



École Polytechnique Fédérale de Lausanne

Process Injection for Threat Simulation

by Jonas Konrad

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer  
Thesis Advisor

Sylvain Heiniger  
External Expert

Sylvain Heiniger  
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE  
BC 160 (Bâtiment BC)  
Station 14  
CH-1015 Lausanne

March 14, 2024

# Acknowledgments

I would like to express my sincere gratitude to Sylvain Heiniger, my thesis supervisor, for his guidance, and support, throughout this project. Thank you very much to Prof. Mathias Payer for advising my master thesis.

I extend my heartfelt appreciation to Compass Security, for providing me with the opportunity to conduct my research within their organization. I am grateful for the resources, access to company knowledge, and practical insights that I gained during my time at Compass Security. Special thanks to Patrick Vananti and the entire Bern office team for their assistance and cooperation during the course of this project.

I am sincerely grateful to my girlfriend and to my parents for their unwavering support throughout my time at EPFL.

*Bern, March 14, 2024*

Jonas Konrad

# Abstract

Threat simulation, also known as red teaming, is used to evaluate the security stance of a company, uncover its weaknesses, and identify improvement potential. Custom tools are important for this task, since they are less likely to be detected by security solutions, and will better emulate the threat posed by real-world advanced malicious actors. Process injection is a technique commonly used when attacking a system. Many techniques and implementations are publicly available, but most are easily detected in practice. In this project, we present the knowledge required to understand, research, and evaluate the effectiveness of process injection techniques for Windows systems. We show how to efficiently and reliably test against modern, cloud-based detection technologies. With the example of the Windows memory enclave API, we show the pitfalls of research in this subject. We demonstrate that, though it provides in theory all the desired characteristics, this API cannot be used for process injection in practice. Finally, we improved upon an existing injection technique, by extending its functionality, and making it undetected by a state-of-the-art security solution, without the need for advanced evasion techniques. Our implementation can be used to establish a reverse shell in the presence of Microsoft Defender XDR, even when using a heavily signed payload.

# Contents

<b>Acknowledgments</b>	<b>2</b>
<b>Abstract</b>	<b>3</b>
<b>1 Introduction</b>	<b>7</b>
<b>2 Background</b>	<b>9</b>
2.1 Operating System . . . . .	9
2.1.1 Processes and Threads . . . . .	10
2.1.2 Address Spaces . . . . .	11
2.2 Windows . . . . .	11
2.2.1 Architecture . . . . .	12
2.2.2 Windows API and System Calls . . . . .	14
2.2.3 Portable Executable . . . . .	15
2.2.4 Processes and Threads . . . . .	16
2.2.5 Virtual memory . . . . .	17
2.3 x64 Architecture . . . . .	17
2.4 Mitigations . . . . .	18
2.4.1 Data Execution Prevention . . . . .	18
2.4.2 Address Space Layout Randomization . . . . .	18
2.4.3 Control Flow Integrity . . . . .	19
2.4.4 Virtualization Based Security . . . . .	20
2.5 Detection . . . . .	20
2.5.1 Windows Native Telemetry Sources . . . . .	21
2.5.2 Antivirus . . . . .	21
2.5.3 Endpoint Detection and Response . . . . .	22
2.6 Microsoft Defender XDR . . . . .	23
2.7 Evasion Techniques . . . . .	24
2.8 Memory Enclaves . . . . .	24
2.9 Process Injection . . . . .	26

<b>3</b>	<b>Design</b>	<b>30</b>
3.1	Design Constraints . . . . .	30
3.2	General Design Decisions . . . . .	33
3.3	Threadless Injection . . . . .	34
3.3.1	Allocation . . . . .	35
3.3.2	Writing . . . . .	36
3.3.3	Execution . . . . .	37
3.4	Injection using Memory Enclaves . . . . .	38
3.4.1	Payload Execution in Memory Enclaves . . . . .	38
3.4.2	Enclaves as an Allocation and Writing Primitive . . . . .	40
3.4.3	Injection using VBS Enclaves . . . . .	41
<b>4</b>	<b>Implementation</b>	<b>43</b>
4.1	Threadless Injection . . . . .	43
4.1.1	Shellcodes . . . . .	43
4.1.2	Parsing the Export Table . . . . .	45
4.1.3	Evasion . . . . .	45
4.2	Injection using Memory Enclaves . . . . .	47
4.2.1	SGX Enclaves . . . . .	47
4.2.2	VBS Enclaves . . . . .	49
<b>5</b>	<b>Evaluation</b>	<b>50</b>
5.1	Challenges of Testing against Security Solutions . . . . .	50
5.2	Design . . . . .	51
5.2.1	Methodology . . . . .	54
5.2.2	Gathering Results from Defender XDR . . . . .	56
5.2.3	Setup . . . . .	57
5.3	Results . . . . .	58
5.3.1	Baseline . . . . .	59
5.3.2	Original Threadless Injection . . . . .	60
5.3.3	Improved Threadless Injection . . . . .	60
<b>6</b>	<b>Related Work</b>	<b>62</b>
6.1	Known Techniques . . . . .	62
6.1.1	DLL . . . . .	63
6.1.2	Callback . . . . .	63
6.1.3	Other . . . . .	63
6.2	Systematization . . . . .	65
6.2.1	Explore Underutilized Operating System Functionality . . . . .	65
6.2.2	Documentation Quality of Operating System Functionality . . . . .	65
6.2.3	Account for Implications of the Underlying Technology . . . . .	66
6.2.4	Understand the Complexity of the System . . . . .	66

<b>7 Conclusion</b>	<b>67</b>
<b>Bibliography</b>	<b>69</b>
<b>A ETW-TI Events</b>	<b>77</b>
<b>B ThreatCheck</b>	<b>79</b>
<b>C CAPA</b>	<b>81</b>

# Chapter 1

## Introduction

Advanced threat actors, such as the Lazarus Group [40] and their actions [93], have gained increased public awareness. Organizations are performing penetration tests or threat simulation exercises against their own infrastructure to improve their security stance in the face of the threat posed by such groups. Threat simulation exercises, also called red teaming exercises, aim to simulate attacks as they would be performed by a real world malicious actor. In contrast to penetration tests, this implies that the blue team of the organization is not aware of the exercise. The former tests in depth, whilst the latter tests in breadth. Threat simulations hence require custom tools, such as they would be used in real world attacks, to reliably avoid detection. Indeed, advanced actors use tools whose implementation, or even technique, are not public. Through the usage of custom tooling, ethical hackers can emulate their behavior more realistically.

Malware often requires knowledge in breadth of the target system. Since enterprise infrastructure usually uses the Windows operating system on client machines, knowledge of it is required when using process injection. Windows is closed-source software, whose internals are scarcely documented and few academic work is done on it. This leads to a high barrier of entry when first working with this operating system. The same is true for modern security solutions. State-of-the-art solutions are black box applications, whose central logic and functionality executes in a cloud-computing environment. When preparing for a threat simulation exercise, any tool and software that will be used has to be tested on an environment mimicking what will be found in the target organization. This involves testing if these tools, which are for all intents and purposes malware, are detected and blocked by security software. Consequently, during the preparation phase, the target security solution has to be accounted for. A poor understanding of its detection logic makes this task difficult.

Process injection is a technique used to improve malware stealthiness, or to elevate privileges during an attack. There are many publicly available process injection techniques and implementations thereof. Due to the fragmented nature of research on the subject, found mostly in blog posts

and proof-of-concept tools published on GitHub, there is no consensus on naming and terminology. This makes it time intensive to find out what the current state-of-the-art techniques are, or even to gain an overview of which have been published. Since detection logic adapts to emerging threats, it is important for red teaming operators to understand how and why their malware is detected, so that they are able to write their own implementations and adapt them to their needs. The challenges of evaluating a technique against modern detection solutions is an under-explored topic. Injection techniques that are detected are useless in practice, hence only evaluating them for their functionality is an incomplete approach. Very few published techniques claim evasion capabilities in practice, and if they do, no actual evaluation results are provided.

The reference work on the Windows operating system are the *Windows Internals* book volumes one and two [1, 7], which go into great detail on the internal workings of the operating system. Whilst it should be considered as mandatory reading to get a good overview of the subject, we aim to provide the necessary baseline knowledge of both general operating system, and Windows specifics, to enable the reader to work efficiently on process injection. The common resources on process injection rarely present this knowledge, and if they do, the prerequisite knowledge is either too high, or the explanations do not go into enough detail. The same is true for documentation or research on existing mitigations and security solutions. In this project, we present the required knowledge of the latter, and apply it to our evaluation methodology. We argue that current public knowledge and understanding of how state-of-the-art detection is implemented is poor and too little care is given on evaluating proposed injection techniques.

Researching process injection is closely related to researching blind spots in detection, as well as finding exploits in legitimate operating system functionality. By trying to develop a novel technique leveraging the built-in Windows memory enclaves API, we illustrate the pitfalls researchers should expect to encounter. We demonstrate why this API cannot be used for process injection in combination with Intel SGX enclaves. The possible designs using VBS enclaves are presented, one of which we argue could in theory be implemented.

Process injection is a vast topic with many interdependent parts. It is closely related to the target system and its configuration. This makes general statements on its performance difficult. Through this project, we aim to provide an entry point to red teaming operators which gives them the knowledge required to utilize process injection in their threat simulation assignments efficiently and with success. We introduce an evaluation methodology and guide to determine whether an implementation evades detection, and if it does not, how to find out which parts need to be improved. A companion tool that can be used when evaluating against Microsoft Defender XDR [48] to efficiently export information necessary for this evaluation is made available in this project's repository. We also implemented an improved version of the known Threadless Injection technique. Our implementation successfully injects and executes a heavily signed payload in the presence of Defender XDR.



## Chapter 2

# Background

This project requires background knowledge in operating systems (OS) and detection techniques. We introduce the reader to operating systems in general, and to Windows specifics. Security mitigations implemented in Windows are presented, and how security solutions, such as Microsoft Defender XDR, provide detection capabilities. Finally, we define process injection. We recognize that the background section of this project is long despite our best efforts to keep it at a minimum. Indeed, in depth understanding of process injection requires knowledge of a large part of the operating system, and of detection techniques in general.

### 2.1 Operating System

The *operating system* (OS) manages the correct and efficient use of both hardware and software resources. It is a complex system with many elements, such as input/output, memory allocation, and file system management. Every component of the OS can be categorized in one of four categories: virtualization, concurrency, persistence, and security [3]. We focus on the topics most relevant to this project, namely address spaces, processes and threads. There are different kinds of operating systems architectures. We focus on monolithic kernel operating systems, i.e., operating systems whose entire operating system executes in kernelspace, since this category of OSes are the most prevalent on desktop computers (e.g., Linux, macOS, and Windows).<sup>1</sup>

---

<sup>1</sup>Apple and Microsoft actually refer to their OS as being a hybrid kernel, meaning that some parts of the kernel run in userspace. This distinction is not relevant for this section (and arguably the same as a monolithic kernel).

### 2.1.1 Processes and Threads

Modern computers execute multiple programs concurrently, but the CPU has no notion of this. It simply executes an endless stream of instructions. This concurrency capability is provided by the operating system. The concept of virtualization is introduced to make concurrency transparent to the program. Virtualization provides the illusion to each program that it is the only one running on the system. This technique is applied to CPU resources, and memory. We present the former now, and the latter is explained in subsection 2.1.2.

We first introduce definitions and concepts. A *program* is static code and data residing somewhere in storage. A *process* is an instance of a program currently executing. There can be more than one instance of the same program at the same time. Each process has an associated *process state* which captures everything the executing instructions can affect, e.g., registers or persistent storage. This state exists both from the point of view of the OS, and from the process itself. It might contain different information in each point of view.

A process has at least one *thread* of execution. All threads of the same process share the same process state, but each thread has its own stack and set of registers. The *stack* contains the function invocation frames, calling context, and sequence of currently active frames. Basically everything that is local to a function call and the current scope. Threads are not strictly part of the resource virtualization concept. They are instead a concept of the process, and hence the programmer is aware of them. Threads are what is scheduled for execution by the OS, though they can in some cases also be scheduled by the program itself.

*Scheduling* is how the operating system decides when to let which thread execute on the *central processing unit* (CPU). Each CPU execution core can execute one thread at a time<sup>2</sup>. The OS has to choose which thread is allowed to execute if there are more threads than execution units, effectively implementing *time-sharing* between each of them. When changing which thread is executing, a *context switch* is performed, where the process state is changed. Scheduling challenges such as fairness and efficiency are not relevant for this project. We refer to the OSTEP book [3] for further reading.

The combination of address spaces (see subsection 2.1.2) and per-process program state leads to isolation between processes. Processes are not aware of the existence of other processes and cannot access their resources directly. This isolation is enforced by the OS, with the help of hardware mechanisms. Additionally, processes are isolated from the operating system as well. Indeed, processes run in *userspace*. They have no direct access to hardware resources, and functionality implemented in the operating system kernel.

---

<sup>2</sup>We note that this is not strictly true on modern superscalar CPUs with hardware multithreading, where one core can execute multiple threads at the same time by utilizing otherwise wasted execution cycles between (some) instructions. An example of this is SMT, of which *Hyperthreading* is the Intel proprietary implementation of.

The operating system has higher privileges than other processes. This is enforced using the different execution modes provided by the CPU architecture. One of them will be used for user mode and another for kernel mode execution. In x64 these modes are called ring 3 and ring 0 respectively. The isolation between these execution modes is provided by the hardware, as is explained in section 2.3, but leveraged by software. For example, if a user mode application wants to create a new process, it will have to call an API function exposed by the kernel. During this function call, the execution mode will change from user mode to kernel mode. Once this is done, the kernel code implementing process creation is reachable. Note that kernel mode code has access to the entire user mode functionality, but the inverse is not true. Additionally, some instructions are available only in kernel mode.

