added to further decrease the detection rate. Extra research could also be done on other common network traffic to imitate. The Burp2Malleable script could also be expanded upon by including a way to automate the other sections of the profile besides the network traffic.

Chapter 4

Related Work

Marpuang et al. give a great overview of existing malware evasion techniques for AV [11]. They split up the evasion techniques further and explain them in more detail. The division contains seven concepts: Obfuscation, fragmentation, application specific violations, protocol violations, inserting traffic, denial of service, and code reuse attacks. The more detailed insights give a great overview of all the existing methods, however the implementation is not discussed. The second part of the paper looks into possible mitigations of the previously mentioned evasion methods.

Kalogranis uses existing AV evasion techniques and their mitigations, implements these techniques, and uses them to test popular AV products [8]. Testing the AV evasion techniques places the effectiveness of them in a real world perspective.

Chailtyko et al. also looks at the effectiveness of AV evasion methods, but focuses specifically on sandboxes [3]. In the research they look into a popular open-source sandbox product. The detection methods of the sandbox are investigated, and the problems with these detection methods are discussed. The research focuses more on the defending side by providing a solution for all of the found problems.

Not only AV is represented, great research has been done in EDR products as well. Arfeen et al. give insight in the motivation and the functionality behind EDR [1]. The research gives more insights than the vendors like CrowdStrike [4] give by also showing the limitations of the product.

Research by Karantzas et al. assesses the security of EDR products

using Cobalt Strike [9]. Different bypassing methods than we used, like DLL sideloading, were tested for the detection rate of popular EDR products. The focus is not on the bypassing methods, but how the EDR handles them, therefore the paper gives a good insight in how EDR works in real scenarios.

No specific research has been done on bypassing network detection, however the detection itself has been researched. Moussaileb et al. dive into malicious traffic generated by ransomware [17]. Bekerman et al. have a more general approach to malware network traffic detection [2]. The research focuses on detecting unknown malware using machine learning, specifically classification. Both studies show where the detection focuses on when looking for malicious traffic, which is very useful for developing evasion techniques.

Chapter 5

Conclusions

There are many ways to bypass AV and EDR, we only showed some of these methods. From our research we can conclude that the methods we implemented showed improvement for the detection rate of the Cobalt Strike Beacons.

Signature evasion methods can be implemented on many levels. Basic methods are easy to implement and can already show great improvement in detection rate. More advanced signature evasion methods take more time, but spending more time results in an even lower detection rate. For the evasion of AV and EDR, implementing signature evasion methods are very useful as the methods greatly reduce the detection rate of AV products with a relative small amount of work.

Behavior detection bypassing methods can also be implemented in different parts of the Beacons. Sandbox detection, together with unsupported functions like named pipes, can be implemented in a small number of lines. The sandbox evasion methods will reduce the detection rate of AV products. Implementing direct syscalls for evasion of API hooks takes more time, even when the implementation is made easier using SysWhispers. The extra effort does pay off, as using direct syscalls reduces the detection rate of both AV and EDR.

Malleable profiles can modify the behavior of the payload in many ways, which enables the user to adapt to any environment. Making a good profile for every environment takes a lot of time, but makes the payload a lot harder to detect. Using a universal profile takes a lot less time, but can possibly increase the detection rate compared to situational profiles. Using a malleable profile, either universal or situational, does decrease the detection rate compared to not us-

ing a profile at all, therefore malleable profiles are very useful for bypassing AV and EDR.

All methods we researched showed a decrease in detection rate, but the Red Team should be careful when combining multiple methods. Implementing one evasion method can result in detection by another method. One example is the Malleable Profile, while the network detection decreases, the payload gets spotted again by the AV signature detection. Another example is the sandbox evasion, when we implemented the named pipe, we also had to change the name of the pipe to avoid signature detection. Balancing between bypassing methods is very important when we want to evade both AV and EDR.

