UCLouvain

École polytechnique de Louvain

Let's analyze malware with symbolic execution: a practical study

Author: **Serena Lucca**
Supervisor: **Axel Legay**
Readers: **Charles-Henry Bertand Van Ouytsel, Charles Pecheur, Ramin Sadre**
Academic year 2021–2022
Master [120] in Computer Science and Engineering

Abstract

Malware are constantly on the rise and their capabilities to avoid analysis are getting better each day.
It is necessary to develop new tools and techniques in order to be able to detect and classify the large amount of malware that appear every day.
In this context, the SEMA Toolchain was created to apply symbolic execution to malware samples in order to create a system call dependency graph that can be used as a signature to classify them.
The goal of this work is to use the SEMA Toolchain to perform an in-depth analysis of some RAT samples. RATs are a a type of malware that present a variety of malicious features. It is controlled by a command and control server which sends commands to ask for a specific feature to be executed. We will explain in detail the process of applying this type of analysis to two RAT samples.
We will also apply symbolic execution to another sample to demonstrate the effectiveness of this type of analysis against anti dynamic analysis technique.
Finally, we present our thoughts and suggestions about the automation of this kind of analysis to be able to process a larger number of samples.

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Axel Legay for providing such an interesting topic and for his encouragement throughout the year. I also would like to express my deepest gratitude to Charles-Henry Bertrand Van Ouytsel and Christophe Crochet for their consistent support all year long, as well as for their feedback and advice.
Furthermore, I would like to thank the readers and member of the jury, Charles-Henry Bertrand Van Ouytsel, Prof. Charles Pecheur and Prof. Ramin Sadre for giving the time to review this work.
On a more personal note, I would like to say that I am very grateful for my friends and family and in particular Vladimir, Charlotte and Milo for their presence and patience in the past few months.
Finally, I must thank my computer for sticking with me through the good and the bad runs. He never let me down and I am grateful for that.

Contents

Chapter 1

Introduction

Each year, Malicious software (malware) grow in number and in complexity [11] [21]. Indeed, in the 80s, malware were usually simple programs used for pranks or personal purposes with not much ambition to widely harm computers. Nowadays, malware represent complex programs with many different features for all type of malicious use. On one hand of the spectrum we have very sophisticated programs which exploit zero-day vulnerabilities and try to find new techniques in order to avoid detection. These are often written by organized and well funded development teams and tend to have a particular goal in mind such as infecting military or governmental systems. On the other hand, we have the malware as a service. The products can have different features such as keylogging, encryption of files, DDoS capabilities, etc, and are packaged in a nice little user friendly interface. This way, it can be sold to anybody who wishes to become a cybercriminal.
Moreover, malware have a much wider surface of attack then before. The world we live in is super connected. It is full of data to steal and appliances to infect. These days, anything with a microprocessor is a potential victim for malware.

Fortunately, the interest in cybersecurity from industries and academics is growing as well [23]. More and more malware analysis techniques arise in order to be able to keep detecting and classifying malware despite their efforts to go unnoticed.
The two best known malware analysis techniques are signature based detection using static analysis and dynamic binary instrumentation using dynamic analysis. The first technique suffers from obfuscation which consists in modifying the syntactic properties of a malware without modifying its behaviour. The second technique can be undermined by malware able to detect an analysis environment. If they realize that they are running in the environment of a malware analyst, they will not display their malicious behaviour.

In this work we are going to be using another type of analysis which is called symbolic execution. This technique is not subject to the limitations of the two previously mentioned techniques but it comes with its own set of problems.
Symbolic execution is essentially trying to give a result similar to a dynamic execution but without actually executing the malware. Instead, it uses static analysis to apply the effects of each instruction to a state. The state is a representation of the system at a certain point of the execution. The execution starts with an entry state and each step creates a new state which is modified according to the instruction that is executed. When a fork occurs in the execution, two states are created and the execution continues in the two separate paths. This way, a tree of all the possible paths is visited during the symbolic execution.
Since we are not actually executing the malware, we must make a modelization of the environment. The environment of a real execution is quite complex so we must make some abstractions to make it usable by the symbolic execution. This can make the symbolic execution incoherent or not very precise. Moreover, symbolic execution suffers from the problem of path explosion. Indeed, the number of paths grows exponentially with the number of forks. This problem can make it very difficult to analyze some larger samples. These are the main problems that we will have to address during our work.

We are going to use the SEMA Toolchain[8], which is using the angr framework [24], in order to apply symbolic execution to some samples. The SEMA Toolchain outputs a System Call Dependency Graph (SCDG) which can then be used to classify the malware. It also outputs a json file which contains all the API calls and their arguments. The API calls give a lot of information on the behaviour of a program. It is then possible to learn a lot on the functioning of a malware by analyzing this second output.

Two of the malware that we will analyze are RATs. This type of malware is particularly interesting. Most malware base their whole personality off of a single feature, such as ransomware, keyloggers, stealers, etc. RATs, on the other hand, can have all of these features, or at least some of them, nicely bundled in a command processing loop. The person in control of the RAT can send commands to execute these features as he wishes.
This kind of behavior induces a quite complex control flow which can give some trouble to the symbolic execution. It also comes with a lot of communication with its command and control server, which can sometimes be tricky to simulate. All of this will give us plenty to work on.

Contributions In this work, we will use the SEMA Toolchain and more particularly its module that performs the symbolic execution and creates the SCDG, in order to analyze three malware samples. We will show that it is possible to use this tool to perform a complete and in depth analysis.
We want to show the challenges that malware bring to this type of analysis and the modifications that have to be applied in order for it to succeed. We will also bring some ideas and suggestions to improve the symbolic execution module. The goal that we have in mind is to provide a complete analysis to a larger amount of samples with the automation of the modifications that had to be made manually for our analysis.

Organization This work is organized in two important chapters, the background in Chapter 2 and the analysis in Chapter 3. In the first part, we will go over many different topics to set up the context. These topics will include the different techniques for malware analysis in Section 2.1, The angr framework in Section 2.2 and the SEMA toolchain in Section 2.5. We will also talk about the behavior of RATs in Section 2.4 and some more general information about malware on the Windows OS in Section 2.3.
In the second part, we will present the analysis of three malware with the use of the SEMA Toolchain. Two of these malware are RATs and the last one is a Ransomware as a Service that we will use to demonstrate the effectiveness of symbolic execution against anti dynamic analysis techniques.
At the end of this work, we will discuss about the contributions that were made in Chapter 4 and we will talk about the prospect for some future work in Chapter 5 before concluding with some final thoughts in Chapter 6.

Chapter 2

Background

In this section, we will present the theoretical knowledge that will be useful for the practical malware analysis in Chapter 3. We will first talk about the different techniques for malware analysis in Section 2.1. Among these, we will find the symbolic execution which is the technique that we will use for our analysis. We will also present angr and Ghidra, tools for malware analysis, in Section 2.2. We will continue with a presentation of the windows API and some typical behaviours of windows malware in Section 2.3. Since our analysis will mostly be done on RATs, we will present them in Section 2.4. Finally, we will conclude with a presentation of the symbolic execution module of the SEMA toolchain (Symbolic Execution for Malware Analysis) [8] in Section 2.5.

2.1 Malware analysis techniques

Different techniques exist to analyze malware [10]. In this section, we will present static analysis, dynamic analysis and finally, symbolic execution.

2.1.1 Static analysis

The concept of static analysis is to analyse a program without executing it, i.e. off-line computation. One way of doing this is to disassemble the binary. This technique consists in transforming the binary code into a low level assembly language. This type of analysis can be used to rapidly gather information about the syntactic properties of a malware in order to deduce its syntactic signature. The syntactic properties represent information about a binary such as the length, the number of section, the checksum, certain strings, etc and the syntactic signature is the sequence of bytes that describes these properties. These signatures can be used to detect and classify a malware by comparing it to the signatures of other malware.

Unfortunately, it is very easy to modify the syntactic properties of a malware while still having the same behavior. This practice is called obfuscation and it is one of the great limitations of static analysis.
Static analysis can also be used to analyze more in depth the behavior of a malware although the process is very long and tedious. With the use of tools such as Ghidra or IDA, it is possible to not only disassemble but also decompile our binary. Decompiling means transforming the binary code into a high level language that can easily be read by humans. This version is usually more readable than the assembly version. Once the code is decompiled, the analysis consists in looking at the functions trying to understand what they do and dive into the code following the function calls. Some binaries are very straightforward and easy to analyse statically but most malware are heavily obfuscated which makes it very hard to understand the behavior of the malware just by looking at the decompiled code.
In later sections, we will use this kind of static analysis to complement the symbolic execution while trying to understand the behavior of different malware. Static analysis will allow us to find the addresses needed for some special cases and help us have an idea of what piece of code exactly is being "executed" at a certain address when we will encounter difficulties in the execution.

2.1.2 Dynamic analysis

Dynamic analysis consists in executing the malware and observing what happens. Usually this execution is done in a sandbox because we don't want the malware to spread or to have actual consequences on a real system. The sandbox allows us to isolate the malware and to easily recover from it afterwards.
In the same way that static analysis can create syntactic signatures, dynamic analysis can create behavioral signatures. The dynamic analysis will define the behavioral properties of the malware by analysing the traffic on the network and by observing the interactions with the memory.
The behavioral signature is usually more precise than the syntactic signature and it is not affected by code obfuscation.
The downsides of this type of analysis is that it requires a lot of resources to emulate the system in which we run the malware. It is also too dangerous to allow the malware to have access to the internet but quite complicated to emulate logical interactions with the internet.
Moreover, dynamic analysis will follow only one execution path. This can be a problem because some malware use techniques to avoid dynamic analysis. These techniques consist in performing some checks on the infected host to try to determine if they are running in a sandbox or in the environment of a malware analyst. If it is the case, the malware will not display its malicious behaviour. Since

no other execution path can be explored, the malware will not be detected. The subject of dynamic analysis evasion will be addressed in more details in Section 2.3.4.

2.1.3 Symbolic execution

Symbolic execution uses symbolic input values and symbolic variables which allows the execution to visit a tree of all the paths that can possibly be reached. This technique makes it possible to analyze malware that hide their malicious behaviour when they realize that they are running in a sandbox environment. Indeed, with concrete execution, we explore only one path and if that path fail then we have nothing. With symbolic execution we will explore all the possible paths so we can be assured that we will find the path that shows the malicious behaviour of the malware.
Unfortunately, this analysis technique suffers from the problem of path explosion. This problem comes from the fact that when the execution finds a branch that depends on a symbolic variable, the execution will fork and both paths created by the branch will be visited. This prevents the symbolic execution to scale up to large binaries with many forks because the number of paths that have to be explored becomes way too large.
We can counter this side-effect of symbolic execution by using some domain-specific optimizations and search heuristics such as avoiding error paths, setting a limit to the number of iterations in a loop, defining which states should be explored first, etc.
The symbolic execution is the technique that we will use the most to analyze malware in Chapter 3. We will explain more concretely what can be done to avoid the problem of path explosion. We will be using angr to perform the symbolic execution, this tool is presented in the next section.

2.2 Malware analysis tools

In this section we will talk about Ghidra [2] which can be used for static code analysis and angr [24] which performs symbolic execution.

2.2.1 Ghidra

Ghidra[2] is an open-source framework created by the NSA that was released to the public in 2019. This tool was created for software reverse engineering. Ghidra provides disassembly and decompiling for a large variety of architectures.
When a binary is loaded, the tool provides different windows to visualize it. We

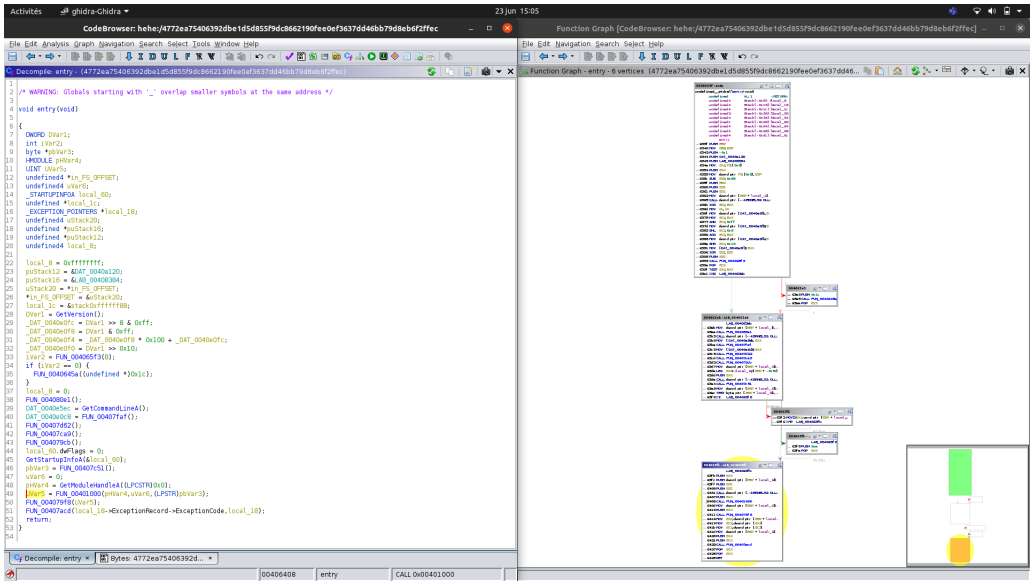

Figure 2.1: Decompiled view and function graph view in Ghidra

have the binary view which shows us the raw bytes in the different sections, the listing view which gives the bytes interpreted into the values that they should take (instruction in assembly, strings, etc), the function graph view which also contains the disassembled code inside the blocks of the graph (right panel in Figure 2.1) and the decompiled view which gives us the decompiled code in a high-level language (left panel in Figure 2.1).
This last view is the easiest to read and to understand but the view with the disassembled code is quite interesting as well because it is not as much interpreted as for the decompiled view so we can see exactly the addresses and the instructions that are being executed.
Another interesting feature is the symbol table with the symbol references where we can see the functions and API calls and the addresses where the calls happen.
As mentioned earlier, static analysis can hardly be used to perform deep and complete analysis on malware because most of them are heavily obfuscated. We might lose ourselves in the execution flow because of the many functions that call many other functions and so on. Static code analysis is also limited by all of the encrypted data such as the name of API calls or even some pieces of code that will be injected in some other process.
We will use Ghidra to analyze some restricted portions of code in order to help the symbolic execution, for example to replace the loops or to reduce the number of possible paths.