### 2.1.2 Address Spaces

The time-sharing mechanism introduced in subsection 2.1.1 enables virtualization of the computation resources. For memory, space has to be shared, and not time. This means, that the same mechanism cannot be used to virtualize memory. *Space-sharing* is used instead of time-sharing [3].

Memory is accessed using an *address*. The set of all addresses available to a process is called its *address space*, and is essentially a map from pointers to bytes. This means that the same pointer value (i.e., the same address) in different processes can point to different bytes. Because the process uses addresses that do not match the actual, physical addresses in memory, the address space is also called *virtual* address space. The OS, with the help of hardware, interposes itself in each memory transaction. The *virtual addresses* (VA) the process uses are translated into the corresponding *physical addresses* (PA). Each process can allocate memory in its own virtual address space, as if it had access to the entire physical memory. This OS enforced interposition mechanism allows sharing memory between programs, even if they each have their own virtual address spaces.

Modern operating systems allow dynamic memory allocation at page granularity. A *page* is a fixed-size, contiguous area of memory. Each page has an associated *memory protection* set regulating how it can be accessed. This can be any combination of read, write, and execute permissions. Exactly how virtual memory is implemented, and more complicated issues related to virtual memory are not relevant for this project. We refer to the OSTEP book [3] for further reading.

## 2.2 Windows

In this section we introduce the specificities of the Windows operating system required to understand this project. Windows server versions have slight differences to desktop versions; we focus on the desktop version in this project. When working with Windows, since it is closed source, we rely heavily on reverse engineering and third party documentation. Indeed, the official documentation provided

by Microsoft [59] provides valuable insights, but many of the OS components and implementation details are undocumented. The reference work on Windows is Windows Internals [1, 7], which has not yet been updated to include the features introduced by the latest version. Because of this, we focus on Windows 10 in this project, and not on the more recent Windows 11. Another useful resource is the ReactOS project [80], an open-source Windows compatible operating system, and its documentation [81]. Finally, Microsoft provides some resources that make reversing the OS and exploring its internal structure easier. These are the *Software Development Kit* (SDK) [71], the *Windows Driver Kit* (WDK) [46], the Windows debugger [53], and the public debugging symbols [66].

There exist more than one version of Windows; we focus on the 64-bit desktop Windows version. Other versions, such as the one running on Xbox game consoles, or on ARM based systems are out of scope. Additionally, we gloss over many aspects that are not relevant to this project. Windows is a large OS with many layers, functionalities, and complexities linked to maintaining backwards compatibility. We refer to the Windows Internals book [1, 7] and the Windows documentation [59] for further reading. We note that some content in this section refers to undocumented interfaces and components, implying that they are subject to change.

### 2.2.1 Architecture

In this subsection, we present an overview of the Windows architecture and its key system components. A simplified architecture is represented in Figure 2.1. There are different process types executing in user mode [7]. *System processes* are started at boot and will always appear on a system. The *service control manager* is a special system process responsible for starting *services*. These run independently of user sessions and are started every time the computer boots. They are either server processes, or device drivers. Services are transparent to the user. For example, dialog boxes raised from within them will not be shown to the user. Another special system process running only kernel mode threads is called *System Process*. Finally, *User processes* are started by the user from executable images.

The concept of *subsystem* is important in the architecture of the Windows operating system. The idea is to expose only a subset of the base Windows executive to application programs. Each application links against exactly one subsystem. A subsystem is composed of at least one DLL, possibly drivers, executables, and other subsystems. A subsystem can be implemented across both user and kernel mode.

As we already noted, the operating system operates with different privileges than other code executing on the machine. A mechanism is in place to transition between these privilege levels, i.e., from user mode to kernel mode. At the core of this mechanism, is the `syscall` instruction, which is presented in more detail in subsection 2.2.2. It triggers a transition in the CPU from ring 3 to ring 0. The Windows component where these transitions are intended to take place is `ntdll.dll`. This library exports *system calls*, i.e., it implements calls to functionalities of the Windows API which

are themselves implemented in the kernel. These are internal functions, meaning that they are, to a large extent, undocumented. Programmers are supposed to use *subsystem DLLs* instead of calling exported `ntdll` functions directly. For examples, the main Windows subsystem DLLs are `Kernel32.dll`, `Advapi32.dll`, `User32.dll`, and `Gdi32.dll`<sup>3</sup> [7]. Starting with Windows 7, the Windows kernel developers have incrementally been moving functionality to new low-level binaries, making the existing ones smaller and less complex, at the cost of introducing new ones [61]. For example, much of the implementations of `Kernel32.dll` have been moved to `Kernelbase.dll`. Some Windows API functions are now implemented entirely in userspace in some of these subsystem DLLs<sup>4</sup>. If an API function is not implemented entirely in userspace, then the subsystem will call the corresponding function in `ntdll`, from where the transition into kernel mode will take place.

The `ntdll.dll` library is the lowest common denominator of any Windows subsystem [7]. *Native* images are images that are not tied to any particular subsystem, meaning that they only link to `ntdll`. Because this library is mostly undocumented and subject to change, mostly executables built by Microsoft themselves link directly against it<sup>5</sup>. This is why, in Figure 2.1 only environment subsystems and system process are shown to directly interact with it.

Both the executive and kernel are implemented in `Ntoskrnl.exe` [7]. The routines exposed by the *executive* layer are called during transitions from user to kernel mode. Additionally, it contains functions that are exported, but only callable from kernel mode. These are mostly documented for usage by driver developers, but some functions are undocumented because they are intended for the kernel developers.

The *kernel* provides the fundamental OS mechanisms, such as scheduling, interrupts, etc. No policies are implemented in the kernel, they are all in the executive [7]. The kernel utilizes *objects* to represent shareable resources, such as processes and threads. Objects are opaque data structures, and hence interacting with them requires using Windows API functions. Accessing them directly is not possible from userspace, since they reside in the virtual address space of the kernel. Access control to these objects is enforced by various security checks implemented in the kernel. References (i.e., pointers) to kernel objects are called *handles*. Getting access to an object is referred to as *opening a handle to an object*. For internal usage in the kernel, these objects are represented using *kernel objects*, that do not require the additional overhead incurred by security checks in place when accessing objects.

The *hypervisor*, whose implementation in Windows is called Hyper-V, enables virtualization features. In addition to handling virtual machines, it is also used to implement Virtualization Based Security, which is presented in subsection 2.4.4.

---

<sup>3</sup>When referencing to these libraries, we might omit the file extension, e.g., *kernel32* instead of *Kernel32.dll*.

<sup>4</sup>These user mode implementations of API functions is why Microsoft refers to the Windows kernel as hybrid instead of monolithic.

<sup>5</sup>Microsoft provides some guidance on using it directly.

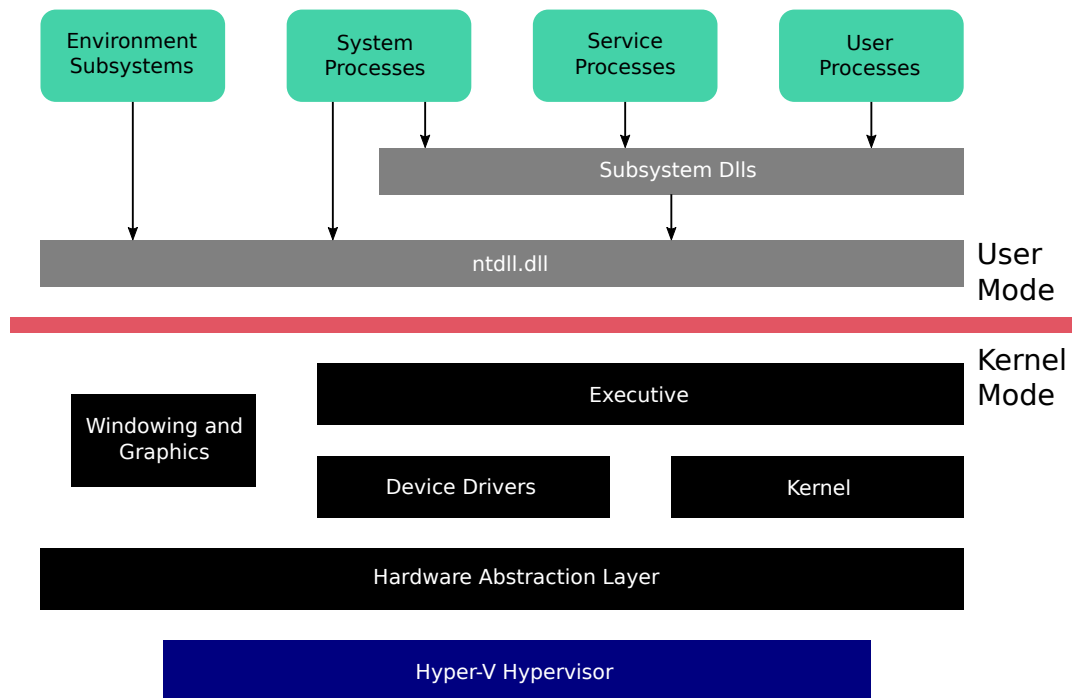


Figure 2.1: Simplified Windows architecture, most notably missing networking components and device driver layering [7].

## 2.2.2 Windows API and System Calls

The Windows *Application Programming Interface* (API) is how user mode programs can interact with the operating system. We refer to functions and their implementing modules by `module!function`, e.g., `kernel32!HeapAlloc`.

As mentioned in subsection 2.2.1, there are layers of indirection between the API functions that are documented and exposed to the user, and the final system call. As an example, let us consider the `CreateFileW` function as implemented in `kernelbase` version 10.0.19041.3758. When this function is called, it will perform various checks before entering the non-exported function `CreateFileInternal`. This function again performs various checks before calling `NtCreateFile` in `ntdll.dll`. The implementation of this function, represented at Listing 2.1, is typical of `ntdll` functions performing a system call. The first step is to save the `RCX` register in `R10`. The x64 ABI, described in section 2.3, dictates that the first argument to a function is to be passed in `RCX`, but since `syscall` overrides `RCX`, its initial value is saved and then restored after entering the kernel. Next the *system service number* (SSN) is loaded into `EAX`. The SSN is what defines with which function the kernel will continue execution. Note that the SSN is not static and can change between kernel versions. The lines 4 and 7 in Listing 2.1 check if the `SystemCall` field of the `Shared User Data` structure is set. This is the case if `Credential Guard`, a security feature relying on virtualization, is

enabled. In that case, the `int 2Eh` instruction is used instead of `syscall`, because the hypervisor can react to this instruction more efficiently [7]. Notice the `Nt` in `NtCreateFile`. Windows uses prefixes to indicate the type of API function. For example, `Nt` indicates that `NtCreateFile` is an NT<sup>6</sup> system service function.

```

1  ntdll!NtCreateFile:
2      mov     r10,rcx                ; save rcx
3      mov     eax,55h                ; set SSN
4      test    byte ptr [7FFE0308h],1 ; check if
5                                      ; KUSER_SHARED_DATA.SystemCall
6                                      ; is 1
7      jne     ntdll!NtCreateFile+0x15
8      syscall                        ; transition into kernel mode
9      ret
10 ntdll!NtCreateFile+0x15:
11     int     2Eh                    ; trap into kernel
12     ret

```

Listing 2.1: Disassembly of `ntdll!NtCreateFile` from `ntdll.dll` version 10.0.19041.3636.

Whilst the native Windows API consists of C-style functions, the *Component Object Model* (COM) was introduced to deal with the increasing amount of API functions. COM exposes new ways to interact with the Windows API, but still leverages the C-style functions under the hood. Yet another way to program for Windows is to use the *Windows Runtime* API. It is intended to be used for *Windows Apps*, such as the ones published on the Windows Store, as opposed to the traditional *Windows desktop applications*. The latter are also called *Win32 apps*. They can also use some parts of the runtime API using, in the example of C/C++, additional header files. Windows apps are installed as part of the OS or provisioned [63]. Finally, the *.NET Framework* is another way to interact with the Windows API. It provides a managed runtime called *Common Language Runtime* (CLR), which uses just in time compilation, and provides functionalities such as garbage collection.

### 2.2.3 Portable Executable

Every executable image in Windows follows the *Portable Executable* file format. This includes two different types of files. A *Dynamic Link Library* (DLL) is a collection of callable subroutines linked together as a binary file. DLLs can be dynamically loaded by applications at runtime. Even if multiple applications share the same DLL, the DLL will only be loaded once into memory. An *Executable* (EXE) is an executable image, i.e., an image that can be loaded into the virtual address space of a process and executed starting from an entry point routine.

The portable executable format is composed of two types of elements: headers and sections. Its general structure is represented in Table 2.1 [64]. These structures are defined in `winnt.h`. There

<sup>6</sup>The current architecture of the Windows kernel is called *NT*.

Element	Usage
MS-DOS Header	Used for backwards compatibility
PE File Signature	Bytes 'PE\0\0'
PE File Header	Describe content of PE file (number of sections, etc.)
PE File Optional Header	Optional in name only (image base, entry point, etc.)
PE Sections Headers	One header per section
Sections	One entry per section

Table 2.1: Simplified overview of the PE file format.

are predefined sections, such as the *.text* section containing the executable code.