As shown in the Future Perspective sections for each method (Section 3.3.4, Section 3.4.3, and Section 3.5.3), the methods we have researched show a lot of possibilities for future adaptation and improvement. Future perspective is very important when the AV and EDR products will also keep updating the detection methods in the future.

Bibliography

- [1] Asad Arfeen et al. Endpoint Detection Response: A Malware Identification Solution. url: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9703010&tag=1>.
- [2] Dmitri Bekerman et al. Unknown Malware Detection Using Network Traffic Classification. url: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7346821&tag=1>.
- [3] Alexander Chailytko and Stanislav Skuratovich. Defeating Sandbox Evasion: How To Increase The Successful Emulation Rate In Your Virtual Environment. url: https://blog.checkpoint.com/wp-content/uploads/2016/10/DefeatingSandBoxEvasion-VB2016_CheckPoint.pdf.
- [4] CrowdStrike. What is Endpoint Detection and Response (EDR)? url: <https://www.crowdstrike.com/cybersecurity-101/endpoint-security/endpoint-detection-and-response-edr/>.
- [5] Fortra. Malleable Command and Control User Guide. url: https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/malleable-c2_main.htm#_Toc65482834.
- [6] SANS Institute. What is Anti-Virus? url: https://www.rwu.edu/sites/default/files/downloads/it/what_is_anti_virus.pdf.
- [7] Center for Internet Security. Signature-Based vs Anomaly-Based Detection. url: <https://www.cisecurity.org/insights/spotlight/cybersecurity-spotlight-signature-based-vs-anomaly-based-detection>.
- [8] Christos Kalogranis. AntiVirus Software Evasion: An Evaluation Of The AV Evasion Tools. url: <https://dione.lib.unipi.gr/xmlui/handle/unipi/11232>.
- [9] George Karantzas and Constantinos Patsakis. An Empirical Assessment of Endpoint Detection and Response Systems against Advanced Persistent Threats Attack Vectors. url: <https://www.mdpi.com/2624-800X/1/3/21>.

- [10] Google LLC. Chromium Named Platform Channel Source Code. url: https://raw.githubusercontent.com/chromium/chromium/c4d3c31083a2e1481253ff2d24298a1dfe19c754/mojo/public/cpp/platform/named_platform_channel_win.cc.
- [11] Jonathan A.P. Marpaung, Mangal Sain, and Hoon-Jae Lee. Survey on malware evasion techniques: state of the art and challenges. url: <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6174775&tag=1>.
- [12] Microsoft. Behavioral blocking and containment. url: <https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/behavioral-blocking-containment?view=o365-worldwide>.
- [13] Microsoft. Microsoft Defender for Endpoint Overview. url: <https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/microsoft-defender-endpoint?view=o365-worldwide>.
- [14] Microsoft. Microsoft Defender for Endpoint URL list for commercial customers. url: <https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/configure-proxy-internet?view=o365-worldwide#enable-access-to-microsoft-defender-for-endpoint-service-urls-in-the-proxy-server>.
- [15] Microsoft. Named Pipes Documentation. url: <https://learn.microsoft.com/en-us/windows/win32/ipc/named-pipes>.
- [16] Microsoft. Programming reference for the Win32 API. url: <https://learn.microsoft.com/en-us/windows/win32/api/>.
- [17] Routa Moussaileb et al. Ransomware Network Traffic Analysis for Pre-encryption Alert. url: https://link.springer.com/chapter/10.1007/978-3-030-45371-8_2.
- [18] Raphael Mudge. In-Memory Evasion. url: <https://www.cobaltstrike.com/blog/in-memory-evasion/>.
- [19] NTinternals. NTAPI Undocumented Functions. url: <https://undocumented.ntinternals.net/>.
- [20] Outflank. Red Team Tactics: Combining Direct System Calls and sRDI to bypass AV/EDR. url: <https://outflank.nl/blog/2019/06/19/red-team-tactics-combining-direct-system-calls-and-srdi-to-bypass-av-edr/>.
- [21] Stack Overflow. 2022 Developer Survey. url: <https://survey.stackoverflow.co/2022/#most-popular-technologies-webframe-prof>.