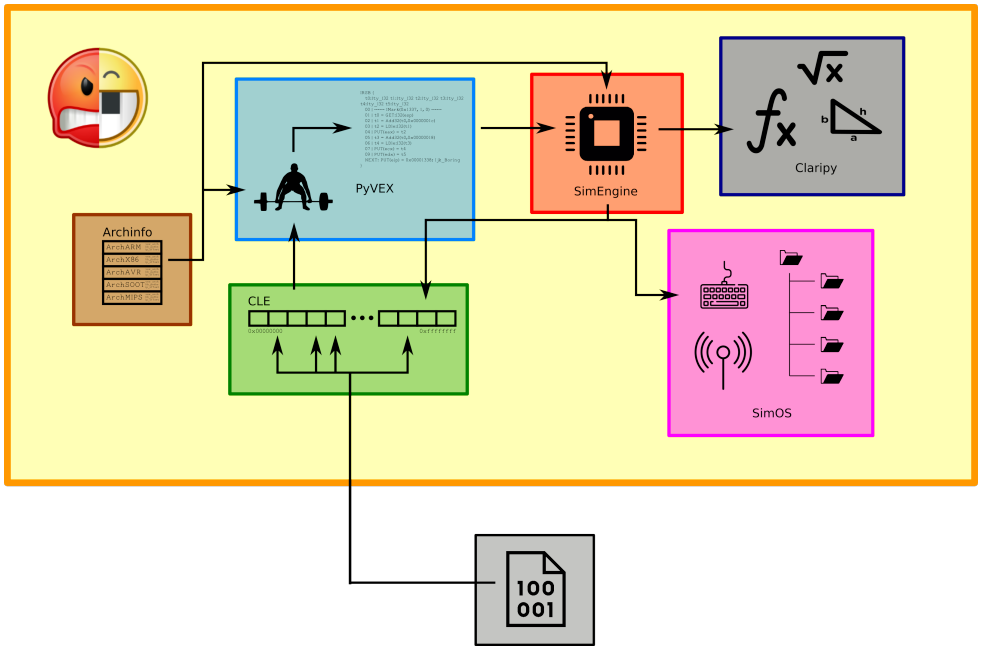

Figure 2.2: Components of the angr framework

2.2.2 angr

angr [24] is an open-source framework that performs symbolic execution on binaries. It is conveniently written in python and has a large community on github.
angr is made of 6 big components that work together to apply symbolic execution on binaries. We can see their organization in Figure 2.2 that is from an article [25] on the angr blog about the internals of angr.

CLE is the name of the loader that is responsible to load the binary. In the case of this production, the loader has to load portable executable (PE) files. The loader will dissect the header to find information about the architecture, the sections, the libraries that must be loaded and other things about the PE. It will then create a representation of the memory of the program and load the libraries that are required.

Archinfo uses the name of the architecture that was found by the loader and uses it to return an object that contains the register file, the endian-ness and other information about the architecture.

PyVEX will translate the machine code found by the loader into an intermediate representation called VEX. This way, the engine that will symbolically execute the code only has to understand this representation of the code instead of all the different machine code that exists.

SimEngine is the execution engine that will interpret the VEX representation. The engine will apply the effects of the execution to some states. These states contain the memory, the registers and such things that define a certain moment of the execution on a certain path. At each step, the execution engine will take a state, apply a block worth of code on it and output a list of successors that are corresponds to the state that can be reached from the previous state after the execution of the block of code in question.

Claripy makes the link with the SMT-solver that will do all the work on the symbolic values. Claripy will create and resolve the symbolic values in the execution as well as performing calculation on them when needed.

SimOS gives us the higher level abstractions that are usually given by the OS. These include the file system, the network but most importantly, the symbolic summaries of syscalls also called SimProcedures about which we will talk more in depth in Section 2.5.

2.3 Windows malware

In this section we will discover the basics of windows malware i.e. the most common functions and behaviours that we can expect to see during our analysis in Chapter 3.

2.3.1 Windows API

The Windows API, better known as Win32, provides a set of functions that can be used to write applications for the Windows OS. These functions are exported in dynamic link libraries (dll) files. The portable executable (PE) files that use such functions have an import section that lists the dll that should be loaded and the functions that we will use from these dll.
It is also possible to load other libraries dynamically at run time. Malware typically like this technique because they can keep the name of the libraries and functions that they need encrypted in memory and then decrypt and load them at run time. This way we cannot find any information about the API calls that the malware will make by analyzing it statically.

Two functions are needed to perform this. First of all the `LoadLibraryA()` function is called and the name of the dll to be loaded is given in argument. This function returns a handle to the loaded library that can be given as argument to the `GetProcAddress()` function with the name of the function that we want to extract from the library. The `GetProcAddress()` function returns the address of the function exported by the dll.

2.3.2 Processes and Threads

A process is an instance of a program that is executing on a machine. Processes are created and terminated by the OS with functions such as `CreateProcess()` and `TerminateProcess()`. A process can create another process. In that case, the created process is called a child process and the process that created it is called the parent process.
Processes are managed thanks to different structures including the process environment block (PEB) which is located in the process address space, the process control block (PCB or `KPROCESS`) and the `EPROCESS` structure which are in the system address space [15].
Each process has its own virtual address space. A process can read and write to its own virtual address space or the virtual address space of another process with the functions `ReadProcessMemory()` and `WriteProcessMemory()`. It can also allocate memory to its own virtual address space with the function `VirtualAlloc()` and to the address space of another process with `VirtualAllocEx()`.

A thread is a part of a process that can be scheduled by the CPU for execution. A process can have one or more threads and they all share the virtual address space of the process. The threads also have some stuctures to represent them. There is the thread environment block (TEB) that is found in the process address space then there are the `KTHREAD` and the `ETHREAD` structures that are located in the system address space [15].
A process can create a thread for itself with the function `CreateThread()`. It can also create a thread in another process with the function `CreateRemoteThread()`. A thread can be suspended, resumed and terminated with the corresponding functions `SuspendThread()`, `ResumeThread()` and `TerminateThread()`.
Threads can also store some local value that don't have to be shared with the other threads of the process thanks to the thread local storage (TLS).

2.3.3 Process injections

Process injection consists in injecting malicious code in the address space of another process. This technique can be used to avoid detection because the malicious code will be running in a legitimate process. We can also use process injection for privilege escalation by injecting our code into a process that has higher-level privileges.
There are several ways to perform process injection as we can see on MITRE ATTACK [5] but we will only present the two ways that we will see later during our analysis in Chapter 3. Most of the information for this part is found in a tutorial about abusing windows internals on the TryHackMe website [30]. We will borrow their convenient little graphs that nicely summarize the windows API calls needed for process injection.

Shellcode injection Shellcode injection is one of the simplest kind of process injection. This injection is done in 4 steps. We can see these steps in Figure 2.3 with their API calls and their interactions with the memory.
The first step is to open an existing process with the function `OpenProcess()`. This function requires the id of the process to be opened. We can find it with the API calls `CreateToolhelp32Snapshot()`, `Process32First()` and `Process32Next()`. The first function returns a snapshot of all the processes on the host then the two other functions are used to iterate over the snapshot to return several `PROCESSENTRY32` structures that contain information about the processes. Another way to find the process id is to call the function `NtQuerySystemInformation()` with the parameter SystemInformationClass set to 0x5 which corresponds to `SystemProcessInformation`. In this case the function will return an array of `SYSTEM_PROCESS_INFORMATION` structures that describe each process running on the host. Now the malware can iterate over this array to find the id of the desired process. Malware will usually inject the code in the process `explorer.exe` because it is a trusted process that is present on all the windows hosts.
The second step is to make some space in the target process memory to be able to write our shellcode in there. The function that is used here is `VirtualAllocEx()`. This function is able to allocate memory in the virtual address space of any process whose handle is given in argument.
Now is the time to inject our shellcode (or whatever code) in the memory of the other process. This is done with the use of the function `WriteProcessMemory()` which will use the region of memory that was just allocated in the previous step.
Finally, we call the function `CreateRemoteThread()` that will create a thread in the process whose handle is given in argument. This function also takes the address at which this thread should start to execute. In this case the address is the one

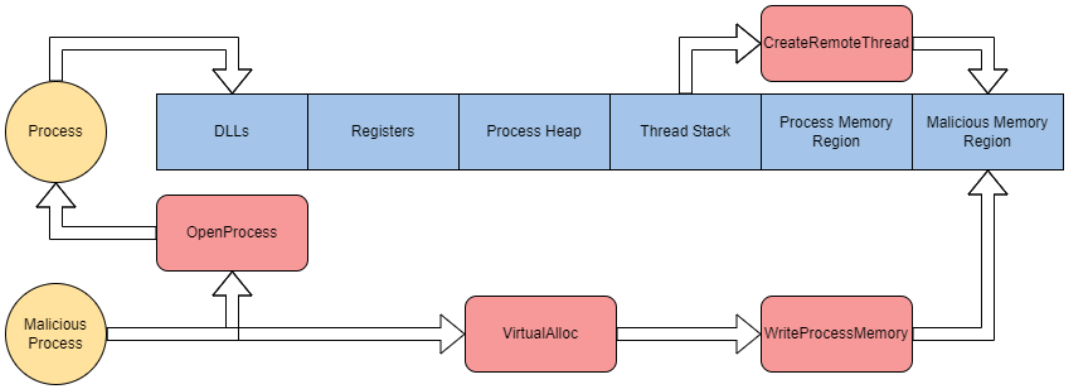

Figure 2.3: Windows API calls for shellcode injection

returned by `VirtualALlocEx()` in the second step because this is the address of the beginning of our shellcode. We can also give an argument to this new thread by putting it in the lpParameter argument of the `CreateRemoteThread()` function.