#### 2.2.4 Processes and Threads

A general explanation on processes and threads is given at subsection 2.1.1. In this subsection, we introduce some elements specific to Windows. Each process in Windows has a *security context*, which describes the security identifiers and credentials of the process, the owning user, the security group it is part of, and its privileges. The context is stored in an object called *access token*. This is what makes process injection useful, as is explained in section 2.9.

In Windows, threads have two stacks. One for execution in user mode, and one for execution in kernel mode. Threads can have a secondary security context, which is not necessarily the same as the process they are part of. When the thread uses its own security descriptor, and it is different from the one of the process, it is called *impersonation*. This mechanism is often used in Windows, e.g., for client/server applications. Note that, if a thread gains access to an object this way, all the threads in the parent process will also have access to it, since the handles are per process. *Discretionary access control* uses the security context of the process requesting access to an object and compares it to the access control list associated to it.

Processes and threads, like many other structures defining system resources, are described using objects, meaning that they can only be accessed using handles. This mechanism is described in subsection 2.2.1. The *process ID* (PID) uniquely identifies a running process. Threads have their own unique identifier *thread ID* (TID).

Each process is described by its own *executive process* (EPROCESS) kernel object, whilst each thread has an *executive thread* (ETHREAD) kernel object. The *Process Environment Block* (PEB) resides in the process' user address space, and is the representation of a process in user mode. It, amongst other things, keeps track of each module loaded in the process address space.

When creating a process, their structures and context need to be set up. This is mostly handled by the *image loader*, which operates in user mode, and is implemented in ntdll. It is the first piece of



code to be run inside any user process. As one of the many tasks, the image loader parses the *import address table* (IAT) of the application, and loads all the specified DLLs, recursively handling their respective IATs. It will load kernel32 and kernelbase first, no matter if they are in any of the IATs [7]. This means that the order of DLLs loaded in a user process PEB is always ntdll, then kernel32, then kernelbase, and then the others.

### 2.2.5 Virtual memory

Memory virtualization (see subsection 2.1.2) in x64 Windows operates at 4 KB page size granularity. The full address space is of 128 TB, of which the highest addresses are assigned to the system (i.e., kernel, drivers, etc.), and accessible only from kernel mode. User process spaces grow from the low addresses. There is no mechanism in place to protect user mode memory from access from kernel mode.

## 2.3 x64 Architecture

The *x64* architecture, also called *AMD64* or *x86-64*, is prevalent on Intel and AMD consumer CPUs. The *Instruction Set Architecture* (ISA) of x64 defines, amongst others, which instructions, data types, and registers exist and how they are to be used. The x64 ISA is documented in the Intel Software Developer Manuals [29]. The *Application Binary Interface* (ABI) is described at the official Microsoft documentation [72]. This defines how two functionality layers interact at the assembly level. Of most interest to us in the ABI is the calling convention, which defines how the arguments are passed to functions.

We give a simplified overview of the Windows x64 calling convention. The first four arguments of functions are passed through registers RCX, RDX, R8, and R9. Any remaining arguments are passed on the stack, in reverse order. The first of the stack arguments will appear at `RPX+0x20`, because the first four bytes of the stack are left for the so-called *home space* or *shadow stack*. Listing 2.2 contains an example of the calling convention. The return value will be in the RAX register.

```
1 // Function signature
2 HANDLE CreateThread(
3     SECURITY_ATTRIBUTES *lpThreadAttributes, uint64 dwStackSize,
4     THREAD_START_ROUTINE *lpStartAddress, void *lpParameter,
5     uint32 dwCreationFlags, uint32 *lpThreadId);
6
7 // Usage
8 void MyStartRoutine();
9 uint32 thread_id = 0;
10 CreateThread(null, 0, MyStartRoutine, null, 0, &thread_id);
11
```

```

12 // How the arguments are passed:
13 // RCX      <- 0                (1st arg)
14 // RDX      <- 0                (2nd arg)
15 // R8       <- MyStartRoutine  (3rd arg, the address of
16                               MyStartRoutine)
17 // R9       <- 0                (4th arg)
18 // RSP+0x20 <- &thread_id      (6th arg, the address of thread_id)
19 // RSP+0x28 <- 0                (5th arg)

```

Listing 2.2: Calling convention example with the CreateThread Windows API call.

## 2.4 Mitigations

Mitigations are mechanisms put in place to improve the security of a system against attacks. Each mitigation is designed to limit a common attack vector. They cannot prevent an exploit, but aim to make it as hard as possible, without introducing a prohibitive overhead to the system. In this section, we discuss the ones implemented in Windows 10 22H2, and which are relevant to this project.

Some mitigations are enabled at the OS level, whilst others are enabled on a per-process basis, for all processes through policies, or at compilation time for the binary. This means, that some mitigations set for an image at compilation time can be overridden when creating a process running the image [65]. We do not go in depth on configuration of these mitigations in Windows, as this is a complex topic and out-of-scope of this project. We will assume that all mitigations are configured correctly. Note that this list is non-exhaustive and only contains ones relevant to our project. For a full list, consult the Microsoft documentation [60].

### 2.4.1 Data Execution Prevention

*Data Execution Prevention* (DEP) allows the operating system to set memory pages as non-executable by using the *No eXecute* bit. It is hardware enforced, and enabled by default for x64 applications [60]. This successfully mitigates standard remote code execution attacks, such as buffer overflows, relying on an executable stack. It does not prevent allocation at runtime of memory that is both writable and executable. Changing memory protection during runtime of RW pages to RWX is also still possible using documented Windows API functions.

### 2.4.2 Address Space Layout Randomization

If malware wants to exploit an existing application, it needs to know where the target lines of code are located in memory. *Address Space Layout Randomization* (ASLR) makes it harder to find

these addresses. It randomizes where data is placed in memory, and is implemented for both the kernel and user address spaces. This technique is important to understand for any application that manually resolves addresses as runtime, for example with the intent of hooking a function.

ASLR support needs to be enabled in the image PE header, and a relocation table needs to be present if the program is not position independent. We note that executable images in Windows are typically not position independent. When an executable is launched, DLLs it depends on, and which fulfill these conditions will be processed by ASLR [7]. For each, an image offset is chosen and applied. This offset is selected from 256 possible values generated at boot, each 64 KB aligned. The base address of the executable itself is randomized each time it is loaded. A pseudorandomly generated 8-bit, 64 KB aligned, offset is added to the executable's preferred load address.

DLL randomization is only performed once per DLL to minimize the overhead incurred by image relocation. The memory manager keeps track of the offsets for each of them. This is done this way to still enable code-sharing of DLLs across processes, i.e., each unique DLL is still present at most once in memory. The stack base and each heap allocation are randomized as well, once again using a pseudorandomly generated offset.

### 2.4.3 Control Flow Integrity

*Control Flow Integrity* (CFI) is designed to prevent abuse of indirect calls. It ensures that execution of a program does not deviate from the valid control-flow defined at compile time. An attacker should not be able to redirect the control-flow to a location that is not the target of an indirect call when the program was initially compiled. A generalized implementation is to build a valid control-flow graph using static analysis on the source code. The precision of this analysis is crucial; either it is too permissive and there are too many targets in the set that should not be there, or it is too restrictive, and the program can break when executed. To enforce CFI, there are checks during run time at each occurrence of an indirect call. The target of the call is verified to be inside the target set, and if it is not, the program gets terminated.

The Windows implementation of CFI is called *Control Flow Guard* (CFG). Applications compiled with the appropriate flag will advertise this in their PE header under *DLL characteristics* as supporting *Control Flow Guard*. Windows builds one set of valid targets for the entire program. This is less secure, but offers better compatibility, than constructing one target set per function prototype. Indeed, with the former approach, it is possible to redirect execution to a function in the valid target set, but that was never intended to be a target of the function prototype were the address was overwritten. The valid targets are stored in a `.gfids` PE section, which is merged by default in the `.rdata` section. A bitmap enables constant time lookups, how exactly this is done is described in Windows Internals [7]. Note that, if executable memory is allocated in a process with CFG enabled, all the addresses inside the newly allocated memory will be set as valid targets. The same is true when protection is changed to executable.

#### 2.4.4 Virtualization Based Security

Windows utilizes the Hyper-V hypervisor to implement various security mitigations. These mitigations form what is called *Virtualization Based Security* (VBS), or *Virtual Secure Mode* (VSM). This introduces a new security boundary, orthogonal to the user mode and kernel mode separation, called *Virtual Trust Levels* (VTL). VBS adds security mitigations in case of a kernel compromise.

As represented in Figure 2.2, there are two VTLs, where VTL 1 has higher privileges than VTL 0. The VTL 0 side operates as described in subsection 2.2.1, the difference lies in VTL 1. The kernel part of this trust level is called *Secure Kernel*, and the user part *Isolated user Mode* (IUM). They run in ring 0 and 3 respectively. The secure kernel is implemented in `securekernel.exe` (as opposed to `ntoskrnl.exe`). IUM provides `iumdll.dll`, which is a VTL 1 version of `ntdll.dll` exposing additional system calls only available in this trust level. A VTL 1 counterpart of `kernelbase.dll` is provided under `iumbase.dll`. Only *trustlets* are allowed to run in IUM. If a Windows API function is not exported by one of the IUM specific subsystems, it is marshalled and then executed by the kernel in VTL 0 [54]. VBS is enabled by default since Windows 10 version 1607 [7].

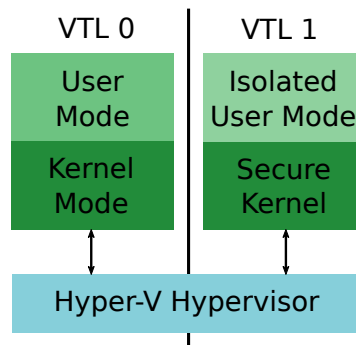


Figure 2.2: High-level overview of the VBS architecture.

## 2.5 Detection

Mitigations are designed to limit attack vectors, but they cannot fully remove them, hence the need for detection. Ideally, an attack should be detected and prevented before it could successfully infect the target system. For the case where this does not work, the system should be monitored in real-time and any suspicious activity should be detected, analyzed, and measures should be taken if necessary. We distinguish two types of detection solutions: AVs (subsection 2.5.2) and EDRs (subsection 2.5.3).

### 2.5.1 Windows Native Telemetry Sources

Before we discuss detection solutions, we will present the functionality integrated into Windows to provide telemetry on the system, and available to developers building detection software. *Event Tracing for Windows* (ETW) is the main facility providing applications (both in user and kernel mode) the ability to consume, and manage log and trace events [1]. ETW is mostly implemented in the kernel, and can be controlled using three types of components:

1. A *controller* starts and stops event tracing, and manages which providers are used.
2. Each *provider* contains event tracing instrumentation, and defines which events it can produce. A provider is identified by a unique ID *GUID*.
3. The *consumer* is an application that selects from which providers it wants to read the trace data, and from which storage source.

Many providers come preinstalled, and are actively used by various OS components [1]. The link between providers and consumers is called a *session* (or *logger instance*). It defines which events from providers should be consumed, and how they should be processed.

Parts of ETW is implemented in user mode, which opens it up to attacks [8, 91]. To mitigate this, Microsoft introduced the *Even Tracing for Windows Threat Intelligence* (ETW-TI) provider for usage in security relevant event logging. The events logged by this provider are listed in Appendix A. This provider is implemented inside the kernel [35], meaning that it cannot be tampered with from userspace.

### 2.5.2 Antivirus

An *antivirus* program is designed to detect malicious software on a device. It typically uses static analysis of files and memory to detect specific artifacts, such as hashes or strings. *Microsoft Defender Antivirus* is installed by default with the Windows operating system. We demonstrate detection based on static signatures using the ThreatCheck tool [79] in Appendix B. An antivirus could also detect behavior statically. For example, Mandiant’s tool CAPA [42] uses pre-defined rules to identify capabilities of a program. In Appendix C, we give an example of the kind of capabilities that can be detected.

The biggest limitation with antivirus software, is that it typically relies on static analysis alone. This implies that many obfuscation methods, such as encryption and name hashing will prevent it from detecting malicious capabilities inside executables that are not known to be malicious. Additionally, since it does not monitor the system itself, an antivirus cannot follow an incident through time. It does not collect data, nor does it find relationships between incidents. This kind of

detection approach can only identify known malware, but it typically has a very low false positive rate.

### 2.5.3 Endpoint Detection and Response

The *Endpoint Detection and Response* (EDR) approach complements antivirus software. Whilst antivirus software performs static analysis on files, or parts of virtual memory, at a single point in time, EDR systems aim to monitor behavior on and across endpoints over time. They are built using different components [22], whose relationship is represented in Figure 2.3:

1. The *agent* application is deployed on each endpoint. It controls and consumes *telemetry* from sensors, and performs basic analysis locally. It sends the gathered data to the backend collection system.
2. The *backend collection system* aggregates all the telemetry and analyzes it. In depth dynamic analysis of potentially malicious files is performed there as well.
3. *Sensors* available on endpoints, either as native operating system functionality, or installed by the EDR itself, sits inline of system processes and gathers data. Some sensors are tasked to monitor network traffic.
4. *Detections* are the logic implemented in the server that correlates telemetry gathered from one or multiple endpoints. Machine learning can be used to detect previously unknown malicious activity. Sandboxes can perform dynamic analysis on suspicious files.

At the same time, agents perform the antivirus tasks described in subsection 2.5.2. This combination of *brittle* and *robust* detection techniques helps to balance false positives and false negatives.