- [22] Michael Schierl. Facts and myths about antivirus evasion with Metasploit. url: <http://schierlm.users.sourceforge.net/avevasion.html>.
- [23] Secura. Penetration Testing. url: <https://www.secura.com/services/information-technology/vapt/penetration-testing>.
- [24] Secura. Red Teaming. url: <https://www.secura.com/services/information-technology/vapt/red-teaming>.
- [25] Secura. Vulnerability Scan Assessment. url: <https://www.secura.com/services/information-technology/vapt/vulnerability-scan-assessment>.
- [26] National Cyber Security Centre UK. What is an antivirus product? Do I need one? url: <https://www.ncsc.gov.uk/guidance/what-is-an-antivirus-product>.

Appendix A

Request and Response pairs

A.1 Request and response read.aspx page

A.1.1 Request

```
1 POST /aspx/read.aspx HTTP/1.1
2 Host: www.ru.nl
3 Cookie: cookie-agreed-version=1.0.0; cookie-agreed=2;
   ↪ _fbp=fb.1.1670593551589.1024575632; _ga=GA1.2.23700656.1670593551;
   ↪ _ga_6C86NZ2VXH=GS1.1.1670593550.1.1.1670593571.0.0.0;
   ↪ cookie_notification=functional; _gid=GA1.2.2062466150.1670764872;
   ↪ cookies_consent=-1
4 Content-Length: 63
5 Sec-Ch-Ua: "Chromium";v="107", "Not=A?Brand";v="24"
6 Accept: application/json, text/javascript, */*; q=0.01
7 Content-Type: application/x-www-form-urlencoded; charset=UTF-8
8 X-Requested-With: XMLHttpRequest
9 Sec-Ch-Ua-Mobile: ?0
10 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
   ↪ (KHTML, like Gecko) Chrome/107.0.5304.107 Safari/537.36
11 Sec-Ch-Ua-Platform: "Linux"
12 Origin: https://www.ru.nl
13 Sec-Fetch-Site: same-origin
14 Sec-Fetch-Mode: cors
15 Sec-Fetch-Dest: empty
16 Referer: https://www.ru.nl/
17 Accept-Encoding: gzip, deflate
18 Accept-Language: en-US,en;q=0.9
19 Connection: close
20
21 AppId=10000009&MwtId=16703&IpxId=0&VarId=3297&ItmId=670677
```

A.1.2 Response

```
1 HTTP/1.1 200 OK
2 Cache-Control: public
3 Content-Type: application/json
4 Vary: Host
5 Server:
6 P3P: CP="NOI DSP MON CUR ADM DEV TAI OUR NOR STA"
7 X-Frame-Options: SAMEORIGIN
8 Permissions-Policy: fullscreen=self, geolocation=self
9 Access-Control-Allow-Origin: https://radboudgezichten.ru.nl
10 X-Content-Type-Options: nosniff
11 Referrer-Policy: strict-origin
12 Date: Sun, 11 Dec 2022 13:22:54 GMT
13 Content-Length: 364
14 X-Cache: MISS from roesje02.uci.ru.nl
15 X-Cache-Lookup: MISS from roesje02.uci.ru.nl:80
16 Connection: close
17 Strict-Transport-Security: max-age=63072000; includeSubDomains; preload
18
19 {"token":"","session_csrf_id":"3db520a3ae2d48a1b36917c938277538",
20 "startEnvIdt":"2","root":"https://cms.ru.nl/","elements":[{"id":"Ipx_0",
21 "screens":[{"alias":"login","size":"Small","args":{"AppIdt":"00000000",
22 "exitAppIdt":"00000000"}}]}],"labels":{"console":"Redactie","stick":
23 "Pennetjes altijd tonen","unstick":"Pennetjes tonen bij zweven","login":
24 "Inloggen CMS"}}}
```