Process hollowing This kind of process injection is much more complex then the previous one. Process hollowing allows us to inject an entire PE file into another process. The graphical representation of the different API calls for this technique can be found in Figure 2.4.
The first thing to do is to extract the malicious image that will be injected. Most of the time this image is contained inside of the file of the malware. The malware will call `GetModuleFileName()` to find the name of the file in which it is contained. Then it will open it with the `CreateFileA()` function with the dwCreationDisposition parameter set to 0x3 which stands for `OPEN_EXISTING` so that the file is opened only if it already exists. After that, a call to `GetFileSize()` is made to know which size should be passed to the `VirtualAlloc()` function that will allocate the space needed to copy the whole content of the file in memory with the function `ReadFile()`.
At this point, the sections of the PE file that will be injected are probably stored encrypted somewhere inside the big buffer that contains the content of the whole file. This is the time to decrypt the code and data that will be injected and put them in a smaller buffer just for them.
The next step is to prepare the process to be injected. First of all we create a process in a suspended state with the function `CreateProcessA()`. Then, we call the function `GetThreadContext()` that will give us a `CONTEXT` structure containing, among other things, the CPU registers for this process. We need this because the pointer of the base address of the process is equal to the address contained in

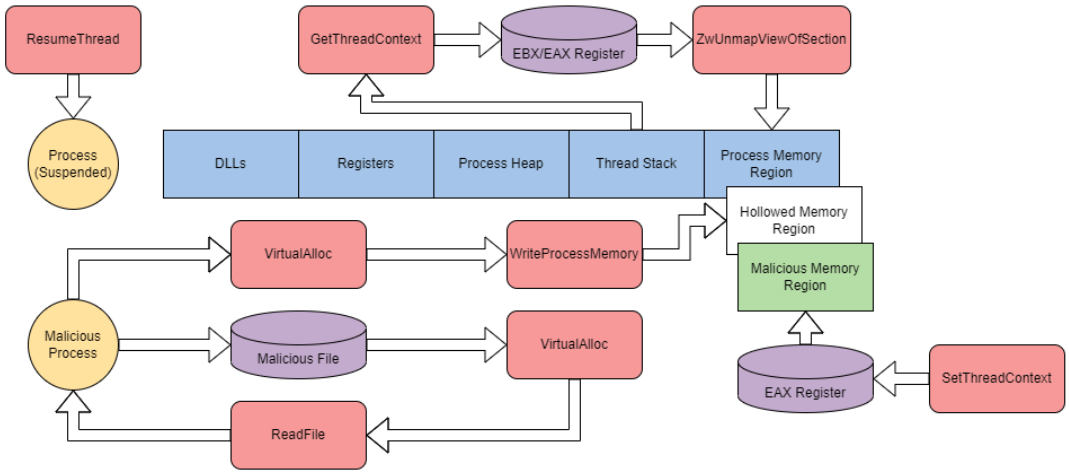

Figure 2.4: Windows API calls for process hollowing

`ebx` + 8. We can now find this base address by reading in the address space of the process with the function `ReadProcessMemory()`. Finally we can unmap the memory of the process by giving the handle to the process with the base address that we just found to the function `ZwUnmapViewOfsection()` or equally to the function `NtUnmapViewOfsection()` which essentially does the same thing but has a different name.
Now that the hard stuff is done we basically follow the steps of the basic shellcode injection. We will allocate exactly the size of the malicious image that we decrypted in the memory of the new process by using `VirtualAllocEx()` then we will make several calls to `WriteProcessMemory()` to write the PE header then all the sections of the PE one by one. We can find the sizes and addresses necessary for all these calls by looking into our malicious image at some specific offsets.
Finally the injection is done, now we have to update the entry point in the new process. To do so we just have to modify the `eax` register from the `CONTEXT` structure that we got earlier from the call to `GetThreadContext()` then update it by passing this sructure to the `SetThreadContext()` function. The last thing to do is to resume the thread so that our injected file can execute by calling the function `ResumeThread()`.

2.3.4 Dynamic analysis evasion techniques

Malware become more and more capable to avoid dynamic analysis. This is one of the reasons why symbolic execution becomes interesting in the domain of malware analysis. Symbolic analysis allows us to visit all the paths that can possibly be

reached which means that we can definitely find the path that has overcome the techniques that were meant to avoid dynamic analysis.
In this section, we will find a non exhaustive list of dynamic analysis evasion techniques. The dynamic analysis evasion techniques can be separated in two categories. On one hand we have the anti-sandbox techniques which consist in detecting if we are running in a sandbox environment. We can find many information about those on the dedicated MITRE article [6]. On the other hand we have the anti-debug techniques, these techniques are meant to detect if the malware is being debugged. We can find many of these techniques with a very thorough explanation and examples on the anti-debug tricks website [12]. We will also use the information from [1] which surveys and classifies many dynamic analysis evasion techniques.
Finally we will illustrate these techniques in Chapter 3 with the analysis of the Satan RaaS. We chose this malware specifically because of the many evasion techniques it uses. We base this analysis on this article [26] that presents a very complete analysis of the malware.

Debugger evasion

- `IsDebuggerPresent()` checks if the BeingDebugged flag of the Process Environment Block (PEB) is set or not.
- `CheckRemoteDebuggerPresent()` uses the `NtQueryInformationProcess()` function to retrieve the value of the ProcessDebugPort for this process.
- `NtQueryInformationProcess()` retrieves different information about a process depending on the value of the parameter ProcessInformationClass. With this parameter set to ProcessDebugPort (`0x07`), we can find out if the process is being debugged or not (which is exactly what is done for the `CheckRemoteDebuggerPresent()` function). We can also set the parameter to ProcessDebugFlags (`0x1F`) or to ProcessDebugObjectHandle (`0x1E`) to retrieve some value that will also allow the malware to determine if it is being debugged or not.
- `NtQuerySystemInformation()` called with its parameter SystemInformationClass set to SystemKernelDebuggerInformation (`0x23`) will return two flags indicating if a kernel debugger is present or not.
- `GetThreadContext()` can retrieve the registers of the process and among them are the debug registers which contain the addresses of the hardware breakpoints set by a debugger. if these registers contain a value that is different than 0, it means that the process is being debugged.

- `CloseHandle()` or `NtClose()` with an invalid handle generates an exception when the program is debugged.
- `AddVectoredExceptionHandler()` followed by the int 3 (`0xCC`) instruction which is actually a software breakpoint that is usually set by debuggers. In this case, the instruction is a debugger trap because if a debugger is present, it will handle the instruction as a breakpoint instead of giving the control to the VectoredExceptionHandler that was just set.
- `SetUnhandledExceptionFilter()` then create an exception (typically `xor eax, eax` followed by `div eax`). This is also a debugger trap, if the UnhandledExceptionFilter is not called after this exception it means that the program is being debugged
- `OpenProcess()` with the process csrss.exe will succeed only if the program is being debugged.
- `BlockInput()` works only if a debugger is present and when called with the argument true, blocks the mouse and the keyboard which is a interesting way to end a debug session.
- `FindWindow()` can be used to detect if the window of a debugger is present. The tested inputs usually are OLLYDBG, WinDbgFrameClass, Immunity Debugger, Zeta Debugger, Rock Debugger and ObsidianGui.
- `NtSetInformationThread()` can modify the value of ThreadHideFromDebugger in order to hide the thread from the debugger.
- `SuspendThread()` is used to suspend the thread in which the debugger is running when the malware finds that there is one.
- `CreateThread()` creates a new thread in which the malicious code is executed, this new thread is running outside of the debugger.
- `DebugActiveProcess()` attaches a debugger to the current process. Since a process can only have one debugger, the function will fail if there is already a debugger attached to the malware.
- `GetTickCount()`, `GetSystemTime()`, `GetLocalTime()`, `QueryPerformanceCounter()`, `timeGetTime()` are used to evaluate the time that passed during some part of the execution. If a debugger is present then the execution will be much slower than in a normal execution without debugger.

Sandbox evasion

- `GetModuleHandle()` called on specific libraries used for malware analysis. If they can be found on the host then the malware is probably being analyzed. Some libraries are also specific to sandboxes so if they are present it means that the malware is in a sandbox.
- `GetProcAddress()` of the function wine_get_unix_file_name() to see if it is running in a Wine sandbox.
- `GetModuleFileNameA()` to get its own path name and compare it to file names and folder names that are often automatically used in sandboxes.
- `GetUserNameA()` to compare the user name to the user names that are often used in sandboxes.
- Compare all the processes in a system to a list of processes related to malware analysis and sandbox environment. The list of processes is retrieved with `CreateToolhelp32Snapshot()` `Process32First()` and `Process32Next()`.
- `Sleep()` is often used to delay the execution to avoid being detected if it is run automatically for a fixed amount of time in a sandbox. Other techniques exist to stall the execution of a program such as running a long computation or executing many calls to some benign function.
- Some malware await for a trigger to execute their malicious behaviour. This triggers can be a certain date and time, an input from the network, a specific sequence of keystrokes, etc.
- Malware can distinguish between sandbox and real user by probing some user activity such as mouse left-click, cursor position with `GetCursorPos()`, or any input event with `GetLastInputInfo()`.

2.4 RATs

Originally, RAT was the acronym for Remote Administration Tool (or Remote Access Tool), a type of legit software used to manage a server or to repair a computer remotely. Some people started to use this kind of tool for malicious purposes by making them undetectable and by adding some malicious features. And so began the history of the Remote Access Trojans (RATs).
This type of malware allows an attacker to control remotely an infected machine. The attacker communicates with the infected machine with a control and command server (also called C2 or CC). [13]

RATs can be used for almost anything since once an attacker has full control over the computer of a victim, he can do whatever he wants with it. The RATs have some features that will be executed on the demand of the attacker. These features can include keylogging, taking screenshots, stealing files and passwords, recording webcam and microphone but also more advanced functionalities such as process injection, downloading or spreading of malware, etc. [31] [14]

RATs have greatly evolved during their 30 years of life on earth [31], in the recent years they have become a product of the market that is sold with a nice interface on the attacker side and various features for the malware that is distributed to the victims. We will only be analyzing the malware on the victim side. In order to make it easier for us to understand our findings during the analysis, we will take a look at some existing analyses of RATs that are publicly available. This way we can try to find patterns and discover the typical control flow of a RAT as well as making ourselves a clearer idea of the features that are found in different RATs.

NukeSped The first analysis is about the NukeSped RAT and is presented by Fortinet [29]. This family of malware encrypts the API names then it decodes and import the functions dynamically with `GetProcAddress()`. All the strings are encoded and decoded using a xor based encryption. In the main part of the RAT, the received command is decoded then it is passed in a big switch/case statement that will execute a specific function based on the received command. They describe this behaviour as the typical control flow of a RAT.

FALLCHILL The next analysis is also presented by Fortinet and is about the FALLCHILL RAT [28]. The strings are also encrypted and the function loaded dynamically at run time. This RAT creates a thread that launches the payload. In the payload, the RAT decodes the address of the C2 server, to which it then connects. The next part is an infinite loop with a function to receive the commands followed by the very distinguishable switch/case block. This block for handling the command is very much similar to the one of the NukeSped RAT and has similar features.

DarkComet MalwareBytes presents many analysis of RATs. One of them is the analysis of the DarkComet RAT [20]. This RAT is available for free under the title of a legitimate remote administration tool. Some people use it for malicious purpose which makes anti-virus softwares detect it as a malicious file. This RAT has a huge amount of features, going from the classical keylogging, file stealing, webcam recording, etc to some fun features to mess with the victim such as hiding

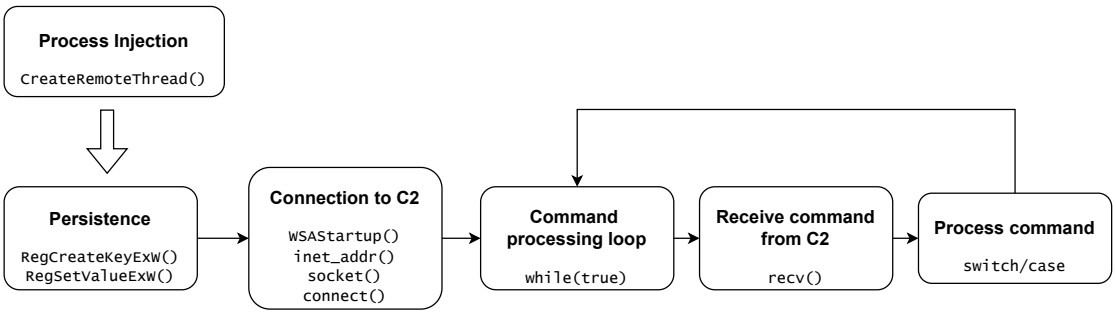

Figure 2.5: Typical control flow of a RAT

the toolbar or messing with the CD tray (quite old malware). It is also possible to perform DDoS attack from the infected computer which is an odd feature for a legit remote administration tool.
Once again, we encounter the typical RAT schema in the functioning of the DarkComet. Once the victim is connected to the C2 server, the RAT will check in every 20 seconds while waiting for new commands. When a command is received, it is decrypted and authenticated then the RAT will execute the function for the command ans send back the result to the C2 server.

There is a clear pattern for the execution flow of a RAT from what we observe in these analysis. First of all we should observe a connection to the C2 server, then a distinctive infinite loop which contains the code that receives and executes the command. The part that executes the command takes the form of a switch/case statement or a block of if (and else) statements. Let's note that this pattern is the "remote access" part of the RAT, before all of this we have the "trojan" part of the RAT which usually consists in some sort of process injection and inside the injected process, a piece of code to make itself persistent.

We can find even more hints of what we might find in our future analysis by looking at some actual code of RATs. We already know that the malware that we will analyze will be windows malware and coded in C++ and that we are particularly interested in the function calls that occur during the execution. We saw earlier the kind of API calls that are used for process injection. For the "remote access" part of the malware, we can take a look at the code of an open-source remote access tool [22] and take notes of the different API calls found in different parts of the software. The summary of the typical execution flow of a RAT with the calls that we might find along the way is represented in Figure 2.5.