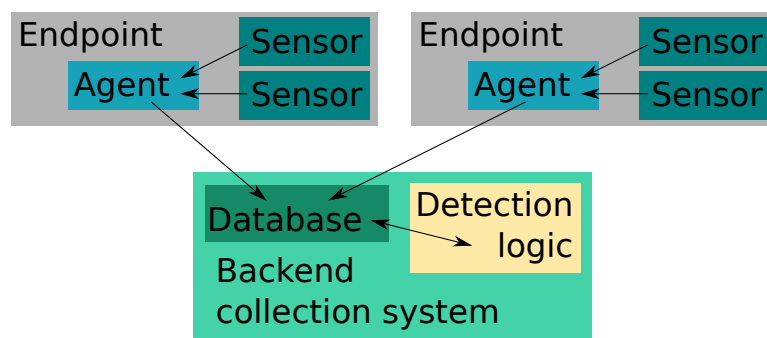


Figure 2.3: High-level overview of an EDR system. The arrows represent data flow.

Telemetry sources be classified into different categories [22].

1. Static scanning of files and virtual memory

2. Hooks placed in userspace DLLs loaded in memory
3. Operating system native functionality, such as ETW (see subsection 2.5.1)
4. Kernel mode drivers
5. Network filter drivers collecting network data

Any activity that is considered as suspicious by the EDR will contribute to a *risk score* [22], also called *detection score*. Once this score reaches a certain threshold, an alert will be raised. This implies that many small events can lead to a detection, but also one single very suspicious event, such as a known malware binary making its way on disk.

Because rules can only be written for threats that are known, machine learning is getting increasingly popular to detect previously unknown threats [22]. Training such a model is non-trivial. It should be able to differentiate between normal system activity and abnormal events, whilst minimizing false positives. This poses issues such as, for example, how to avoid false positives on a software that got updated with new, legitimate, functionality? Because of this, custom and specific rules still have their place in modern detection systems. They can be fine-tuned to minimize false positives, whilst also detecting common threats reliably [22]. For example, a signature-based rule triggering on the precompiled Mimikatz [14] binaries will already detect many unsophisticated threat actors.

## 2.6 Microsoft Defender XDR

During this project, we interact with *Microsoft Defender XDR*. Whilst EDR stands for Endpoint Detection and Response, XDR stands for *Extended Detection and Response*. Defender XDR aims to ingest telemetry not only from endpoints, but also from other Microsoft services such as Azure [48]. This means that it also integrates the EDR *Microsoft Defender for Endpoint* [57]. The *Microsoft Defender Portal* [58] is where all of this information is gathered and presented to analysts. *Microsoft Defender Antivirus* [56], the antivirus integrated into the Windows OS, is leveraged to provide signature based detection. It offers real-time protection using local, signature-based scanning.

Defender XDR is a proprietary solution, meaning that we have to rely on the official documentation [57], and third party reverse engineering projects. We will require some assumptions on its internals during this project, which we present here, as well as our reasoning behind them. Alerts are raised once the risk score passes a certain threshold. The main sensor used is `MsSense.exe` [57]. The telemetry sources used by the agent are: static scanning of files and virtual memory, ETW, kernel mode drivers, and network filters. There are no userspace hooks. We verified this on `ntdll` using our own tool, which can be found in this project's repository. As a leading EDR with intimate knowledge of the Windows internals, since both are developed by Microsoft, sensitive Windows API calls are monitored, detection rules tailored to the OS are implemented. A machine learning model is used

to uncover unknown threats in the cloud-based detection logic [82]. Dynamic analysis of files is performed in a sandbox, both locally as evident by delay in execution of unknown executables, and in the cloud [57].

## 2.7 Evasion Techniques

This project is not about evasion techniques. Nonetheless, as explained in chapter 5, we need to utilize some basic evasion techniques. A well-known technique in offensive development is the so-called *direct syscalls* technique. It is used to avoid triggering hooked functions (read subsection 2.5.3) in `ntdll`. This technique simply resolves a syscall's SSN by parsing `NTDLL` at runtime, and then performs the system call manually following the convention described at Listing 2.1. This basic variant is also known as HellsGate [39]. There are known improvements on this technique, such as Recycled Gate [94], which does not perform the system call manually, but sets up the stack and jumps to the location in `ntdll` just before the `syscall` instruction.

Static detection utilizes signatures, such as hashes, to recognize malware. To avoid detection based on these, the *name hashing* technique can be used. Instead of the binary containing function or library names as strings, they are instead hashed before compilation. The strings they are compared to are themselves hashed at run time. Another aspect of static detection are signatures on shellcodes, i.e., compiled code. Instead of re-engineering payloads each time they are detected due to signatures, it is more convenient to encrypt them before compilation, and decrypt them dynamically at runtime.

## 2.8 Memory Enclaves

Operating systems enable isolation between processes (section 2.1). If some malicious code executes inside a process, for example through process injection, it will have access to this entire process's address space. The same is true if a malicious driver is installed on the system; since drivers run in kernel mode, they have access to the entire user mode address space. This can allow an attacker to obtain sensitive data. *Memory enclaves* are designed to mitigate this risk by introducing intra-address space isolation. The memory inside an enclave is inaccessible to the outside process memory and from the kernel, but the enclave retains full access to the host process's memory. A simplified overview of the memory enclaves concept is shown in Figure 2.4. Memory enclaves are an implementation of the *Trusted Execution Environment* (TEE) concept. The Windows operating system provides native support for two memory enclave technologies [1, 7]:

- Virtualization based security enclaves: utilize VBS, introduced in subsection 2.4.4, to provide hypervisor enforced memory enclaves



- *Intel Software Guard Extensions (SGX)*: proprietary hardware enforced memory enclaves for Intel CPUs, requires a 6th generation or later Intel CPU, and is supported since Windows 10 version 1511

The SGX ecosystem is complex and contains many aspects. We only present the basics required to understand our process injection attempt presented at section 3.4. Relevant resources for further reading on SGX are [12, 29, 32].

SGX memory enclaves reside entirely in DRAM. A special region of memory, called *Processor Reserved Memory* (PRM), that cannot be directly accessed by other software is used to store SGX enclaves. They are located in the *Enclave Page Cache* (EPC), which is a subset of PRM [12]. To support enclaves, the Windows OS enumerates these pages at boot, and sets a specific bit in their respective VADs to mark them as such [7]. Since enclaves provide attestation, i.e., they can provide a cryptographic proof of their initial state, there needs to be a root of trust. The root of trust is an Intel implemented and provided SGX enclave called *Launch Enclave* (LE) [12]. On Windows, this enclave is started at boot and hosted by the `aesm_service.exe` service [7].

The life cycle of memory enclaves is as follows:

1. Create the enclave: the OS allocates pages in the EPC for the enclave and its control structures
2. Load data into the enclave: copy the code and data of the enclave into its memory pages
3. Initialization: verification of the loaded data with respect to the expected initial state, and verification of the enclave hash
4. Enclave code execution
5. Enclave deletion

The enclave content is readable from the hosting process until initialization, but it will be encrypted afterwards whenever the current execution context is not inside the enclave. To allow verification of the enclave, i.e., that it is indeed running the expected code, a cryptographic hash is built recording the entire construction and loading steps. An enclave hash, signed by its developer, is provided before enclave creation. Another hash is computed during every creation and initialization, and compared with the computed one at the initialization step by the LE. Additionally, the hash signature is verified by the LE to have been correctly signed using a key whitelisted by Intel [30], if it is not an Intel enclave. If the hash value is not as expected, or if the signature is not correct, then the enclave will not be initialized. The LE is verified by the processor itself, using a hard-coded signing key provisioned by Intel during manufacturing. The CPU will only execute enclaves that have been initialized by the LE beforehand [12].

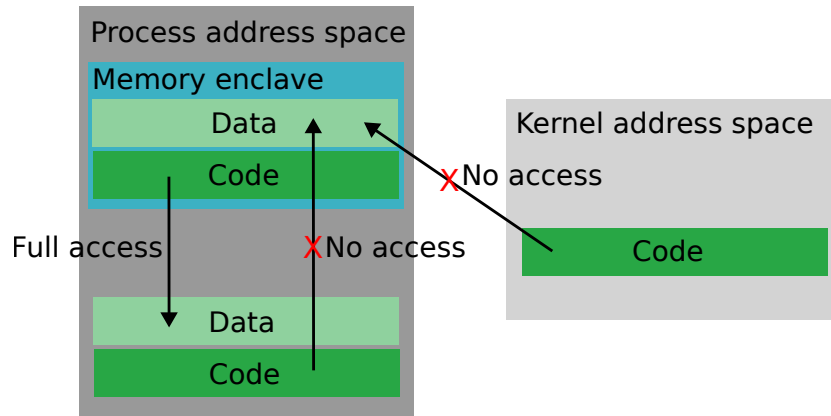


Figure 2.4: Simplified overview of the memory enclave concept.

Instruction	Usage	Can target an uninitialized enclave
<code>ecreate</code>	Create the structures required to manage an enclave	Yes
<code>eadd</code>	Add a page to an uninitialized enclave	Yes
<code>eremove</code>	Remove a page from an enclave	Yes
<code>einit</code>	Initialize an enclave	Yes
<code>eenter</code>	Enter an enclave	No
<code>eresume</code>	Re-enter an enclave	No
<code>eexit</code>	Exit an enclave	Yes
<code>edbgrd</code>	Read from an enclave in debug mode	Yes
<code>ewb</code>	Copy a page from the enclave to main memory	Yes

Table 2.2: Project relevant x64 ISA related to SGX.

SGX enclaves can operate in debug mode. This breaks the security guarantees provided by enclaves, because their content can then be read using the Intel SGX debugger [12]. Debug enclaves will be initialized by the LE, even if their hash has been signed by a non-whitelisted key [31].

Software Guard Extensions add new instructions to the x64 ISA. They are documented by Intel in the Software Developer Manual volume 3D [29]. The instructions relevant to this project are presented in Table 2.2.

## 2.9 Process Injection

During threat simulation exercises, *process injection* can be used to execute malware payloads. The primary intent behind it is to execute the payload inside the context of a benign process in order to avoid process-based defenses. Outgoing TCP connections are, for example, less suspicious when

they come from the same process as a browser , as opposed to a previously unknown application freshly downloaded from the internet. Additionally, the target process will still be backed by legitimate files on disk, since the payload resides only in its memory. Another property of process injection is that it can be used to elevate privileges. A process holding a token granting access to some privileged resource can be targeted and subsequently impersonated, since every operation performed by a process executes with the security context of this process. The concept and effects of process injection is represented at Figure 2.5.

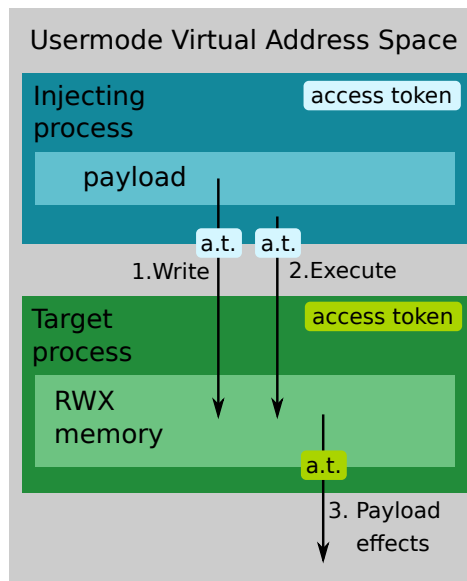


Figure 2.5: The process injection concept and how it affects the system.

Process injection involves two live (i.e., running) processes: the *injecting process* and the *target process*. The injecting process is controlled by the adversary and the target process is a legitimate and benign process already running on the system. An adversary with arbitrary code execution capabilities starts the injecting process by using a malware program we call the *injector*. Note that both processes run on the same system, and in the same user address space. Remote code injection, such as SQL injections in a web application, are not considered process injection, since in that case the injecting process runs on a different system. The injector executes the three process injection primitives in order:

1. Allocation: allocate memory inside the target process
2. Writing: write the payload to the allocated memory
3. Execution: trigger execution of the payload

We note that allocation is not needed in some techniques where memory already allocated in the target process address space is reused. As explained in section 2.1, not every location in memory

is writable and/or executable. Managing these memory permissions can be done either at the allocation, and/or at the execution steps. A generalized process injection approach is represented in Figure 2.6. In general, process injection techniques do not rely on vulnerabilities, but abuse legitimate operating system functionality and API functions. Under Windows, this means that the simplest form of process injection, which we reference as *classic process injection*, uses the following API calls:

1. `Kernel32!OpenProcess` to open a handle to the target process with the required privileges.
2. `Kernel32!VirtualAllocEx` to allocate RWX memory in the target process.
3. `Kernel32!WriteProcessMemory` to write the payload into the allocated RWX memory.
4. `Kernel32!CreateRemoteThread` to create a new thread in the target process and execute the payload in it.

Detection software can detect process injection at three points:

1. At initial infection: if the injector is placed onto disk, static or dynamic analysis can be performed on it. Process injection capabilities can be detected, or if it is a known binary it might be signed.
2. During execution: once the injector has been launched, it will make various API calls during the injection. These can be monitored, and a chain of calls, such as the ones used by classic process injection, can indicate an ongoing process injection.
3. When the payload executes: process injection is not the final attack, it serves to deploy a malicious payload. This payload activity might trigger a detection. An EDR system can then perform automated analysis of the process in which the payload was executed, and utilize the events logged previously. Then, it can make a link back to the injector binary.

What makes code injection difficult to detect, is that there are legitimate use cases for it. For example, debugging, diagnostics, and live patching all leverage code injection. This means that the false positives and false negatives have to be balanced out for the system to still be usable.