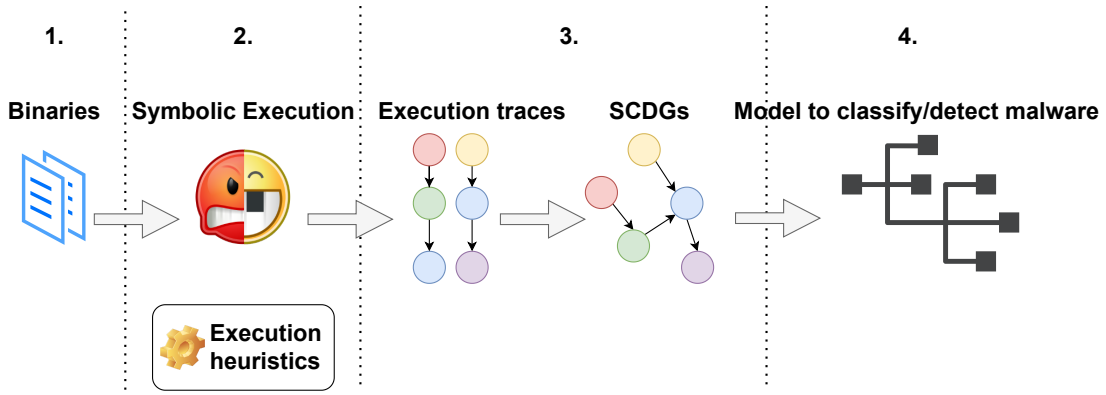

Figure 2.6: Execution flow of the toolchain from [9]

2.5 ToolChainSCDG

The main tool that we will use for the analysis is called the SEMA Toolchain which stands for Symbolic Execution for Malware Analysis [8].
The execution flow of the toolchain is represented in Figure 2.6. First of all, symbolic execution is performed on the binary with the angr framework. Then, from all of the API calls that have been encountered during the symbolic execution, the toolchain outputs a System Call Dependency Graph (SCDG) that can be used to detect or classify the malware.

Binaries The binaries that we will feed to the toolchain are malware used for the windows OS. We can quite easily find those on MalwareBazaar but not all binaries are suited for this type of analysis.
The SCDG is constructed from the system calls that happen during the execution so if a malware was to replace all the system calls by their source code, we wouldn't have much to work on.
We already know that obfuscation is a vast problem for static analysis but it can also be problematic for symbolic execution. Indeed, some obfuscation techniques are specifically used to hinder symbolic execution [32] usually by voluntarily creating some sort of path explosion.

Symbolic execution with angr At the heart of the SCDG toolchain is the angr script that performs the symbolic analysis on the binaries. The script loads the binary and creates an entry states at a certain address (usually main but can be modified).

It also loads the libraries and functions required during the execution by hooking the address of the function to a stub that summarizes the behavior of said function. These stub functions are called SimProcedures and are essential to control the path explosion problem. Ideally all of the API calls that are executed during the execution should be implemented as a SimProcedure. Some functions do not require much, simply returning 0 or 1 is sufficient but some other functions have much more complex behavior and require complex SimProcedures to make sure that the control flow follows the way that it is supposed to. If nothing is done, angr will automatically return a symbolic value and do nothing else inside the function or with the parameters. This can often cause massive path explosions.
The main job of the script, that is the execution, is performed by applying some exploration techniques starting from the initial state in order to visit all the possible paths. The exploration techniques that are implemented in the ToolChain are BFS, DFS, CBFS and CDFS. The first two are the very well known breadth first search and depth first search, and the other two, where the "C" stands for custom, give higher priority to states leading to unexplored address space in order to promote higher code coverage. We will mostly use DFS and CDFS because we usually don't need to see all of the possible traces. We can often see the behavior of a malware with just a few traces. Moreover, BFS consumes a lot more memory than DFS which can sometimes crash the program if we execute it for too long.
The other functionalities of angr that we will use for our analysis are the breakpoints and the user hooks. The breakpoints allow us to stop at some desired address and from there, we can read from and write to the memory, the registerS, the stack, etc. The addresses at which we will put breakpoints will usually be found with static analysis on Ghidra.
User hooks are quite like SimProcedures but extended to any address. Basically, we specify an address of the code section, a number of bytes that we want to skip and a function that should execute instead of the specified block. And there you have it, a user hook.

SCDG Once the execution is over, the toolchain outputs a System Call Dependency Graph (SCDG). This graph puts the function calls as nodes and creates vertices between them for every argument that they share as we can see in Figure 2.7.
We can use this type of graph to classify malware [9]. The idea is to apply the toolchain on a large number of malware from a specific family and create a signature for this family from the biggest common graph between all of the SCDGs. Then, repeat the process for different malware families. When we want to classify a malware, we compute its SCDG and compare it to the signatures of each family.

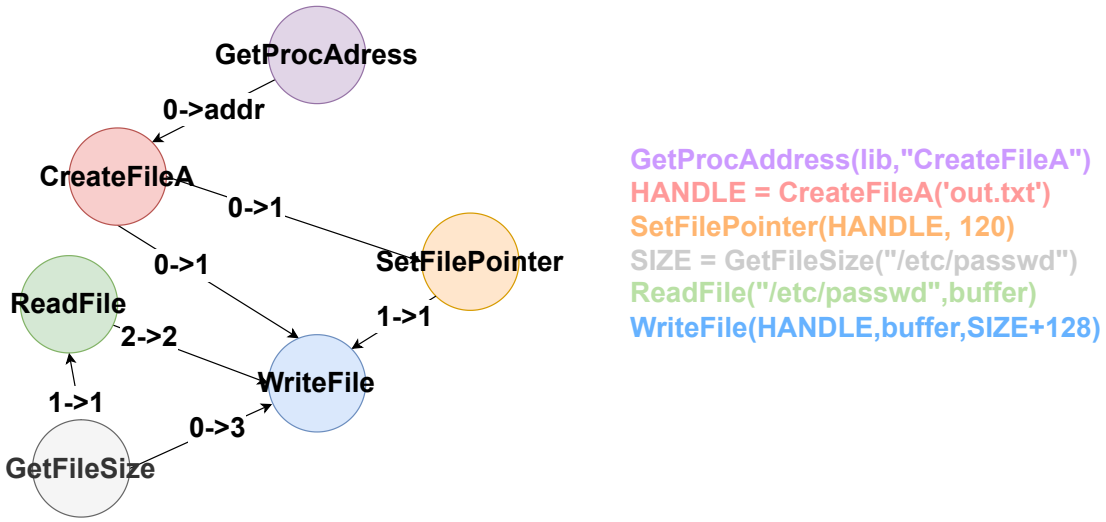

Figure 2.7: Example of a SCDG from [9]

The main limitation of the toolchain comes from the symbolic execution which is not always consistent. One of the causes of this inconsistency is the API calls. Indeed, when the SimProcedure of an API call is not properly implemented, it returns a symbolic unconstrained value but does not apply any effect of the call to the system.
When the program times out, all the API calls that were made so far are printed out. We will try to implement correctly all of the SimProcedures to try and make the execution more consistent by actually applying the effects of the calls on the system.
This will also help with the path explosion problem because we will be restricting the return values of the calls which will create fewer different paths in the case of a branch point that depends on the return value of such call.
The main work during the analysis will consist in implementing these SimProcedures but also defining heuristics to reduce the path explosion and to make the overall execution more consistent. The more complete environment we can give to the malware, the more consistent the execution will be. Hence we will also have to deal with the files that the malware needs, find a way to allow process injection, etc.

Chapter 3

Practical analysis

3.1 Intial paper reproduction

3.1.1 Description

The first malware that we will analyze is a RAT of the Enfal family that can be found on VirusTotal. Its md5 signature is 7296d00d1ecfd150b7811bdb010f3e58.
We chose this particular malware because it is the subject of a case study paper that gives an example of symbolic execution for malware analysis [7]. Our goal is to replicate this analysis but with the use of the ToolChain.
This paper gives us a lot of information for our first analysis. They present different features that can be implemented to speed up or guide the execution. They also depict in great details the functioning of the RAT and most particularly its command processing loop. This way we know what we are looking for during our analysis and we have some hints about the problems that we might encounter.
We should expect to see a thread injection in the first part of the malware. The thread is injected in the process `explorer.exe` with the function `createRemoteThread()`. It will receive a large buffer as argument containing all the information needed for it to work properly.
In the payload we should first see the malware resolve the addresses of all the dlls and functions that will be needed for its execution. Then the malware should create a mutex to have only one instance running. Finally, we will see the command processing loop where at each iteration, the malware will receive information from its C&C server, execute some command then send back information to the server.

3.1.2 First part of the malware

The execution of the first part of the malware is quite straightforward. The control flow is not too complex so we do not have to guide the execution in some direction.

In order to make the execution more accurate, we have to implement the SimProcedures for the functions that are called during the execution.
Some calls do not influence greatly the execution. For example, the ones that perform a simple check or those that free a resource. For those calls, we simply discard the error return value and link a symbolic buffer to the arguments that must contain an output.
Some calls are much more complex and important for the execution. We mean by that, the calls that modify the memory or the state of the program as well as the calls that output information that the malware needs to continue the execution.For these types of call, we must be very careful about their implementation as they have a greater impact on the execution.
Once the execution terminates (usually times out), the tool outputs two files. One of them is the SCDG graph and the other is a json file which contains a summary of the API calls that happened during the symbolic execution. As a matter of fact, the API calls that are found in an execution trace gives us a lot of information on the behaviour of the malware.
For this execution, we can see that the malware starts by collecting some information on the machine (`GetVersion()`, `GetStartupInfoA()`, GetACP(), etc). After that, it calls the function 0x401000 whose control flow graph is represented in Figure 3.1.
In this function, the malware performs a privilege escalation by abusing the token `SeDebugPrivilege`, (1) in Figure 3.1. This will allow him to perform the process injection without being noticed. The malware makes itself persistent by modifying the registry key `SoftWare\\Microsoft\\Windows\\CurrentVersion\\Run` with `RegSetValueExA()`, (2) in Figure 3.1.

3.1.3 Process injection

At this point, the malware carries on with the process injection. We can observe the typical API calls such as `OpenProcess()`, `VirtualAllocEx()` and `WriteProcesMemory()`, (3) and (4) in Figure 3.1. The pieces of code that are injected can be found in plain in the code of the malware but the name of the external functions are not resolved.
The next thing that we see is a large number of calls to `lstrcatA()` and `lstrcpyA()`. The strings passed in argument represent library names, functions names and strings used to communicate with http. This part is in fact the construction of the buffer that will be passed in argument to the injected thread to give it some context for its execution, (5) in Figure 3.1.
Finally, we see the call to the `CreateRemoteThread()` function, (6) in Figure 3.1.

There are three threads created this way but only the first one gives us interesting information on the behaviour of the RAT so we will focus our efforts on this thread.

3.1.4 Threads and processes in angr

The question of the symbolic execution of concurrent threads and concurrent processes in angr currently is an open problem in academia [4].
Indeed, concurrency would greatly amplify the path explosion problem because we have to consider different possible interactions between the different threads/processes. Moreover, it is still unclear how to represent concurrent actions.
As of now, we can only perform a symbolic execution on a program running in a single thread in a single process.
This is the reason why we have to inject the malicious thread in the current process and not in the remote process in which it should normally be injected. In our case, it won't change anything to the execution of the payload because it does not interact with the other threads of the process, that is, it does not make any difference in which process it is injected.
The function `VirtualAllocEx()`, which allocates memory in another process, will have the same implementation as `VirtualAlloc()`, which allocates memory in the current process. Lucky for us, this one is already implemented by angr so we can just reuse it.
In the same way, we will implement the function `CreateRemoteThread()` as if it was a simple call to `CreateThread()`. This SimProcedure was already implemented for the analysis that was presented in the paper mentioned above. We can find it with many other SimProcedures on the github [16] of one of the authors.

3.1.5 Dropped payload

Right after the call to `CreateRemoteThread()` , we observe several calls to `LoadLibraryA()` and `GetProcAddress()` to construct the table that contains the address of each windows function needed by the thread, (1) in Figure 3.2. As expected, we then have a call to `CreateMutexA()`, (2) in Figure 3.2, and finally the command processing loops begin.
Each iteration consists in polling the server to determine the command that needs to be executed, (3) in Figure 3.2, sending an acknowledgment to the server, (5) in Figure 3.2, executing the command, red rectangle in Figure 3.2, and finally sending back data to the server, (6) in Figure 3.2.
When the RAT wants to read commands from the server, we have the following sequence of calls `InternetOpenA, InternetOpenUrlA, InternetReadFile, InternetClosehandle`. On the other hand, when data needs to be sent to

the server, we observe the following calls `InternetOpenA, InternetConnectA, HttpOpenRequestA, HttpSendRequestA, InternetReadFile, HttpEndRequestA, InternetClosehandle`.
At each iteration we also have a call to the function 0x4048a0. This functions checks for the presence and the size of the file `sys32time.ini` and the presence of the file `ipop.dll` then it sends the result and some additional information on the host to the CC server, (4) in Figure 3.2.

3.1.6 Infinite command processing loop

If we run the symbolic execution with the DFS explorer, which is much better than BFS for memory consumption, we quickly realize that the infinite command processing loop is going to be problematic. Indeed, at each iteration, the explorer makes the exact same command choice and the whole command set is never visited. This is a result of the exploration technique choice. DFS is trying to follow a path as deeply as possible and the infinite loop will allow the exploration to keep on going for ever. Each time that the execution enters the loop, it finds a branch which can lead it to the command one or to the rest of the commands. And each time, it makes the exact same choice to follow the path of the command one.
We learn from the paper that the first two loop iterations must execute the command 0x1 (ECHO). At the third iteration, any command of the RAT is reachable. We could simply use the concrete loop counter, which is a limit to the number of iterations a state can do in a concrete loop. We could set the counter to three but then this would be problematic for all the other useful loops that need more than three iterations.
We have to restrict the number of iterations to three but specifically for command processing loop. In practice, we statically retrieve the address of the beginning of the loop, increment a counter in the state each time that it visits that address, and discard any state whose counter exceeds three.
This time, if we let the program run for long enough, all the commands are eventually executed.
We could try to automatize this process by detecting the command processing loop then using an iterative deepening search on that loop.

3.1.7 Command set

The whole command set can be found in the red rectangle in Figure 3.2 The first two commands cannot be seen during the symbolic execution because they do not make any API call. Indeed, command command `0x1` and `0x40` (ECHO and PING) simply send a response to the server but do not make any other call.

Following, is a list of all the commands that we can see during the symbolic execution.

- `Command 0x2` use the function `CreateFileA()` to check for the existence of the file `ipop.dll`.
- `Command 0x3` sends a file to the CC server. The name of the file is in the command.
- `Command 0x4` creates a file and fill it with data incoming from the CC server.
- `Command 0x5` executes one command on the host with `WinExec()`.
- `Command 0x6` deletes the file whose name was received in the command.
- `Command 0x7` moves a file from one place to another. The locations are received in the command.
- `Command 0x9` list all the files in a directory. The name of the directory is in the command.
- `Command 0xA` executes multiple commands on the host with `WinExec()`.
- `Command 0xB` creates the directory whose name is received in the command.
- `Command 0xC` executes the file `C:\Windows\System32\netbn.exe` with `WinExec()`
- `Command 0xD` removes the directory whose name is received in the command.
- `Command 0xE` terminates the process.
- `Command 0xF` executes the file `C:\Windows\System32\netdc.exe` with `WinExec()`
- `Command 0x10` executes the file `C:\Windows\System32\NFal.exe` with `WinExec()`

The malware requires many files from the `C:\Windows\System32` directory. These files, `netbn.exe`, `netdc.exe` and `NFal.exe`, are not present on an uninfected host. They may have been received with the command `0x4`.

3.1.8 Symbolic variables in SimProcedures

The SimProcedures are very important to have a consistent execution, the values returned or modified by the calls are very often used later in the execution so if the call is not implemented, it often causes problems of consistency in the output of the symbolic execution.
In order to speed up the execution and to avoid paths that are repeated for a minor change, we can add more constraints to the symbolic variables modified by the SimProcedures.
The best example is for the function `InternetReadFile()`. This function fills a buffer with data from a specified url. The buffer can hold up to `0x1000` bytes but in practice it does not use more than the first few bytes. We can then restrict the size of the buffer to a few bytes.
There has to be some sort of compromise in every SimProcedure between the arguments/return value that have to stay symbolic and those that can take a concrete value for the sake of simplifying the symbolic execution. This compromise depends on the malware that is analyzed and the way it uses each function call.
When a function outputs a symbolic value, it is likely that this value will create a fork at some point. We must decide if the different paths created by that eventual fork are significantly different. If not, we may concretize this value in order to limit the number of path created by the call.
In the future, we could try to see if it is possible to automatize the process of concretizing some values from the API calls, depending on the use that the malware make of those values.

3.1.9 Conclusion

The point of our analysis is not to produce a complete report of the execution flow of the malware, as it was done in the paper, but rather to produce a consistent SCDG which is the result of a consistent symbolic execution. This SCDG can then be used as a signature for the malware in order to classify it.
If we want more details on the execution flow of the malware, we could use tools such as SymNav[3]. In that case, we could not only have a visual representation of the ongoing symbolic execution, but also manually decide which paths should be discarded or not.

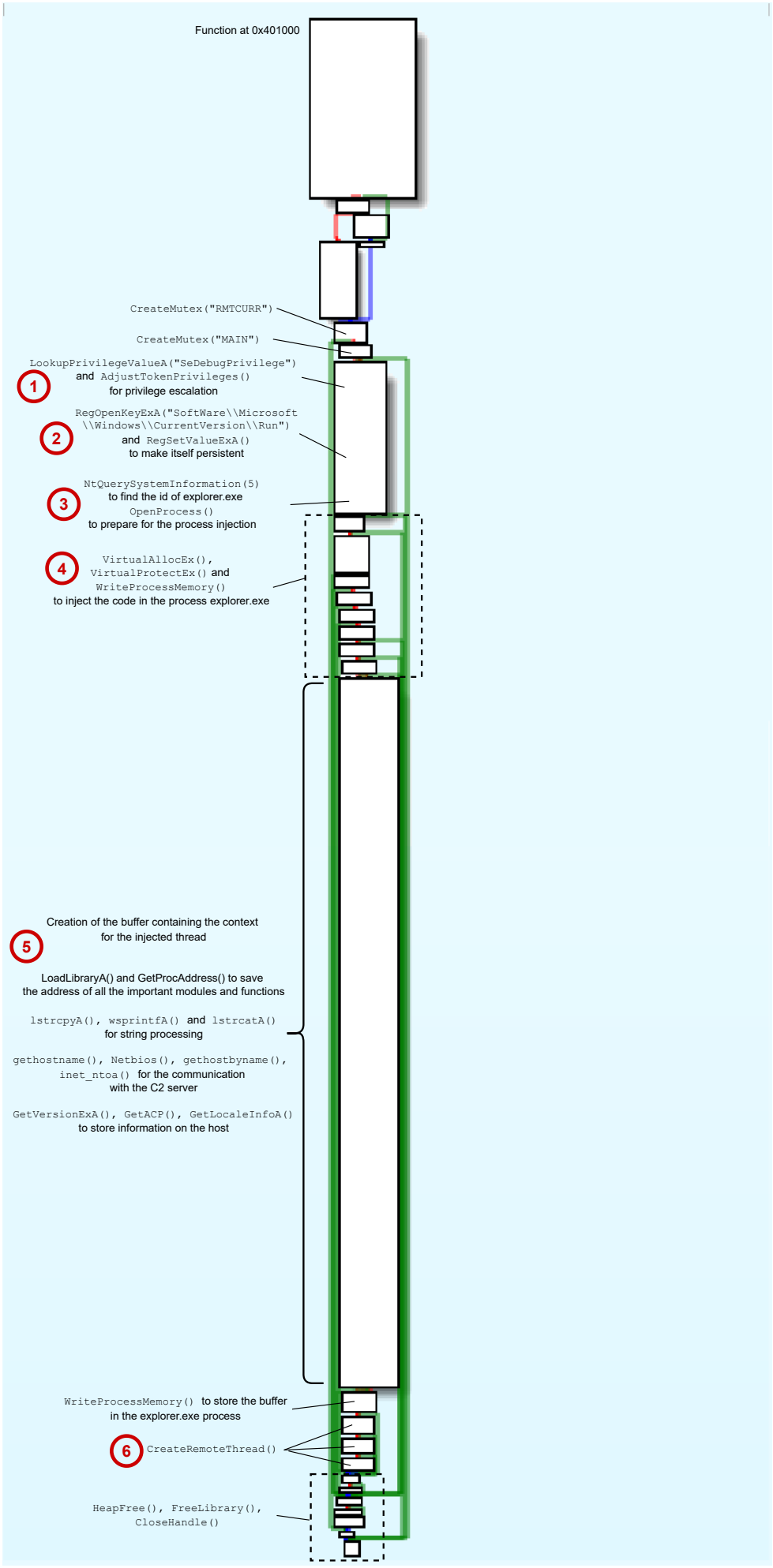

Figure 3.1: Function 0x401000 in Enfal

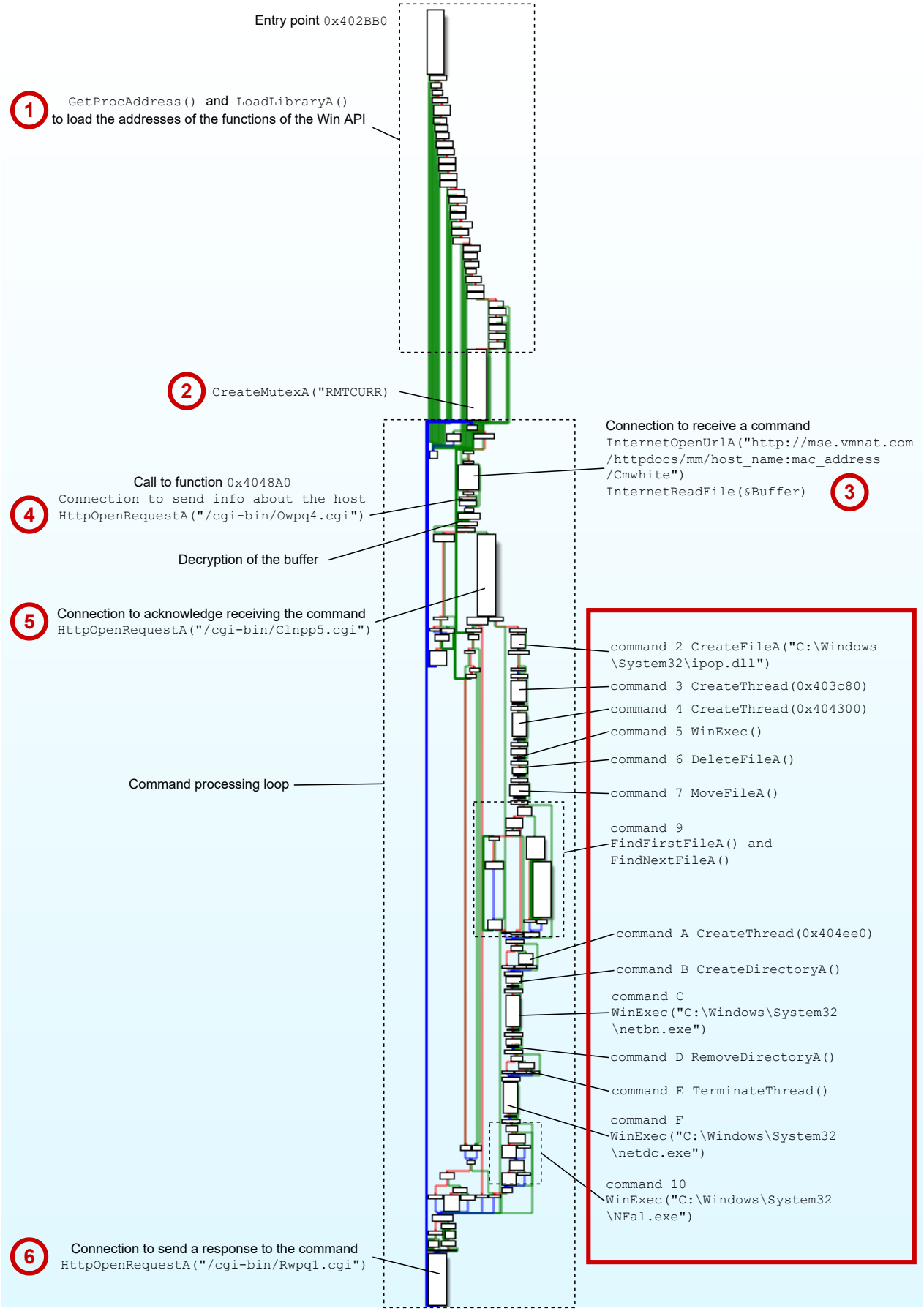

Figure 3.2: Injected function from Enfal in explorer.exe

3.2 Warzone RAT

3.2.1 Description

Warzone RAT (aka Ave Maria) is a RAT that is sold online by Warzone [33]. Different versions of this product exist. The most interesting one is the RAT poison, it presents the same features as the classical RAT but in addition to that, it is fully hidden (file, process and startup). Here is the list of the features that are found in the Warzone RAT 3.0

- Written in C++ (does not need the .NET Framework)
- Cookies recovery
- Remote desktop and hidden remote desktop
- Privilege escalation - UAC Bypass
- Remote webcam
- Password recovery
- File manager
- Download and execute files
- Keylogger online and offline
- Remote shell
- Process manager
- Reverse proxy
- Mass execute (on multiple targets)
- Persistence
- Windows defender bypass

The malware that we will analyze can be downloaded on MalwareBazaar. Its MD5 signature is 93c5434350e0f5dc53a88202ee48e531.
We can find a lot of information about the Warzone RAT online. First of all, there is a YouTube video [17] which explains in details the encoding of the configuration file. We learn that the configuration information are located in the bss section and are encrypted with RC4.
The author uses IDA (a tool similar to ghidra) to analyze statically the beginning of the malware, until the decryption of the configuration. This video gives us a good insight in case we need to use ghidra to analyze statically the code of the RAT in case we get stuck during the symbolic execution.
Check Point Research has written an article about the Warzone RAT [19]. They give a lot of technical details on the UAC bypass, the persistence, and most importantly, the network communication with a layout for the request and the response for each command of the RAT.