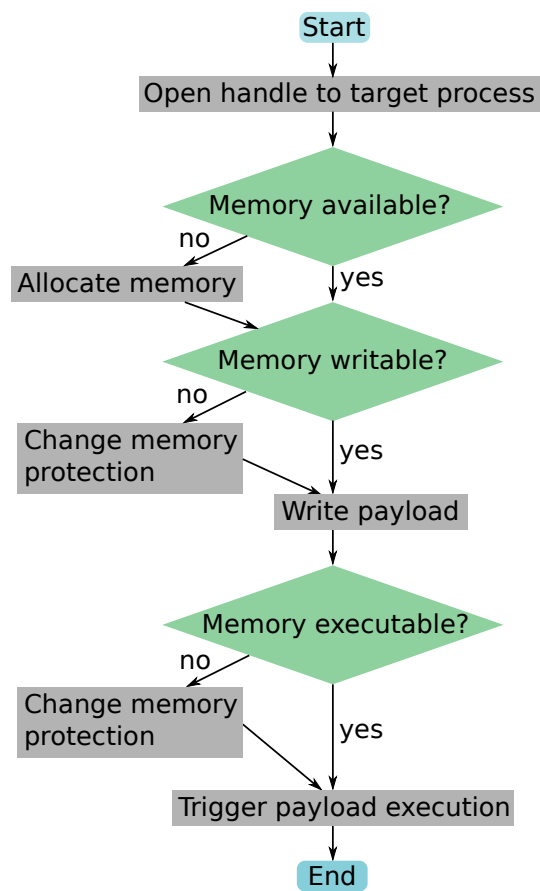


Figure 2.6: Execution flow of generalized process injection.

## Chapter 3

# Design

Malware has to be designed with respect to the target operating system, CPU architecture, installed detection software, and intended outcome. Process injection is no different in this regard; constraints and objectives need to be defined beforehand. The most influential constraints are the target platform and detection system. For example, due to the differences in architecture between the two operating systems, process injection designed to target Linux will be implemented differently to one designed to target Windows. In the same manner, the security solution installed on the target system will dictate which injection technique can be used. Indeed, if a signature-based antivirus is the only detection solution in place, then even simple techniques will be successful, as long as the payload or the compiled injector are not signed. In contrast, endpoint detection and response systems make the task harder by monitoring behavior indicating ongoing process injection.

We first present general design constraints related to process injection in section 3.1, and then our decisions for this project in general in section 3.2. To get a better understanding of real-world applications of process injection, we took an existing implementation, namely Threadless Injection by Ceri Coburn [10], and improved upon it in both functionality and stealthiness. The design decisions taken in this context are presented in section 3.3. We then took the lessons learned, and tried to develop our own technique based on memory enclaves, which is discussed in section 3.4.

### 3.1 Design Constraints

We now formulate the constraints that dictate the design decisions that have to be taken when working on process injection. These are related to the threat model, technical constraints, requirements on functionality, and operational security concerns. The requirements on the process injection technique in the context of this project is summarized in Table 3.1. We discuss them in detail below.

In our threat model, we assume an adversary capable of arbitrary code execution on the target system. They do not have administrative privileges, and have to operate with detection measures in place. Tampering with the detection measures is considered as impossible. The adversary has no knowledge of unpatched, zero-day vulnerabilities. They are familiar with the target system, i.e., they know which software is usually running. They know which security measures, both mitigations and detection systems, are in place. Potential detection solutions and mitigations are assumed to be configured correctly.

Technical constraints on the injection technique are defined by the target system architecture, operating system, mitigations, and security solution. In this project, we target the Windows 10 22H2 10.0.19045 OS running on a x64 CPU. The injector must be able to cope with ASLR, DEP, and CFG mitigations on the target process. There are no OS-level non-default mitigations enabled on the target systems. The target system is protected by Microsoft Defender XDR, and runs a clean installation of Windows. The technique must not rely on software or configuration that might not always be present.

The injector must be reliable, both in functionality and towards detection, i.e., it is expected to always function correctly under the technical constraints defined above. If a detection occurs, it must not be because of the injection. Detection due to the payload is considered out of scope. Any position independent shellcode payload must be injectable, i.e., arbitrary code that might not even terminate must be usable. Persistence is out of scope of the injector, and if desired should be implemented in the payload. Finally, any userspace process with integrity level medium must be targetable.

*Operational security* (OPSEC) is a military term, whose primary meaning is the intent of reducing information disclosure towards the adversary as much as possible. In the context of red teaming operations, this becomes the concept of executing code to minimize the risk of detection by a security solution. Additionally, any exploits and malware should leave as little traces as possible on the system, to make it harder for forensic analysts to discover the origin of the attack. These traces are called *indicators of compromise* (IOC). Process injection must be non-disruptive, i.e., it must be transparent to the user. The target process must execute as expected, without crashing. We aim to not create unnecessary IOCs but don't intend to make the technique as stealthy as possible in this project. The idea is that the process injection technique can stand on its own, and does not require advanced evasion and stealthiness techniques to remain undetected. This balance is hard to find, as will be discussed in more detail in chapter 5. For example, it is necessary to implement some basic obfuscation techniques, such as payload encryption and name hashing. Otherwise, detection is trivial with known payloads and the injector will be detected as malicious even before the injection occurs. Finally, the injection should not cause any unintended side effects on the systems as a whole. Its stability and functionality should not be affected. Only the targeted process must be injected into, and only once, i.e., the payload should execute exactly once.

Kind	Constraint	Target
Threat model	Adversary capabilities	Arbitrary code execution in userspace
	Adversary privileges	Non-privileged
	Adversary knowledge	Full knowledge of the system configuration and installed software
Technical	CPU architecture	x64
	Operating system	Windows 10 22H2 10.0.19045
	Detection	Microsoft Defender XDR
	OS mitigations	None
	Process mitigations	ASLR, DEP, CFG
	Non-native OS requirements	None
Functionality	Disable detection	No
	Target process	Any process of integrity level medium
	Payload	Arbitrary position independent shellcode
	Persistence	Has to be implemented in the payload
OPSEC	Effect on target process	Target process functionality and stability is unaffected
	Indicators of compromise	Best effort
	Side effects	None

Table 3.1: Process injection design constraints and the respective targets of this project.



## 3.2 General Design Decisions

Designing malware to remain undetected in the presence of an arbitrary advanced security solution is non-trivial. The implementations of each component, as introduced in subsection 2.5.3, of different EDRs can vary despite their concept and design being the same at a high level. For example, the detection logic is a primordial component of each EDR, and typically proprietary<sup>1</sup>. Not every EDR solution will trigger in the presence of the same patterns. Whilst usually the same sensors are used by most products, such as usermode hooks and ETW, they might differ in their configuration. This implies that the chosen technique has to be tailored to evade the target EDR, either by using other APIs, or a completely different technique.

There are general design decisions that can be taken, e.g., the payload should always be encrypted to avoid signature-based detection, but others have to be decided on a case-by-case basis. In the context of red teaming assignments, this means that the choice of technique, API calls, how the API is to be called, and so on, need to be reevaluated for each assignment, i.e., for each target system. These decisions are non-trivial; should `syscall` instructions be performed directly to avoid hooks? Should we use the API provided by subsystems? Or is it better to use the underlying functions in `Ntdll.dll` directly? To follow these requirements as well as possible, iterative development is done. We build starting from the minimum working example and test it against the target detection solution, and improve the program iteratively until the predefined requirements are satisfied. We discuss this process and its challenges in more detail in chapter 5. The general approach we took is to follow the KISS principle as much as possible, and deviate from it only if necessary to add additional evasion methods.

To minimize the engineering effort required for each assignment, the injector should not rely on specific exploits, but rather abuse existing functionality. Exploits might be very powerful, and even necessary in highly protected environments, but relying on them is counterproductive for ethical hackers. Indeed, the ethical hacker should perform responsible disclosure towards the manufacturer and encourage a fix of the vulnerability, rather than keeping the vulnerability secret. This is in the interest of their clients. Indeed, the security of the system should not depend on the secrecy of its implementation [85]; if a vulnerability is kept secret there is no guarantee that a malicious actor is not exploiting the same vulnerability, making everyone that is affected insecure. Since the intent behind red teaming exercises is to improve the security stance of the customer, it is therefore counterproductive to withhold information about vulnerabilities. This implies that any vulnerability found when preparing for a threat simulation assignment should be disclosed afterwards at the latest, and a new vulnerability has to be searched for when preparing the next one. Such a vulnerability is not trivial to find, making the research cost prohibitive. In comparison, abuses of intended functionality will stay available longer, maybe for the lifetime of the product, since it is closely linked with its intended use case. These are typically mistakes in the design of an API or protocol that can be abused to cause undesirable effects, for example misconfigurations in

---

<sup>1</sup>A notable exception is Elastic, whose detection rules are published on GitHub [15, 16].

Microsoft Active Directory [73]. Additionally, the process injection technique should use the least amount of undocumented functionality possible, since these are subject to change.

Windows malware can be written using many programming languages, the ones we encountered the most during when performing research for this project are C, C++, C#, Go, Rust, and Nim. The biggest advantage of using less popular languages such as Go, Rust, and Nim, is that they are less signatored, and harder to reverse engineer by forensic analysts. Using C# gives access to popular tooling such as OffensiveCSharp [23] and SharpSploit [9], the latter makes use of the .NET framework [62]. Both the *Component Object Model* (COM) and .NET can be used to write malware. We chose not to leverage these technologies to avoid having to learn them. Notably COM is notoriously complex and has a high barrier of entry, as is evidenced by James Forshaw in his *COM in Sixty Seconds* talk [18]. We chose to use C in this project for the following reasons:

1. The native Windows API is written in C
2. It is easier to interact with the kernel's internal structures using C
3. The .NET runtime might not be available on the target system
4. COM and .NET use the native Windows API under the hood anyway
5. C is the lowest level and most portable, if the technique can be written in C, it should be portable to the other languages

### 3.3 Threadless Injection

Threadless Injection [10] was introduced by Cedri Coburn at the Bsides Cymru 2023 conference. Detection of process injection is often based on the occurrence of a combination of Windows API calls that follow the schema of classic process injection. The idea of Threadless Injection originates from the following assumption: an EDR would detect the chain of `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread` API calls used in classic process injection. Allocation and memory writing is not straight forward to replace; change the execution primitive and see the impact on detection.

This technique uses function hooking to trigger payload execution. It utilizes memory holes left by the Windows loader between DLLs to allocate space for the loader and payload, and writes it there. Then, a chosen function entry point of an arbitrary DLL loaded in the target process' address space is hooked to redirect execution towards the loader. The loader restores the hooked function, redirects execution into the payload, and finally resumes the original function call. This procedure is visualized in Figure 3.1.

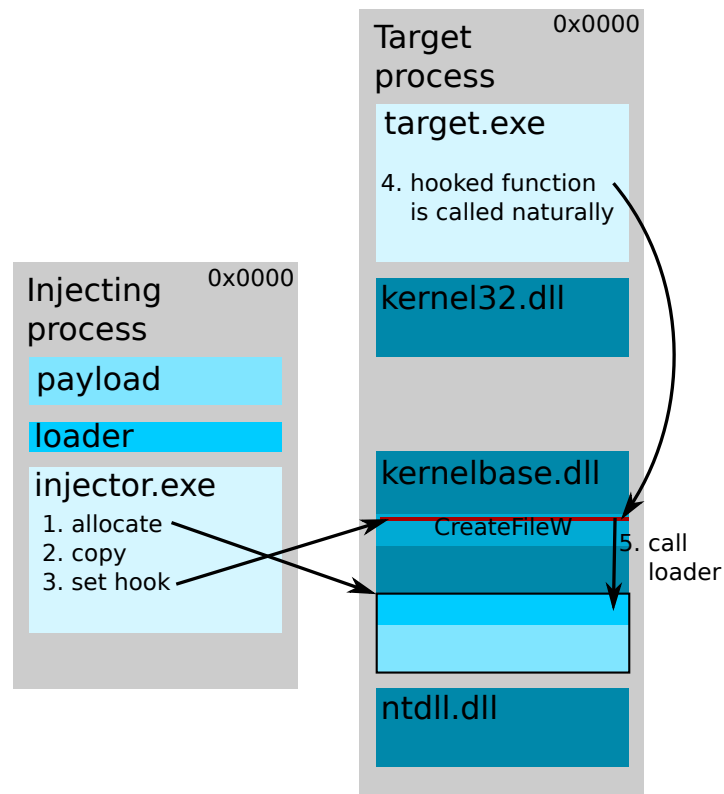


Figure 3.1: High-level overview of the Threadless Injection technique.

We chose to improve upon this technique because it has been published relatively recently, i.e., in February 2023. Novelty is a useful property for injection techniques, because if it is different enough for its pattern of operation to not be monitored by detection solutions, then it can execute covertly. Additionally, it is essentially derived from the classic injection technique, which makes it interesting as a starting point to get hands-on with process injection.

We now discuss the design of this method, and the improvements we apply to it, by decomposing it into the three primitives. A comparative table of our improvements, with respect to the targets formulated in Table 3.1, is presented in Table 3.2.

### 3.3.1 Allocation

When loading modules into a process, the Windows loader leaves gaps in memory, which we call *memory holes*. Due to limitations of the execution primitive, as explained in subsection 3.3.3, the loader shellcode has to be within 2 GB of the hooked function call. This makes the memory holes between DLLs ideal to place the shellcodes, since they can be found within these restrictions around the DLL the hooked function is located at. The memory is allocated with RX permissions initially,

since this is the protection expected of loaded DLLs.