3.2.2 First Run and large loops

The first few runs quickly end up in very long loops. We could let it run for a few hours but since we want to be able to run the symbolic execution multiple times, we might as well try to help the program go through the loops faster.
The first loop is creating a hash of the file to make sure that there isn't an instance of the malware already running on the host. This loop and the value it produces is not critical for the execution. We can just modify the register that contains the counter to a smaller value.
The next loop is looking for the address of the beginning of the PE. This loop is important because the address will be used to locate the bss section in order to decrypt the configuration information. We can make this loop execute faster by injecting the address that it is looking for (`0x400000`) in the correct register. This address is the default base address of executable files for 32-bit images in windows. We could try to locate the functions that are looking for the beginning of a PE file, usually the function is trying to find the bytes 0x5a4d which signal the beginning of such file. If we can manage to automatically find the register which holds the address, we could automatize the process of skipping that loop.
During the execution, we see two functions that are called very often and that usually create large loops in the execution. The first one at address `0x405f10` copies a certain number of bytes from one place to another and the copy is done byte by byte. The second one at address `0x401052` fills a memory space with null bytes. The process is done four bytes at a time.
We can avoid having these loops executing by hooking each function and applying the effects at once to the concerned piece of memory. This will be much faster because each iteration was creating a new state while in this case we only have one new state which corresponds to the end of the loop.
At this point the execution is quite fast but it stops prematurely with an error. The problem is that there is no more space on the heap. This is not very surprising. The malware calls the function `HeapAlloc()` very often but does not seem to call `HeapFree()` that often. We can simply fix this by giving a bigger size to the heap when we initialize it.

3.2.3 SimProcedures

We keep having new API calls. When their SimProcedures are not implemented, they often create a fork, one path if it succeeds and one if it fails. It is interesting to force the function to not fail by returning only success values in order to reduce the number of state created and to avoid exploring useless fail paths. This is a small contribution to reduce the problem of path explosion.

Nevertheless, let's keep in mind that each time that we concretize a value, or discard a concrete value from a symbolic variable, we take the risk to discard a useful path. It could happen that a malware purposefully calls a function and expects to see it fail in order to continue the execution.

3.2.4 Command processing loop

Finally, we see the calls used to connect to the CC server: `getaddrinfo()`, `socket()`, `connect()` and `recv()`. They announce the arrival of the command processing loop.
The first interesting API calls that we see after recv are `URLDownloadToFileW` and `ShellExecuteW`. If we take a look at the analysis made by checkpoint [19], we see that the command 0x22 is a download and execute. Indeed, when we look at the code of the command processing loop in ghidra, the first set of if/else compares a value to 0x22.
Unfortunately if we let the execution follow its path, the program will loop and execute the same command at each iteration. This is the same problem that was encountered with the previous analysis.
We fix it in the same way as we did previously. We set a flag when a state arrives at the end of the command processing loop, and we discard each state whose flag is set. This way, each path executes the command processing loop exactly once. If everything goes well, each path should reveal one command of the command set.

3.2.5 Command set

As mentioned earlier, Checkpoint research provides a complete summary of the commands that can be sent to the malware. Most of the time, it is not too difficult to associate the calls from one iteration of the command processing loop to the command that was sent. For example the download and execute command gives us the `URLDownloadToFileW` and `ShellExecuteW` API calls, the command that enumerates the processes uses the functions command `CreateToolhelp32Snapshot()` and `Process32FirstW()`, the command that lists the disks calls the functions `GetLogicalDriveStringW()` and `GetDriveTypeW()`, and so on.
We can see in Figure 3.3 The control flow of the command processing loop. The control flow is quite messy and there are quite a large number of commands. The execution is very long if we want to see the whole loop. Especially because many commands dive into other threads to execute.

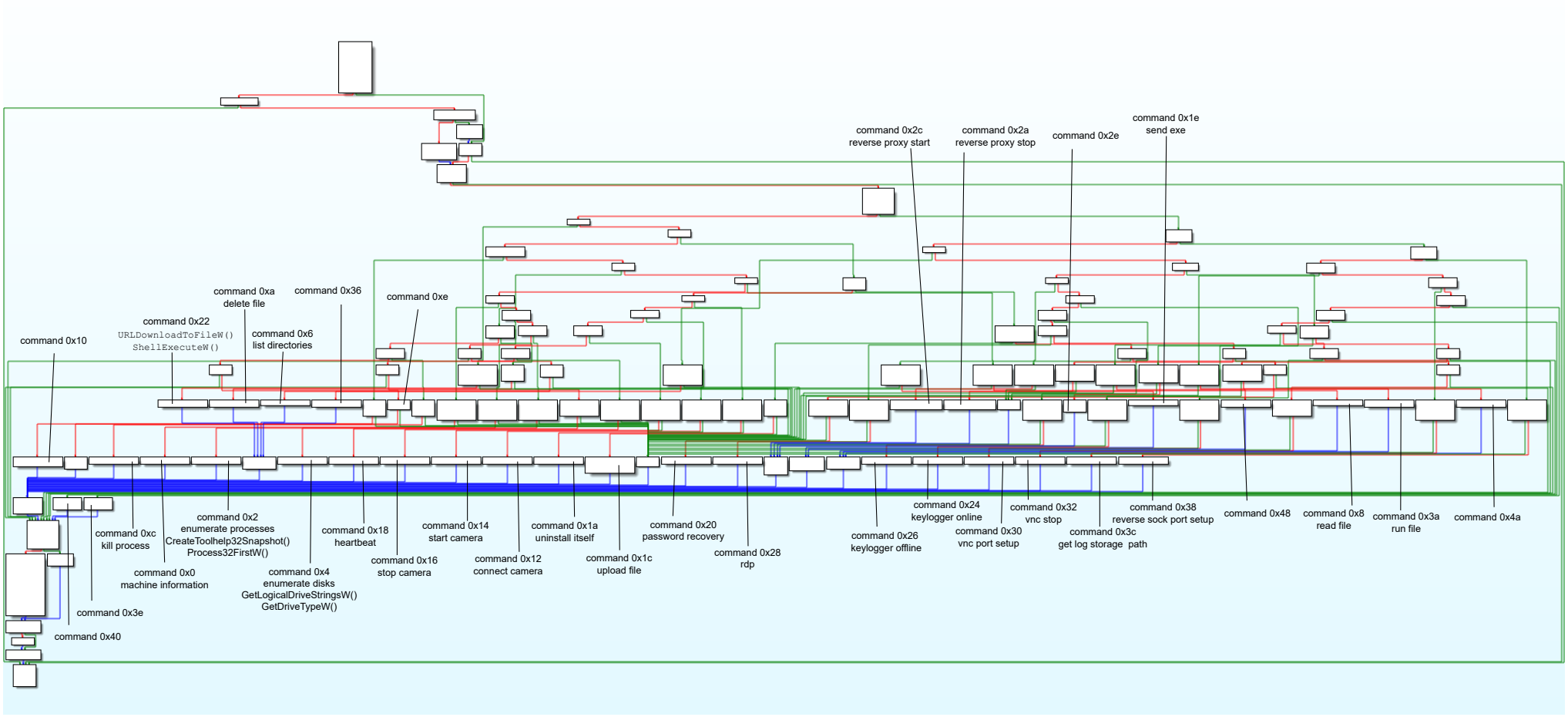

Figure 3.3: Command processing loop of the Warzone RAT

3.3 Satan RaaS

3.3.1 Description

In this section, we will analyze the Satan RaaS (Ransomware as a service). The sample that we will use can be downloaded from MalwareBazaar and its MD5 signature is c9c341eaf04c89933ed28cbc2739d325. This malware uses a lot of evasion and anti-debugging techniques. The first goal of this analysis is to write some SimProcedures with angr in order to overcome automatically these anti-analysis techniques. The second goal is to use all the techniques found on the path to guide the exploration.
In order to facilitate the analysis, we will find information in an article that analyses quite thoroughly the malware [26]. In this article, the author lists all the techniques used by satan to avoid being debugged or analysed. They explain that the malware uses process hollowing to drop an executable that contains the payload. The dropped malware makes itself persistant by creating an autostart registry, then it confirms the infection to its C&C server and finally it encrypts all the files on the host.

3.3.2 First run and SimProcedures

Let's start by running a first symbolic execution with the ToolChainSCDG to see how far it goes. The First thing that terminates the program is the check of dynamic-link libraries (dll) that are usually used for malware analysis, (2) in Figure 3.4. Indeed, the current SimProcedure for `GetModuleHandle` is able to find most libraries but the malware is expecting us to not find them to continue the execution. To counter this, we can create a list of libraries that are checked by the malware and make the function return 0 (function fails) if it is one of these libraries that is asked for. This list can and should be extended for better generalization.
Many other evasion techniques are simple calls to windows functions to try to detect a debug session. These kind of techniques can be overcome by simply writing or modifying the SimProcedure that is used for each call.

3.3.3 Detecting a debug session

Some techniques are harder to find or to overcome. For example, some debuggers traps such as `int3` or `xor eax eax` followed by `div eax` are crashing the symbolic execution, (8) and (11) in Figure 3.4. The first one was preceded by `AddVectoredExceptionHandler` and the second one by `SetUnhandledExceptionFilter`.

These functions are used to introduce a handler or a filter, the functions that are called when an error occurs. If the program is being debugged, the UnhandledExceptionFilter will not be called, instead the exception will be given to the debugger. The program can determine if it is being debugged by checking if the UnhandledExceptionFilter is executed or not. The same logic works for the VectoredExceptionHandler.
We can bypass this anti debugging technique by saving the address of the handler function in the SimProcedure of the `AddVectoredExceptionHandler` function. Then, we have to find statically the address of the exception and the length of the instruction(s) causing the problem to create a hook that will call the handler instead of executing the bad instruction(s).
It would be an interesting work to automatize the process of avoiding this type of evasion technique. One way to do this would be to keep track of the handlers/filters that are created and when an exception occurs in the symbolic execution, call the handler/filter that was saved and then go back to right after the bad instruction.

3.3.4 Encrypted strings

The next problem that came up during the analysis is that many strings were encoded in the file and were only decoded during the execution. The problem with this is that we can see that the malware checks the name of the document (`GetModuleFileName()`), the name of the user (`GetUserName()`) and some processes (`Process32First()` and `Process32Next()`) corresponding to number (5), (14) and (15) in Figure 3.4 but we can't tell what values are checked against the result. Indeed, the malware does not use strcmp() and instead has its own function to compare strings. The only way to find these values is to look at the code to find the functions where the strings are decoded and also where the strings are compared between them. Now we can add some breakpoints at specific addresses inside these functions to be able to recover the information from the registers. This is very specific to this malware but if we can find statically just the right address and the right registers, it is possible to better understand all those checks made by the malware in order to avoid analysis.
Having the list of these strings is not necessary for the execution but it helps us discover more evasion techniques to document some more the behaviour of the malware.
We had to make `Process32Next()` return false which usually means no more processes, because otherwise the malware would loop infinitely checking the name of the processes that the function would infinitely provide.

3.3.5 Anti dynamic analysis techniques

We will now list all the interesting API calls that are made in the Anti dynamic analysis function of the Satan RaaS. Each call will be associated with its number in the Figure 3.4 and a short explanation.
This section might be a bit redundant with section 2.3.4 but we will try to focus on the aspect of the SimProcedure associated with each call.

(1) `BlockInput(1)` is supposed to block the mouse and the keyboard if there is a debugger present. We simply return a nonzero value (success) in the SimProcedure.

(2) `GetModuleHandleW()` checks for the presence of malware analysis libraries. The SimProcedure of this function returns 0x0 (not found) when these libraries are given as input.

(3) `FindWindowW()` checks for the presence of the windows of well-known debuggers. The SimProcedure will return 0x0 (fail) when one of these names is given as input.

(4) `IsDebuggerPresent()` and `CheckRemoteDebuggerPresent()` check for the presence of a debugger. For the first one, the SimProcedure must return 0x0 (no debugger) and for the second one, we must assign the concrete value 0x0 to the second argument (pbDebuggerPresent).

(5) `Process32First()` and `Process32Next()` retrieve information about the processes running on a system. The name of the process is extracted to be compared with a list of processes names used for malware analysis. We can simply make both SimProcedures return 0x0 (no more processes). That way, we simply skip the check.

(6) `GetProcAddress()` checks for the presence of the function `wine_get_unix_file_name()` which would reveal that the malware is running in a Wine sandbox. The SimProcedure must return 0x0 (fail) when this function name is given in argument.

(7) `NtClose()` and `CloseHandle()` called with invalid handles can reveal the presence of a debugger because of an exception that they would raise. We do not have to implement the SimProcedure because these functions would not apply any modification to the limited environment that we have. Indeed, we do not keep track of opened handles so we don't have to close them either.

(8) `AddVectoredExceptionHandler()` introduces a handler that must be called at the next exception. The SimProcedure must simply save the address of the handler in the state. This way, we can call it when the exception occurs.

(9) `GetProcAddress()` is used to retrieve the address of external functions. Usually used to call the retrieved function right after, but in this case the function was never called. The malware was actually reading the code of the functions to make sure that they were not hooked.

(10) `OpenProcess()` called with the argument `"csrss.exe"` will succeed only if a debugger is present. The SimProcedure must return 0x0 (fail) when it is called wih this argument.

(11) `SetUnhandledExceptionFilter()`, similarly to `AddVectoredExceptionHandler()`, we must simply save the address of the filter to be able to execute it when the exception comes.

(12) `NtQueryInformationProcess()` retrieves information about the process. The information that is retrieved is in the third parameter (ProcessInformation). When the first parameter (ProcessInformationClass) is equal to 0x7 (ProcessDebugPort), 0x1E (ProcessDebugObjectHandle) or 0x1F (ProcessDebugFlags) we can put the value 0x0 in the third parameter to indicate that there is no debugger.

(13) `GetThreadContext()` returns a context which contains some fields for the debug registers. If the value of those fields is different than 0 then a debugger is present. We can concretize these fields to hold the value 0x0.

(14) `GetModuleFileName()` currently returns the actual name of the sample which in our case is its md5 hash. The name of the file is compared with values such as sample or virus. The comparisons were not matching so we did not have to modify the SimProcedure.

(15) `GetUserNameW()`, in the same way as `GetModuleFileName()` is not returning a name which matches the names to which it is compared. Those user names are from well known sandboxes as for example, TEQUILABOOMBOOM (VirusTotal sandbox).

3.3.6 Plugin and exploration technique

All these information that we found, can be stored in a state plugin. It would consist of an array filled with strings containing the name of the SimProcedure such as

"BlockInput(1)", "GetModuleHandle("dbghelp.dll")", "FindWindowW("Immunity Debugger")", etc. A state will only have in its plugin the anti-analysis techniques that it has found on his path so far. This way we can use it for an exploration technique. When a state is dead, we have to select another state that was paused to take back on the search. A good idea would be to choose the state that has overcome the most anti-analysis tricks because it probably means that it is on the right way to the interesting part of the malware.
This technique would be useful to guide the search at the beginning of the exploration but once all the checks are passed, there should be another heuristic taking on the exploration.

3.3.7 After the checks

At this point, we can see from the calls that are happening that the malware tries to read its own content (① in Figure 3.5). Thus, before going any further, we have to add the file containing the malware to the file system emulated by angr.
Unfortunately, the symbolic execution quickly falls into massive loops. With the use of Ghidra, we can determine that the first loop is calculating a checksum, then checking that it is equal to some value (② in Figure 3.5). We can skip this loop by modifying the value of the counter and the register containing the checksum.
The next loop is the decryption of the payload that is hidden in memory (③ in Figure 3.5). Unfortunately we cannot skip this loop, we have to execute it and it is very long. For the sake of future executions, we can save the result of the decryption during one execution, then create a hook for the loop that would directly put the decrypted text that we saved in memory where it is supposed to be.
The last loop is actually parsing the decrypted payload into its final buffer before the injection (④ in Figure 3.5). This loop should not be skipped either. Indeed, the program is placing the parameters needed for the process injection at the right offsets in the final buffer. Similarly to the previous loop, we can save the content of this buffer during an execution. Then, for the executions after that, skip the loop and put the buffer in memory manually.
All of these loop can be executed and do not have to be skipped. The problem is that they are taking a very very long time. If we want to repeat the execution multiple time, it is interesting to find the meaning of each loop and apply the effects at once. This way, we can avoid the thousands of new states created by the loops.
It might be interesting to introduce concolic execution to solve the problem of the large loops. Concolic execution consists in executing some part of the code with dynamic execution and some other parts with symbolic execution. The dynamic execution could execute those big loops faster than the symbolic execution. We

would just have to make sure that the variables used in those loops are concrete and that the loops are safe to execute dynamically.

3.3.8 Process Hollowing

Finally, we see the API calls used for the process hollowing as we can observe in the red rectangle in Figure 3.5. These calls are the exact same ones as those presented in section 2.3.3. We can implement all of their SimProcedures to return the value for success and to be able to see what was pointed by their arguments.
At first the SimProcedures were not taken into account because the name of the function was decrypted at runtime and thus was not in the symbols. The problem was coming from the `GetProcAddress` SimProcedure. Indeed, if the function was not loaded at the very beginning of the execution, then it was hooked to a stub function instead of the correctly implemented SimProcedure.
After fixing that, the execution is still not doing the whole process hollowing. This time it is the `VirtualAllocEx()` function that is causing the problem. Indeed, the process hollowing is trying to write at some specific addresses in the new process, such as `0x400000`. As mentioned earlier, we implemented `VirtualAllocEx()` in the same way as `VirtualAlloc()` because angr cannot manage different processes. The two processes are sharing the same address space. This means that the process injection is rewriting the code of the current process in angr with the payload of the malware. `VirtualAllocEx()` understands that this is not legal and so the function fails. We cannot write the new code at the right address so the process hollowing will not work. We can still save the code that is to be written somewhere else and make the function act as if it succeeded.

3.3.9 Conclusion

The process hollowing is not as easy to execute symbolically compared to a simple process injection like we had in the first analysis. Indeed, in this case a whole new PE is created and put in memory at a specific location, this requires a new symbolic execution to take place and the construction of the next PE.
We will not dive into this process for this analysis. We are more interested in the dropper to learn more about the evasion techniques rather than in the payload. Nonetheless, we could use the tool LIEF [27] to reconstruct automatically the injected PE then start a new execution from scratch.

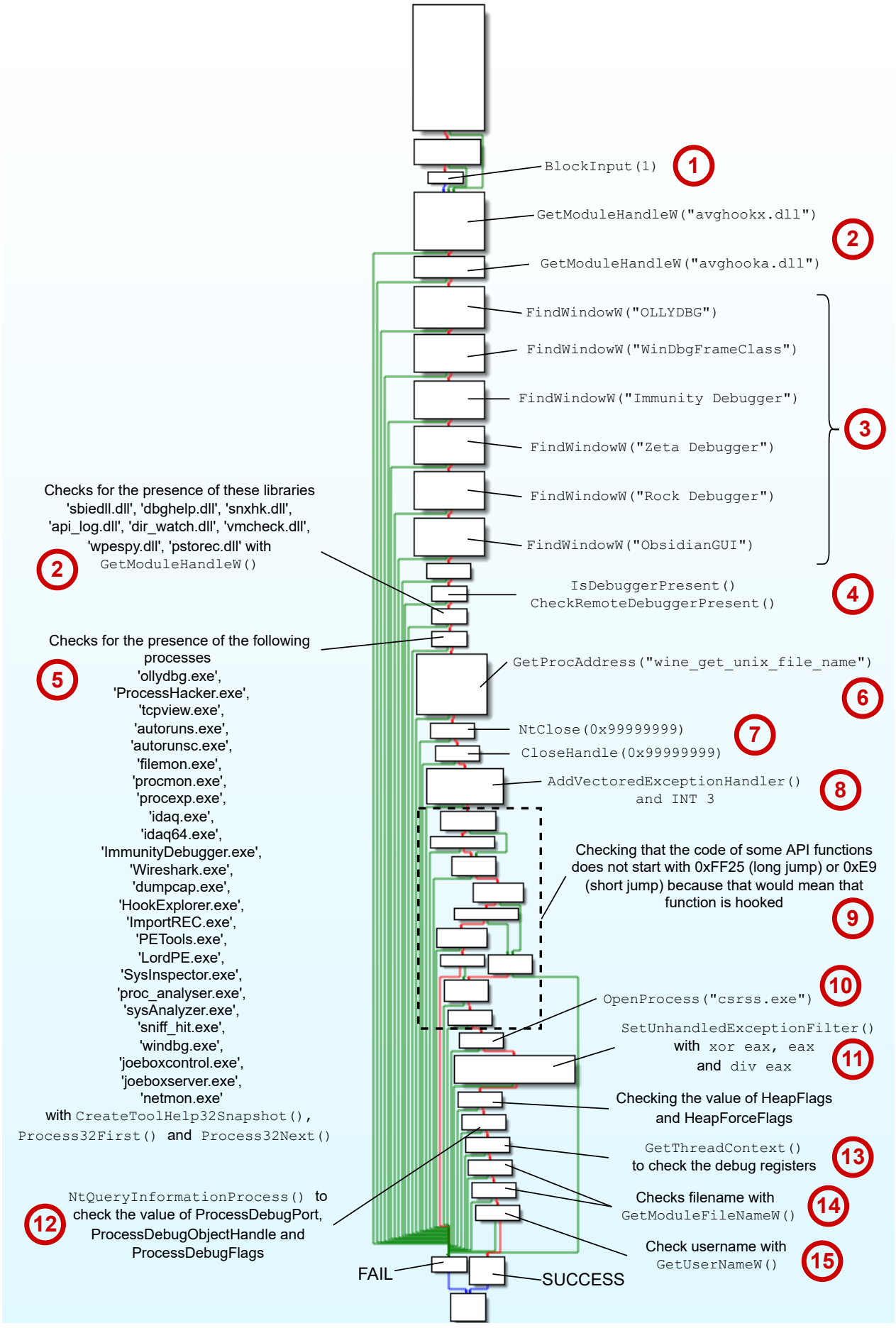

Figure 3.4: Anti dynamic analysis techniques in the Satan RaaS

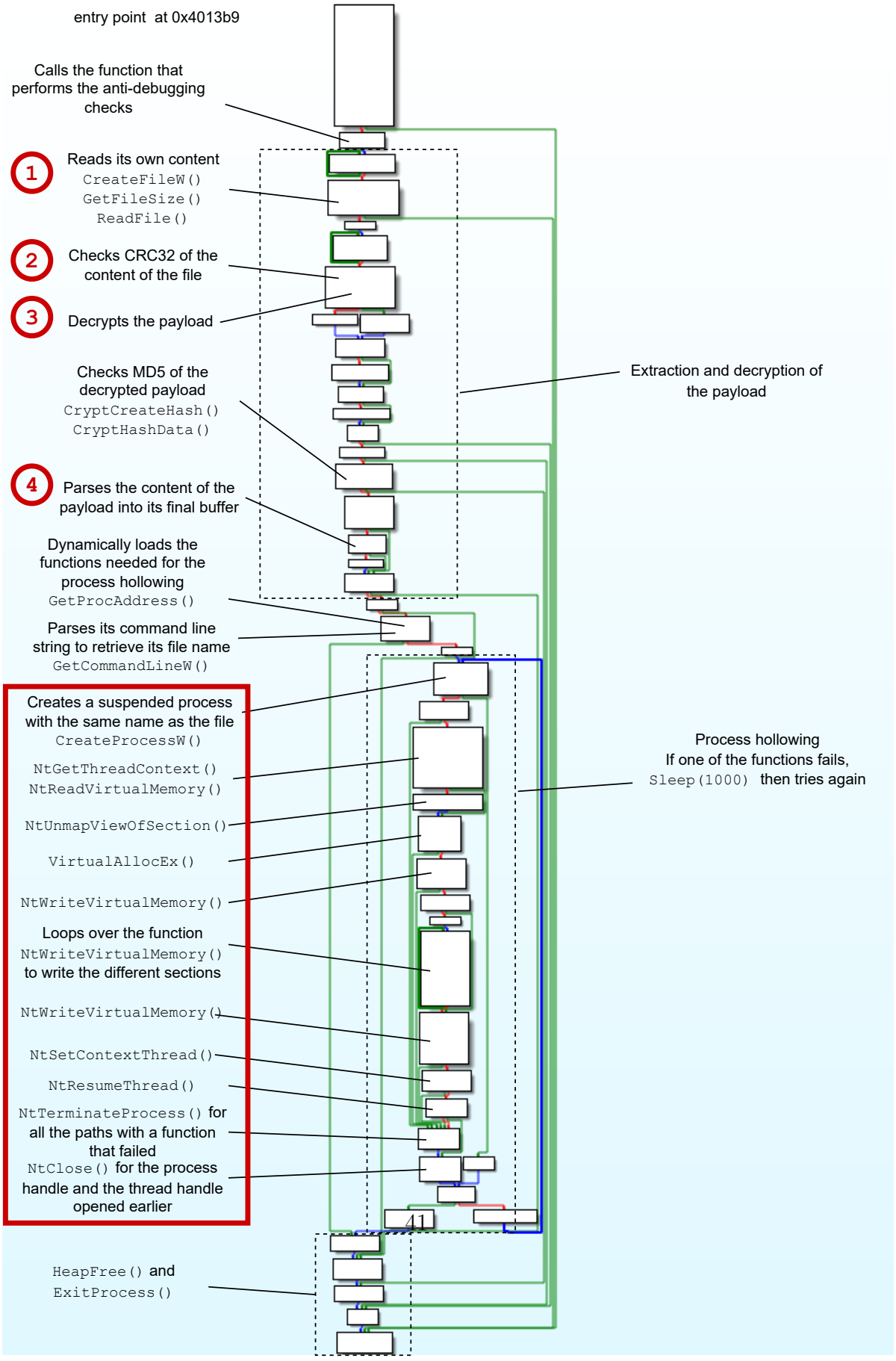

Figure 3.5: Satan RaaS

Chapter 4

Contribution

We showed that it is possible to use the SEMA Toolchain to perform a complete analysis on malware samples. We analyzed in depth three malware samples.
The first two samples were RATs. By applying symbolic execution with the use of the SEMA Toolchain we managed to get a clear understanding of the commands that could be executed by each RAT as well as their communication pattern.