In the original implementation, the way the target DLL is retrieved means that only the ones already loaded in the injecting process' address space, or that are in the known DLL set, can be targeted. We found this to be a limiting factor of the design, since it would not allow placing hooks in proprietary modules of software such as web browser. This would be a desirable property, because it allows to target the exact moment of execution of the payload.

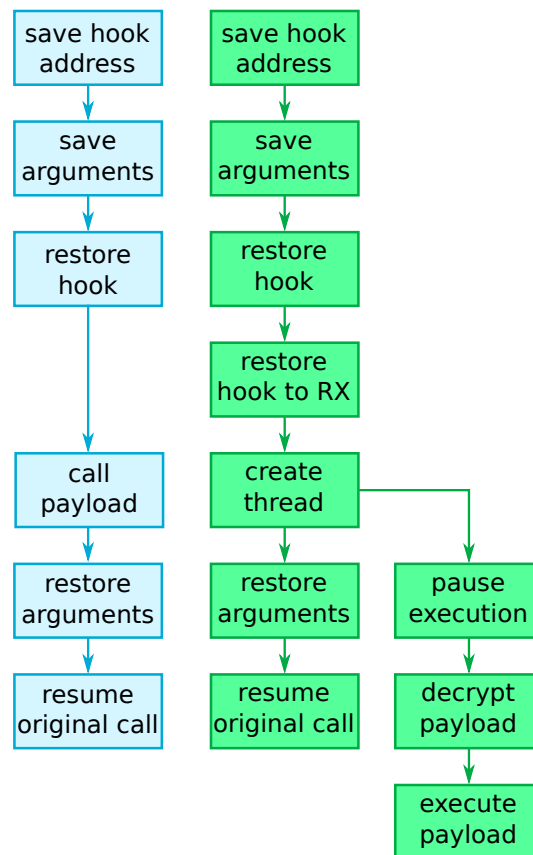


Figure 3.2: Functionality of the shellcodes for threadless injection, the original version is in blue and our improved version is in green.

### 3.3.2 Writing

To write the loader shellcode and payload to the memory hole, the classic writing primitive is used, i.e., a combination of `VirtualProtectEx` and `WriteProcessMemory`.

Constraint	Original	Custom
Adversary capabilities	Compatible	Compatible
Adversary privileges	Compatible	Compatible
Adversary knowledge	–	–
CPU architecture	Compatible	Compatible
Operating system	Compatible	Compatible
Detection	Detected	Undetected
OS mitigations	Not affected	Not affected
Process mitigations	Not affected	Not affected
Non-native OS requirements	Compatible	Compatible
Disable detection	–	–
Target process	Compatible	Compatible
Payload	No, has to terminate	Yes
Persistence	–	–
Effect on target process	–	–
Indicators of compromise	–	–
Side effects	Compatible	Compatible

Table 3.2: Comparative table of functionality targets, as formulated in Table 3.1, of the original threadless injection technique, and our custom one.

### 3.3.3 Execution

To hook the API function, the first instruction of the function is replaced by a `CALL` instruction with the loader shellcode as target. For this to work, the loader has to be within 2 GB of the invoking function since the `CALL` procedure supports a 32-bit signed offset, which can represent an offset of  $\pm 2$  GB.

The original loader shellcode restores the hooked function as a first task, to minimize the risk of it triggering more than once. It then executes the payload, before redirecting execution to the hooked function. This means that, if the payload does not return, then the hooked function will not be called, and the target process will hang. To avoid this, we modify the loader shellcode such that the payload executes in a different thread, allowing the target process to resume even if the payload does not return. To avoid signature detection on the payload, we also add a decryption routine to the loader. A simple sleep before the decryption acts as a basic sandbox evasion, i.e., it prevents detection of a signatured payload during dynamic analysis.

The hook location memory permissions are set to `RWX` for this to work, since the hook has to execute, but it also needs to be restored after it has been called once. To avoid the unnecessary artefact of `RWX` memory, we add a call to `VirtualProtect` in our shellcode, setting the protection back to `RX` after the hook executed. The comparison in functionality between the shellcodes is represented in Figure 3.2.

### 3.4 Injection using Memory Enclaves

As mentioned in section 2.8, Windows provides a native API to interact with SGX enclaves. This API is defined in `enclaveapi.h` and contains functions for each stage of an enclave's life cycle [47]:

1. `CreateEnclave` for enclave creation
2. `LoadEnclaveData` for loading data into the enclave
3. `InitializeEnclave` for initializing the enclave
4. `CallEnclave` for transferring execution into the enclave
5. `TerminateEnclave` and `DeleteEnclave` for stopping and deleting an enclave

What makes this API interesting to us is that both `CreateEnclave` and `LoadEnclaveData` support targeting a remote process. Similarly to `VirtualAllocEx` and `WriteProcessMemory`, these functions can be used to straightforwardly inject memory into any process to which we can open a handle. The inherent detection evasion during payload execution provided by the enclave, since its content cannot be accessed, even by the kernel, is another reason why this technology caught our interest for process injection. Another benefit is that, despite the API being documented, we could not find any working example of an enclave created and started using it. This makes it improbable, in our opinion, that its usage is actively monitored by security solutions. Note that memory enclaves only provide evasion during runtime. The payload still has to be encrypted during injection.

#### 3.4.1 Payload Execution in Memory Enclaves

Concerns about the ability of memory enclaves to hide code from security solutions have been formulated as early as 2013 [83, 84]. Although one notable academic malware leveraging SGX has been published [86], it has been argued that *trusted execution environments* (TEE), such as SGX, do not provide any practical advantage to malware developers [38]. Indeed, computation inside memory enclaves has to follow certain constraints. Functionality can be implemented allowing enclaves to work around these constraints, but this implies that the execution context will switch to outside the enclave, and can hence be monitored by a security solution.

One such constraint is that code running inside SGX enclaves cannot execute instructions that could lead to `VMEXIT`, that are related to I/O, or that may require a change in privilege level (i.e., from usermode into kernelmode) [32]. The latter implies that the `syscall` instruction cannot be used inside an enclave, and hence most Windows API functions cannot be used. This imposes severe limitations on a payload executing inside an enclave. This is not compatible with our design target of supporting arbitrary payloads (Table 3.1).

Our first idea was to utilize the Windows enclave API to create an enclave in the target process, and execute the payload directly inside the enclave. This approach is represented in Figure 3.3. As we already discussed in this section, the restriction this would impose on the payload is too high for real-world usage. Windows does not provide any documented subsystem for its API that could be called from an enclave, nor any undocumented one we are aware of. Such a subsystem would be necessary to compensate for the missing ability to perform `syscalls` from inside the enclave.

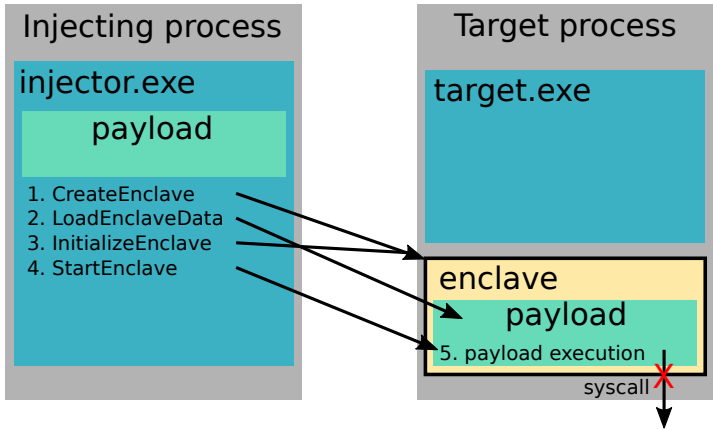


Figure 3.3: Memory injection technique using memory enclaves with the payload executing directly inside the enclave. System calls cannot be performed by the payload.

There are existing projects providing a compatibility layer between the enclave and the OS API [2, 4, 21], but these would require installation on the target device, which we specified in our constraints as undesired (Table 3.1). Such a daemon service has to execute in the context of the target process, requiring another process injection technique. This makes this approach unusable for process injection anyway, since the latter technique might as well be used standalone in this case. Process injection using such a compatibility layer is shown in Figure 3.4.

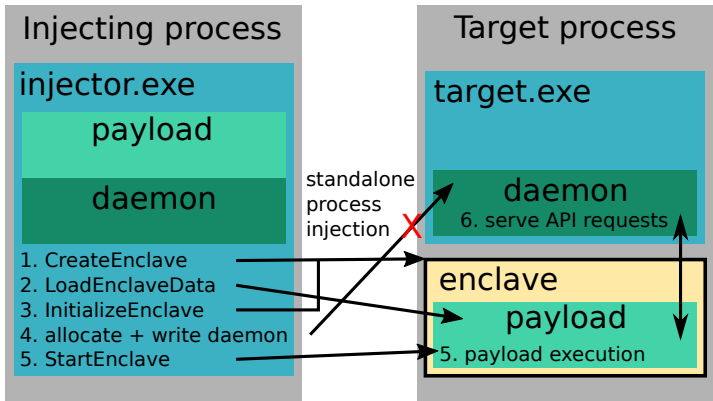


Figure 3.4: Memory injection technique using memory enclaves with the payload executing directly inside the enclave, with a daemon providing Windows API functionality.

A compatibility layer is necessary, because even though an enclave could call an API function by its exported address using the `eexit` instruction, either an `eenter` or an `eresume` instruction has to be used to return to the enclave with the result of the call. This will not be the case when using the standard Windows API, which uses a simple `ret` instruction to transfer execution back to the caller. This approach is represented in Figure 3.5.

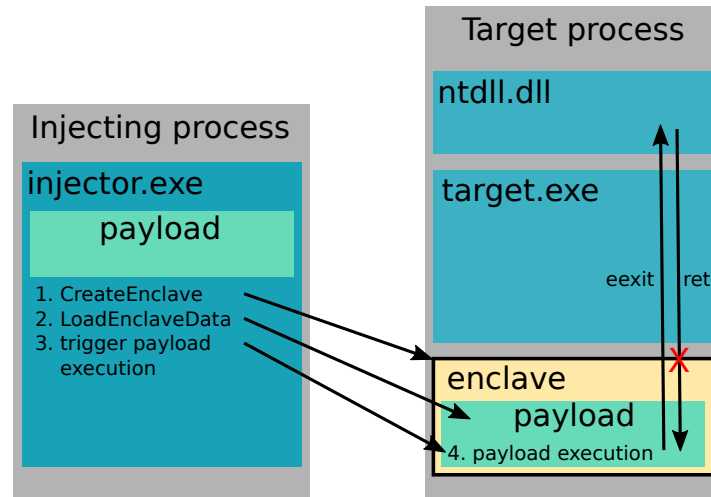


Figure 3.5: Memory injection technique using memory enclaves with the payload executing directly inside the enclave, with API functions called directly.

Another approach would be to use the enclave to store the encrypted payload at runtime, but read the code from outside the enclave at a word granularity, i.e. 64 bits, using the `edbgrd` instruction before executing it. This would, again, not work for process injection because it requires another full injection technique to provide the execution primitive. The same issue exists with using the `ewb` instruction, where the granularity would be an entire page instead of a word.

### 3.4.2 Enclaves as an Allocation and Writing Primitive

The approach we presented of executing the payload inside the enclave illustrates the argument given by [38] that memory enclaves do not provide an adversary with novel capabilities. Our second idea was to abuse the API as an allocation and writing primitive, but execute the injected code without initializing the enclave. Indeed, before initialization, the enclave content is not yet encrypted and freely readable by the host process. This approach no longer leverages the integrated detection evasion provided by the enclave during payload execution. Nonetheless, the usage of novel API functions should still provide the technique with evasion capabilities.

The most interesting function of the enclaves API for this approach is `LoadEnclaveData`. This function copies a buffer to a specified location in memory, whilst at the same time setting the protection of the target location. By setting the protection to `RX` directly, there is no need for



allocating RWX memory or changing protection using `VirtualProtect`, which are two of the usual indicators of process injection. We use this as an allocation and writing primitive, as represented in Figure 3.6. This approach proved successful in injecting our payload into an RX region of memory in the target process. Disappointingly, the payload proved to be impossible to execute. The causes are both in the architecture of SGX, and in the implementation of `LoadEnclaveData`. We argue that no approach to process injection using Intel SGX memory enclaves and their related native Windows API functions is feasible. A detailed explanation is given in section 4.2.

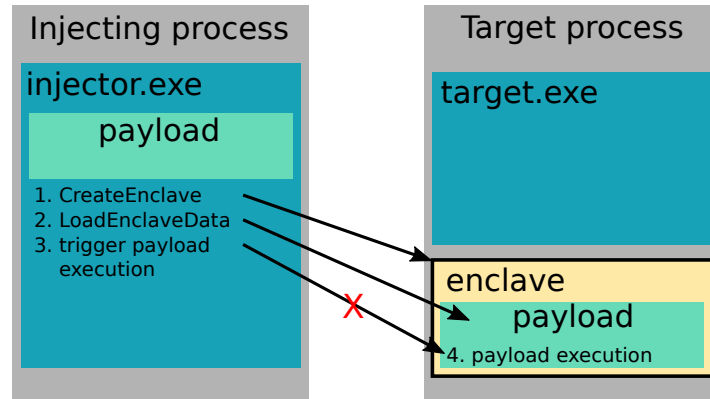


Figure 3.6: Memory injection technique using memory enclaves as an allocation and write primitive.