For the third sample, we were able to see the execution of many anti dynamic analysis techniques and found ways to overcome each of them. We also have a clear idea of the functioning of a process hollowing and some thoughts about their integration in a symbolic execution.

Lastly, we put in light many challenges that occur when we want to analyze a malware sample with symbolic execution.
Among those challenges, we find the implementation of the SimProcedures with the compromise between concrete and symbolic for the different outputs. This is also a challenge when it comes to extending the modelization of the environment, we might want to give some concrete context to the execution but we must keep in mind that a more symbolic context will open more paths to explore.
We also talked at great length about the annoyances that large loops can create. It is tempting to find ways to automatize the process of skipping or accelerating the execution of these loops but it will definitely be a hard and tedious work.

We learned about the open problem of executing symbolically programs with concurrent threads or concurrent processes. During the first analyses we realized that it was still possible to inject a simple thread in another process if it does not interact with other threads or pieces of memory from the remote process. Indeed, in that scenario we simply inject the thread in the current process and let it execute independently just like a simple function call. In the third analysis, we learned

that this was not doable for a process hollowing. In this case the injected process needs its own address space and cannot simply execute in the same execution as the current process.

Finally, we realized that the SCDG and the json containing the calls are not very well suited if we want to analyze the control flow of the malware. It is possible to learn a lot on the behaviour of the malware with the list of the API calls but having them all listed in a json is not as visually comprehensible as a control flow graph that would contain these calls. This is obviously not a main point if we just want to use the symbolic execution to produce a SCDG to use for classification but if we want to integrate a feature to analyze more in depth a sample, it might be interesting to consider it.

Thoughts and suggestions to deal with those challenges are presented in the next section.

Chapter 5

Future work

On the things that we can do to keep improving the consistency of the symbolic execution, we obviously have the implementation of ever more SimProcedures. Indeed, these are essential to the well functioning of a symbolic execution. Some essential functions are already implemented by angr, mostly the functions that deal with the allocation of memory and such. We also implemented some functions during the analysis.
The most important functions to implement are the ones that have significant effects on the execution. Those that modify the memory or the state of the program as well as those that return important information that the malware needs to perform correctly.
We can also implement less critical functions to make them return a success value. This is not necessary but it will help the execution avoid the error paths which are usually uninteresting. This saves us some time and memory to explore more relevant execution paths.
Implementing these SimProcedures is not always easy. Some of the more complex ones may need an output that depends on the malware itself. We must be careful not to be too concrete in the output values so that it can be used in different scenarios but not too symbolic either so that it does not create a huge unnecessary path explosion. It is important to write the implementation conscientiously so that it benefits the execution of all the samples that use the Win32 API.

Another way to give the symbolic execution a more coherent environment is to try and add some structures or information that are found in a real environment. For example we could put a PEB and a TEB where they are supposed to be in memory. We could concretize or leave symbolic the different values depending on their level of interpretability. For example, a pointer to another structure could be concrete so that we can even try and fill this other structure at a specific place in memory. On the other hand, a value that defines the OS version better be

symbolic in case the malware has significantly different behaviour depending on the OS version it is running with.
If we concretize too much, we lose the interest of the symbolic execution and limit ourselves to the capabilities of a dynamic execution.
We could also think about adding some files to the system as well as some processes (actually just the `PROCESSENTRY32` that describe an existing process). This way we could use these pieces of information in the SimProcedures that try to list them. This would mostly be useful for `Process32Next()` and `FindNextFile()`.

We encountered many large loops during our analysis and talked about possible ways to automate the process of skipping or speeding up their execution. For now, most ways require human intervention ans static analysis of the code to find the meaning of each loop. Maybe we could have a collection of pieces of code that corresponds to a specific function that executes a large loop. For example, a function that looks for the beginning of the PE file, or a function that fills a piece of memory with zeros. Hopefully, these functions are reused in other samples and by scanning the code, we could automatically find them and hook them with an appropriate hook function.
Another, maybe more promising, idea would to use the concolic execution. The idea would be to let the execution run concretely when we reach a large loop, for example after 1000 iterations and stop the execution at the end of the loop to take back on with the symbolic execution. We would have to be mindful that the variables needed by the loop are concretized and that the loop does not execute any malicious code.
In practice, angr has developed a tool to perform concolic execution [18]. This tool, called symbion, allows us to easily go from symbolic to concrete execution and vice-versa. It is a very interesting tool that hopefully we can integrate to the SEMA Toolchain.

As mentioned in the analysis of Enfal 3.1, The SEMA Toolchain is not meant to produce a human readable report of the symbolic execution but rather to produce a signature in the form of a SCDG that can then be used to classify the sample. If for some reason we wish to analyze more in depth the behaviour of the malware, we can use the tool SymNav [3]. This tool uses symbolic execution to gradually build a control flow graph. It is also possible to interact with the execution and manually discard path that do not interest us. The symbolic execution is also using the angr framework and the tool has already been tested on malware. This tool seem really interesting and fun to play with in order to acquire more knowledge on the specific behaviour of different malware.

Finally, as we saw in the third analysis 3.3, it is not possible to perform a process hollowing with angr in a single execution. Indeed, with this type of process injection, a whole new PE file is created and replaces the code in the second process. Many malware use this technique to execute their code in another process in order to hide even more. It would be interesting for many malware analysis to have an automated way to execute symbolically the process hollowing.
One way to tackle this problem would be to use LIEF [27] . This python API will allow us to create a PE file from scratch. All we have to do is save the code that the malware wants to inject in each section, create the PE file and then start a new symbolic execution with this newly created executable.
If this technique succeeds, it will allow us to terminate the analysis on the Satan RaaS by executing its payload.

Chapter 6

Conclusion

To conclude, we can say that it is definitely possible to conduct a complete and consistent symbolic execution on malware although for now, it sometimes requires some human intervention. RATs are particularly susceptible with their infinite command processing loop and their complex communication pattern with their command and control server.

The main goal of this work was to analyze RATs with symbolic execution which we managed to do for two samples. We showed a methodology that we hope can be used to analyze more of them and more rapidly in order to scale up to the large number of malware that appear every day.

We also showed an example of symbolic execution against anti dynamic analysis techniques. We saw that it worked fairly well and that symbolic execution can even use them to guide the exploration to the interesting malicious part.

Many challenges were brought to light during our analysis. All these challenges gave us some ideas to keep improving the SEMA Toolchain.

Hopefully, symbolic execution will soon be a widely used technique for malware analysis.

Bibliography

[1] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. Malware dynamic analysis evasion techniques: A survey. *ACM Computing Surveys (CSUR)*, 52(6):1–28, 2019.

[2] National Security Agency. ghidra. `https://github.com/NationalSecurityAgency/ghidra`.

[3] Marco Angelini, Graziano Blasilli, Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, Simone Lenti, Simone Nicchi, and Giuseppe Santucci. Symnav: Visually assisting symbolic execution. In *Proceedings of the 15th IEEE Symposium on Visualization for Cyber Security*, VizSec '19, 2019.

[4] angr. Overarching Research Directions. `https://docs.angr.io/introductory-errata/helpwanted#overarching-research-directions`.

[5] MITRE ATTACK. Process Injection. `https://attack.mitre.org/techniques/T1055/`.

[6] MITRE ATTACK. Virtualization/Sandbox Evasion. `https://attack.mitre.org/techniques/T1497/`.

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. Assisting malware analysis with symbolic execution: A case study. In *International conference on cyber security cryptography and machine learning*, pages 171–188. Springer, 2017.

[8] Charles-Henry Bertrand Van Ouytsel, Christophe Crochet, Khanh Huu The Dam, and Axel Legay. SEMA - toolchain using symbolic execution for malware analysis. `https://github.com/csvl/SEMA-ToolChain`, 2022.

[9] Charles-Henry Bertrand Van Ouytsel and Axel Legay. Malware analysis with symbolic execution and graph kernel. *arXiv e-prints*, pages arXiv–2204, 2022.

[10] Fabrizio Biondi, Thomas Given-Wilson, Axel Legay, Cassius Puodzius, and Jean Quilbeuf. Tutorial: An overview of malware detection and evasion techniques. In *International Symposium on Leveraging Applications of Formal Methods*, pages 565–586. Springer, 2018.

[11] Alejandro Calleja, Juan Tapiador, and Juan Caballero. The malsource dataset: Quantifying complexity and code reuse in malware development. *IEEE Transactions on Information Forensics and Security*, 14(12):3175–3190, 2018.

[12] CheckPoint. Anti-Debug Tricks. `https://anti-debug.checkpoint.com/`.

[13] CheckPoint. What is Remote Access Trojan (RAT)? `https://www.checkpoint.com/cyber-hub/threat-prevention/what-is-remote-access-trojan/`.

[14] TechTarget Contributor. RAT (remote access Trojan). `https://www.techtarget.com/searchsecurity/definition/RAT-remote-access-Trojan`.

[15] Mark E. Russinovich David A. Solomon. Processes, Threads, and Jobs in the Windows Operating System. `https://www.microsoftpressstore.com/articles/article.aspx?p=2233328&seqNum=4`.

[16] ercoppa. Win32 API models. `https://github.com/ercoppa/angr/commit/09a523a12cf576975e4740e3111b2719d3fd607a`.

[17] Sergei Frankoff. Reverse Engineering Warzone RAT - Part 1. `https://www.youtube.com/watch?v=81fdvmGmRvM`.

[18] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. Symbion: Interleaving symbolic with concrete execution. In *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*, June 2020.

[19] Yaroslav Harakhavik. Warzone: Behind the enemy lines. `https://research.checkpoint.com/2020/warzone-behind-the-enemy-lines/`.

[20] Adam Kujawa. You dirty RAT! Part 1: DarkComet. `https://blog.malwarebytes.com/threat-analysis/2012/06/you-dirty-rat-part-1-darkcomet/`.

[21] Anitta Patience Namanya, Andrea Cullen, Irfan U Awan, and Jules Pagna Disso. The world of malware: An overview. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 420–427. IEEE, 2018.

[22] Nick Raziborsky. Lilith. https://github.com/werkamsus/Lilith.

[23] Ricardo J Rodríguez, Xabier Ugarte-Pedrero, and Juan Tapiador. Introduction to the special issue on challenges and trends in malware analysis. *Digital Threats: Research and Practice (DTRAP)*, 3(2):1–2, 2022.

[24] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[25] lockshaw subwire. throwing a tantrum, part 1: angr internals. https://angr.io/blog/throwing_a_tantrum_part_1/.

[26] The BlackBerry Cylance Threat Research Team. Threat Spotlight: Satan RaaS. https://blogs.blackberry.com/en/2017/02/threat-spotlight-satan-raas.

[27] Romain Thomas. Lief - library to instrument executable formats. https://lief.quarkslab.com/, apr 2017.

[28] Minh Tran. A Deep Dive Analysis of the FALLCHILL Remote Administration Tool. https://www.fortinet.com/blog/threat-research/a-deep-dive-analysis-of-the-fallchill-remote-administration-tool.

[29] Minh Tran. A Deep-Dive Analysis of the NukeSped RATs. https://www.fortinet.com/blog/threat-research/deep-analysis-nukesped-rat.

[30] Cryillic TryHackMe. Abusing Windows Internals. https://tryhackme.com/room/abusingwindowsinternals.

[31] Veronica Valeros and Sebastian Garcia. Growth and commoditization of remote access trojans. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 454–462. IEEE, 2020.

[32] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. Linear obfuscation to combat symbolic execution. In *European Symposium on Research in Computer Security*, pages 210–226. Springer, 2011.

[33] WARZONE. WARZONE RAT. https://warzone.ws/index.html.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**