### 3.4.3 Injection using VBS Enclaves

As we presented in section 2.8, Windows implements its own enclave technology called VBS-based memory enclaves. Because they are enforced by the hypervisor, these enclaves are readable by the hypervisor even after their initialization. We argue that this does not weaken the runtime evasion capabilities we are looking for. Indeed, the hypervisor is a core part the modern Windows security architecture, and hence cannot be modified by others than Microsoft [1]. This means that only Microsoft’s own security solution could obtain the capability to read an enclave’s memory, but we are not aware of such capabilities. Additionally, Microsoft cannot provide telemetry of the inside of an enclave, e.g., through an ETW sensor. Both approaches would break the security guarantees of VBS-based enclaves, thus we expect none to ever be implemented in practice.

Despite their concept and design being documented, the VBS memory enclave details are not. We found two modules compiled for usage in enclaves, namely `ucrtbase_enclave.dll` and `Vertdll.dll`. The former provides the C runtime, and the latter some Windows API functions such as `VirtualAlloc` and `VirtualProtect`. Since VBS enclaves are in the same address space as the hosting process [1], these functions should be usable to inject code into the hosting process. The idea would be to first create a VBS enclave in the host process. This first stage uses an API that is rarely used, and hence likely not monitored for. In a second phase, code executing in the enclave then injects the payload from the enclave into the main host memory and executes it, following

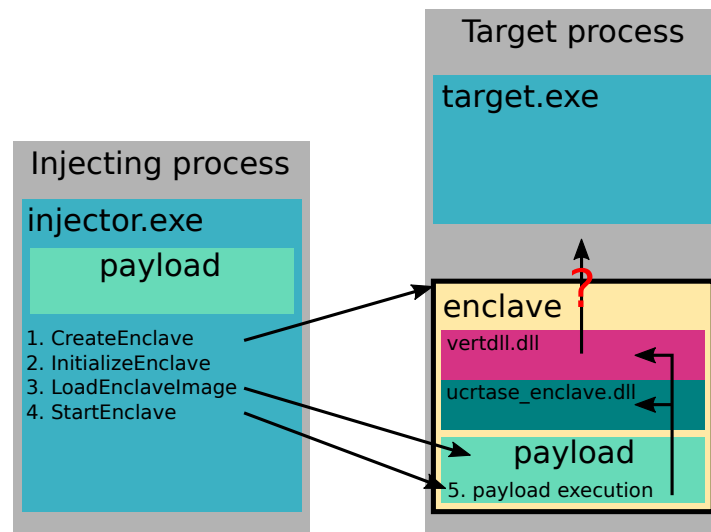


Figure 3.7: Memory injection technique using VBS memory enclaves, abusing the API calls of available DLLs.

for example the classic process injection approach. Since the allocation, writing, and protection setting then execute from inside the process address space, we believe that this should add evasion capabilities. This approach is shown in Figure 3.7. Disappointingly, due to missing documentation, we could not successfully create an enclave running our own code. We provide further details in subsection 4.2.2.

## Chapter 4

# Implementation

Implementation of offensive tools should be performed in an iterative manner, where testing for evasion of detection is done in addition to testing functionality. We define a testing methodology, presented in chapter 5, that can guide the implementation of evasion measures. In this chapter, we present some implementation details of the two injection techniques we implemented. The source code can be consulted in its entirety in the project git repository.

### 4.1 Threadless Injection

In section 3.3 we presented how we improved upon the original threadless injection implementation [10].

#### 4.1.1 Shellcodes

An important element of this technique is the loader shellcode, whose functionality is summarized in Figure 3.2. We split it into two parts, we call the part running in the created thread the *preparer shellcode*, and refer to the main part as the *loader shellcode*.

The loader shellcode performs calls to `VirtualProtect` [70] and `CreateThread` [44]. To enable the shellcode to perform API calls, we resolve all the function export addresses of the required system calls in the injector, and write them into the shellcode before writing it into the target process. This reduces the complexity of the shellcode compared to implementing this functionality inside of it. For example, the main injector logic finds the address of `CreateThread` in the in-memory `Kernel32.dll`, and places it inside the buffer containing the payload. The function is then called following the x64 calling convention. This approach is shown in Listing 4.1. Function addresses are

resolved by parsing the export tables of their exporting DLL, as explained in subsection 4.1.2. The location of the payload in the target process's memory, as well as the original bytes of the hooked function, are set in the same way.

```

1 // parse the module's export table to find the target module
2 PVOID LocateFunction(PLDR_MODULE module, ULONG target_function_hash
   );
3
4 // loader shellcode
5 /*
6 0: pop     rax                                ; set rax to just before the
   hook
7 1: sub     rax,0x5
8 ...
9 1d: sub    rsp,0x78                        ; prepare arguments for
10                                         ; CreateThread
11 21: xor     rcx,rcx
12 24: xor     rdx,rdx
13 27: xor     r8,r8
14 2a: movabs  r8,0x1122334455667788          ; address of thread function
15 34: xor     r9,r9
16 37: mov     QWORD PTR [rsp+0x28],r9
17 3c: mov     QWORD PTR [rsp+0x20],r9
18 41: movabs  rax,0x1122334455667788          ; address of CreateThread into
   rax
19 4b: call    rax                                ; call CreateThread
20 ...
21 5e: pop     rax
22 5f: jmp     rax                                ; resume execution at hooked
   function
23 */
24 byte shellcode_loader[] = { 0x58, 0x48, ..., 0x58, 0xFF, 0xE0 };
25
26 ...
27
28 ULONG name_createthread_hash = 0x3defdc66; // "CreateThread"
29 PVOID addr_createthread = LocateFunction(kernel32,
   name_createthread_hash);
30
31 ...
32
33 // replace the placeholder values in the buffer
34 PVOID addr_preparer = (PVOID)((INT_PTR)loader_location + sizeof(
   shellcode_loader));
35 memcpy(((BYTE*)shellcode_loader) + 0x2c, &addr_preparer, sizeof(
   addr_preparer));

```

```
36 memcpy(((BYTE*)shellcode_loader) + 0x43, &addr_createthread, sizeof
      (addr_createthread));
```

Listing 4.1: Code excerpt showing how we perform `CreateThread` function calls in shellcode.

### 4.1.2 Parsing the Export Table

We first need to resolve a function's export address before we can hook it, or call it manually. Windows provides this functionality natively through the `GetProcAddress` function [50]. Due to its frequent use in malware, we chose to implement this functionality ourselves instead of using it. Additionally, if we need a module that is not loaded in the address space of the injecting process, we would need to load the module using `LoadLibrary` [55] into the injector's address space beforehand. If the target module is proprietary to the software running in the target process, this is suspicious behavior.

The export table of any module loaded in a process's address space can be accessed through its file header. We show the process of locating a module in Figure 4.1, and of accessing the export directory in Figure 4.2. Once the export directory has been found, the target function can be located by name in the `AddressOfNames` array. The same index can then be used to read its ordinal in the `AddressOfNameOrdinals` array. The ordinal value itself indexes into the `AddressOfFunctions` array, where the function's address can finally be read. It is possible to do this because all the required data structures are in the usermode address space, and not in the kernel space. The same approach is usable for modules loaded in the injecting process's address space, and for modules loaded in the target process's address space. Only the implementation will be different, because it requires reading the memory of a remote process.

### 4.1.3 Evasion

We use the subsystem API functions instead of the native functions in `Ntdll.dll` because we believe that our target security solution, since it is developed by Microsoft, monitors at the lowest common denominator of all the API functions anyway. By using the well documented subsystem API functions, we work following the KISS principles, which is one of our objectives formulated in section 3.2. Our research on Microsoft EDR presented in section 5.1 supports this hypothesis. As we will discuss in chapter 5, no detection was triggered due to these subsystem functions, hence our iterative development approach did not require us to go to a lower abstraction level. Performing direct syscalls, as presented in section 2.7, is not used for this project. Indeed, Microsoft Defender EDR does not use API hooks as a sensor. This has been verified in our testing environment, as explained in section 5.1.

We use name hashing, presented in section 2.7, to avoid having function or module names as strings in the binary. This can be an issue during static analysis, especially if the target hook location

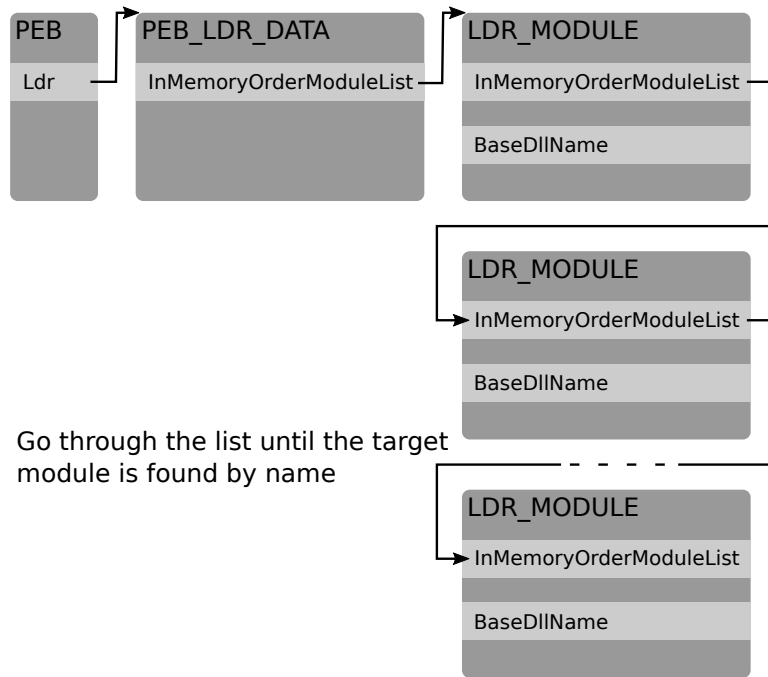


Figure 4.1: The in-memory address of a module can be resolved manually.

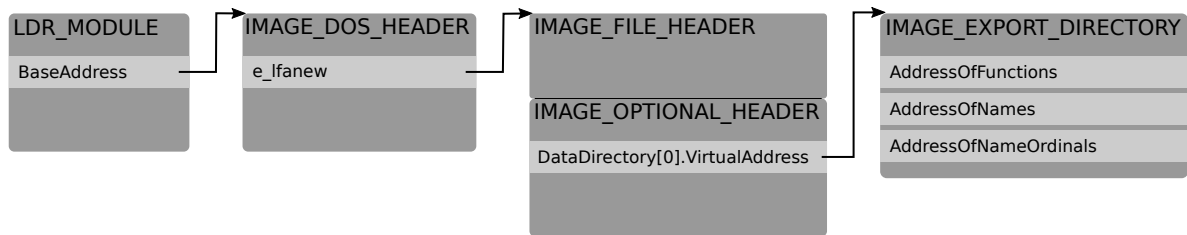


Figure 4.2: Locating the export directory of a module in memory.

is in `Ntdll.dll` since applications are not expected to interact with this module directly. We chose to use the SDBM [25] hash function in this project, because it is easy to implement and collision resistant.

We required our technique to be usable in the presence of ASLR, DEP, and CFG. Because Windows DLLs are not position independent, they are only relocated once per boot. The base address of the same module is identical across all processes, as is the layout in memory of the module. Hence, ASLR does not pose an issue when locating the target function and hooking it. To circumvent DEP, we simply change memory protection as needed using `VirtualProtect` [70]. Finally, CFG does not prevent our technique. The hooked function is CFG valid, hence we can redirect execution to it. The shellcode and payload we inject into memory are located in regions we allocate or set as executable, making them CFG valid as well.

## 4.2 Injection using Memory Enclaves

Using memory enclaves on Windows is a documented functionality [47]. Despite this, we were unable to find any publicly available working example implementation of it. Poor documentation of this API made using it difficult. Other difficulties are related to the specific enclave technologies. Intel SGX is a complex system with limited usage in the consumer software. It seems to be mostly used by large corporations in the context of cloud computing [33]. VBS enclaves are poorly documented by Microsoft, and it is unclear how to and who can sign the DLLs such that the API accepts to load them. Compilation and implementation requirements are also not officially documented.

### 4.2.1 SGX Enclaves

In section 3.4 we explained why memory injection using SGX enclaves is not possible according to our findings. Despite this, we provide some implementation details on it in, because the issues we encountered are relevant for the systematization done in section 6.2.

Software Guard Extensions are thoroughly documented in the Intel Software Developer Manual volume 3D [29]. The Intel SGX Explained paper [12] provides a valuable, despite being long, overview of the technology. There are open source SDKs available: mainly the Intel Linux SDK [34], and the OpenEnclave project [74]. Finally, most of the SGX related resources, such as the SGX101 website [88], focus on how to design secure applications for SGX enclaves, but not on how to implement them. The only resource on the related Windows API, other than its official documentation [47], is the Windows Internals part 2 book [1].

When using the Windows API, some structures used internally by SGX have to be defined and partially filled out by the developer. These are the *SGX Enclave Control Structure* (SECS), *Enclave Signature Structure* (SIGSTRUCT), and *EINIT Token Structure* (EINITTOKEN). Microsoft does not provide definitions for them. We defined our own using the definitions given in the Intel manual [29]. Microsoft does not mention which fields of these structures are to be filled by the developer. We found out how to use them with the help of the pseudocode implementations of the related SGX specific instructions provided in Intel manual [29], the SGX Explained paper [12], and by reverse engineering the Windows enclave API implementation in the kernel.

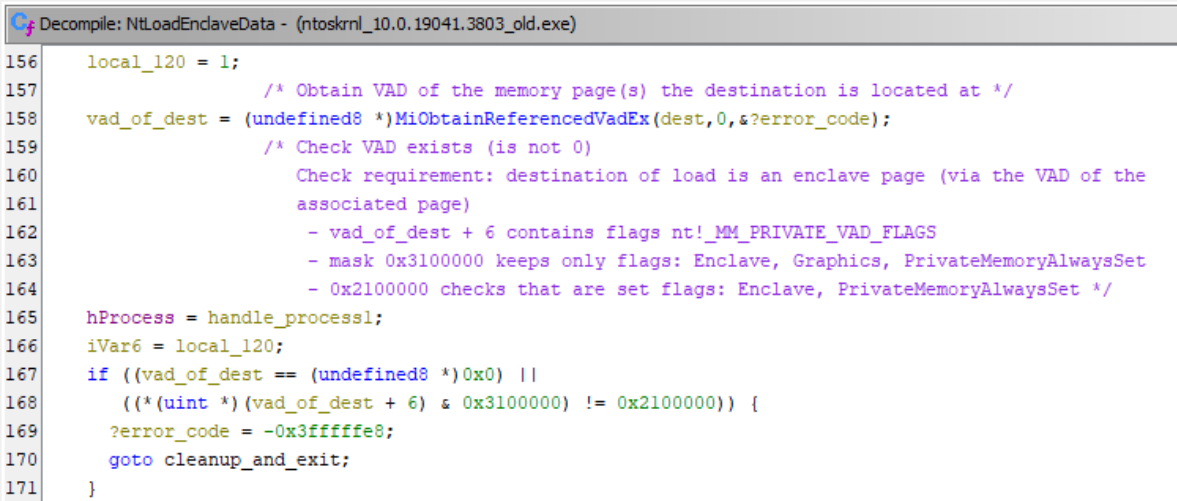
We managed to create an enclave and load data into it. The remaining issue is that, even if running in debug mode, the initialization fails with Windows error 349 `ERROR_ENCLAVE_FAILURE`<sup>1</sup> and enclave error code 8. This error means `SGX_INVALID_SIGNATURE` according to the Intel manual [29] and is returned by the `einit` instruction if the provided signed SIGSTRUCT has not been validated correctly with the enclosed public key. We compared our implementation with the OpenEnclave project, and did not find any obvious errors. Using our own implementation, we could verify our

---

<sup>1</sup>This error code is not listed online, but it can be found in `winerror.h`.

own signature. We request our enclave to be executed in debug mode, which, according to the available documentation, should be allowed to execute even if it has not been signed by a whitelisted key (read section 2.8). Since the launch enclave is provided by Windows, we do not know if this is also the case with it. Other projects utilize their own launch enclave, which allow even production enclaves to not be signed using whitelisted keys.

We first tested the approach of using `LoadEnclaveData` as a writing primitive by targeting an RWX region of memory allocated using `VirtualAlloc`. This call failed. By using WinDbg [53], and reversing the kernel implementation `NtLoadEnclaveData`, we verified that this is because the function correctly verifies that the target memory region's associated VAD is marked as being part of an enclave, i.e., that it is allocated in the EPC. The related decompilation is shown in Figure 4.3.



```

C:\Decompile: NtLoadEnclaveData - (ntoskrnl_10.0.19041.3803_old.exe)
156     local_120 = 1;
157     /* Obtain VAD of the memory page(s) the destination is located at */
158     vad_of_dest = (undefined8 *)MiObtainReferencedVadEx(dest,0,&?error_code);
159     /* Check VAD exists (is not 0)
160        Check requirement: destination of load is an enclave page (via the VAD of the
161        associated page)
162        - vad_of_dest + 6 contains flags nt!_MM_PRIVATE_VAD_FLAGS
163        - mask 0x3100000 keeps only flags: Enclave, Graphics, PrivateMemoryAlwaysSet
164        - 0x2100000 checks that are set flags: Enclave, PrivateMemoryAlwaysSet */
165     hProcess = handle_process1;
166     iVar6 = local_120;
167     if ((vad_of_dest == (undefined8 *)0x0) ||
168         ((*(uint *) (vad_of_dest + 6) & 0x3100000) != 0x2100000)) {
169         ?error_code = -0x3fffffe8;
170         goto cleanup_and_exit;
171     }

```

Figure 4.3: Decompilation of the `NtLoadEnclaveData` function in `ntoskrnl.exe` version 10.0.1.19041.3803. The part shown is where the VAD check is done. The comments in the decompilation are our own.

To pass this check, we first create an enclave, load the payload into it, and then execute it directly. We can successfully load our payload into an RX page in one API call, but execution fails with `NTSTATUS 0xC000001D Illegal instruction` on the first instruction of the payload. This error usually appears when the CPU cannot decode an instruction, e.g., if an AVX-512 instruction is used but the CPU does not support it. In our case, the instruction is valid. Our hypothesis is that the CPU will not execute EPC pages as long as their associated SECS is not set as initialized. Since the SECS is also stored in an EPC, it can only be set as initialized by the `einit` instructions, which is incompatible with this injection approach, because then we can no longer execute the payload from outside the enclave.



#### 4.2.2 VBS Enclaves

Similarly to SGX enclaves, the usage of the Windows API with VBS enclaves is not well documented. There exists no publicly available working example that we are aware of. To test our idea presented in section 3.4, we tried to write our own enclave DLL.

The same API functions are used for VBS enclaves as for SGX enclaves, except for the loading step where `LoadEnclaveImage` is used instead of `LoadEnclaveData`. As this function's name suggests, VBS enclaves expect DLL images to be loaded from disk, instead of an arbitrary buffer of bytes from memory. We proceeded to build our own DLL to execute in an enclave by using information from the official documentation [47], Windows Internals [1], two forum posts [28, 95], and by reversing `Vertdll.dll`, and `SgrmEnclave_secure.dll`. `LoadEnclaveImage` does not load our DLL and returns error `ERROR_GEN_FAILURE` *a device attached to the system is not functioning*. We formulate two hypotheses of what the issue is:

1. The DLL entry point has to implement some required functionality we are unaware of. Indeed, we get the same error when loading `Vertdll.dll`, but `SgrmEnclave_secure.dll` loads successfully. The entry point of the former simply returns, whilst the entry point of the latter does various computations depending on the entry reason.
2. The `IMAGE_ENCLAVE_CONFIG` [52] pointed to by the load configuration is not set correctly. We do not know how this structure can be filled such that the `/ENCLAVE` linker flag uses it.

We found that an enclave DLL has to fulfill following requirements:

1. Required build flags: `/GUARD:cf` to enable CFG, and `/GS-` to disable stack canaries
2. Required linker flags: `/ENCLAVE` and `/ENTRY:{entry_point}` to set the entry point
3. Only link against `vertdll.lib` and `{path_to_wdk}\ucrt_enclave\x64\ucrt.lib`
4. Sign the DLL with EKUs: 1.3.6.1.4.1.311.10.3.24, 1.3.6.1.4.1.311.10.3.37, 1.3.6.1.4.1.311.10.3.6, and 1.3.6.1.5.5.7.3.3

## Chapter 5

# Evaluation

In chapter 3, we explained that process injection techniques have to be developed with respect to various constraints. Evaluating whether the formulated targets are met requires testing against a security solution. We discuss the issues related to this in section 5.1. How we deal with these issues, and how we evaluate the other constraints, is presented in section 5.2. Finally, we present the results of our evaluation in section 5.3.

### 5.1 Challenges of Testing against Security Solutions

Advanced security solutions, such as EDRs, are non-trivial to test against. A first major issue is their black box and proprietary nature. Some exceptions exist, such as the open-source OpenEDR [11], or the public rules of Elastic Security [15, 16]. Nonetheless, the leading EDR platforms such as Microsoft Defender XDR [48], Cybereason [13], and CrowdStrike Falcon [92] are all closed source. There is limited insight into their functionality to be gained from their respective documentation. The second major problem is that, to test against an EDR, it has to be installed on a test system. This implies that either the minimum amount of licenses has to be purchased at high cost, a single license could be obtained through other means, or in the context of a threat simulation assignment, the client could provide a machine running the software.

The black box nature of security solutions is a security feature. Whilst the security of a system should not rely exclusively on the security by obscurity principle [85], it can nonetheless be a desirable property. Security solutions rely on rules to recognize patterns [22]. They are not only used for static analysis, but also during dynamic analysis. With knowledge of these rules, a malicious actor can find, and abuse any potential blind spot. For the same reason, EDRs do not provide analysts with very detailed information on which events are logged, and why exactly an alert was triggered. They try to strike a difficult balance of providing enough information to make forensic analysis and

implementation of custom rules possible, whilst at the same time not providing an easy means for offensive developers to understand why their malware go detected.

Another hurdle when testing against EDRs is their cloud-based nature. None of the major vendors offers a test environment that we know of. This implies that it is only possible to test against a production environment. Consequently, if an injection has been detected during a test, its signature will be integrated into the production database of the solution. The technique or implementation then has to be modified to avoid getting detected again based on this signature. If the signature relies on some static bytes of the binary, then slight modifications in the source code can be used to circumvent these signatures, whilst keeping the same functionality. If the behavior of the technique has been signed instead, then the technique is either entirely unusable, or it has to be modified enough for the rule to no longer match. The consequences of this depends on the exact rule and its quality. In the case where the payload gets detected and a human analyst working for the vendor of the security solution analyzes it, they could go back to the root cause of the alert, i.e., the injection technique, and develop a detection rule specifically for it. We consider this worst case scenario as improbable. Since there is such a large amount of detection events each day, it is not possible to analyze them all manually.

The detection capabilities of security solutions evolve constantly to keep up with adversary knowledge and tactics. For example, the various version information related to Defender XDR are incremented, and updates are pushed to devices multiple times per day. The combination of this constant evolution, and of EDRs black box implementation, makes it difficult to provide testing guarantees. There exist many solutions, each with multiple possible configurations, hence completeness can only be guaranteed with respect to one specific solution and configuration at a time. Because there is no complete knowledge of their implementation, nor exactly which events have been logged, the result of a test depends on what information can be retrieved. Thus, the soundness of the evaluation procedure depends on the quality and exhaustiveness of this data. Finally, reproducibility cannot be guaranteed since the detection logic is updated as close to real time as possible by vendors. Whilst it might be able to force the local detection logic to not update, it is not possible to do the same for the cloud based detection.

## 5.2 Design

We defined the constraints we work under, as well as our targets with respect to them, in Table 3.1. How we evaluate a technique should take into consideration each of these constraints. We summarize our approach in Table 5.1. The technical requirements, such as the CPU architecture, operating system, and specific software requirements have to be taken into account when building the testing setup. We present the setup used in this project in subsection 5.2.3. Most of the constraints have to be included in the design decisions of the testing methodology, which we present here. The resulting methodology is presented in subsection 5.2.1. We discuss our design decisions with respect to the

four properties an evaluation methodology should provide: completeness, relevancy, soundness, and reproducibility.

Our objective is to assess injection techniques according to three criteria: functionality, reliability, and detection. We split the evaluation into two phases. In the first one we evaluate the functionality of the technique, and if it can be detected using static analysis. The second phase is only performed if the first phase was successful, and evaluates the evasion capabilities with respect to an advanced, cloud based security solution. This methodology is presented in detail in section 5.2. Overviews of the setups required by each phase are represented at Figure 5.3 and at Figure 5.4 respectively.

To evaluate the functionality of the technique, it is sufficient to construct the injection performed during testing such that it fulfills the requirements defined in the technique design targets. If the injection is successful under these conditions, then the functionality requirements are fulfilled. As long as the hardware setup, the software setup, the payload, and the target process fulfill the design constraints, the tests are complete and sound. The only relevant functionality of a process injection technique is to successfully perform process injection, which makes a successful injection test relevant. We provide reproducibility by specifying the exact machine configuration, providing source code, artifacts, and the target process.

The reliability of a technique is more difficult to evaluate than its functionality. Because of the many interactions between the elements of the testing setup, this has to be done using an automated script, because otherwise the time cost required to acquire any significant amount of data points becomes prohibitive. Such a script should at least spawn the target process, perform the injection, ensure the payload executed successfully, and repeat. Additionally, it should take into account specificities of the technique under evaluation. For example, threadless injection as presented in section 3.3 is vulnerable to a race condition where the hook is executed a second time before the shellcode removed the hook. Due to the complexity of implementing such a script, we decided that it is out of scope for this project to evaluate the reliability of the technique further than as a byproduct of the other tests.

The main focus of the evaluation is to assess if the injection technique is detected by the target security solution. To achieve this, we remove as many variables that can influence the detection score of the injection as possible. According to our design targets, the adversary already has arbitrary code execution on the target system. Hence, the evaluation of how the injector is delivered on the target system is out of scope. The payload itself, and its execution are out of scope for the same reason. The testing methodology should be payload independent, i.e., it should provide insights into the detection of the technique, and not of the payload. To achieve this we test using two different payloads. The first is a benign payload that executes undetected. The second payload is malicious and successfully detected. By first performing injection using a benign payload, we can assume that any detection occurred because of the injection, and not because of the payload. During the second test, if the injection is successful, but the malicious payload is detected in memory, then the injection's detection score was not high enough to trigger an alert, but high enough to trigger

Constraint	Taken into account in	by
Adversary capabilities	Setup + methodology	Initial situation
Adversary privileges	Methodology	Injector execution
Adversary knowledge	Methodology + technique design	Target process + technique implementation
CPU architecture	Setup	Hardware choice
Operating system	Setup	OS choice
Detection	Setup + methodology	Installed security solution + result gathering
OS mitigations	Setup	OS configuration
Process mitigations	Methodology	Target process
Non-native OS requirements	Setup	OS configuration
Disable detection	Technique design	Technique implementation
Target process	Methodology	Target process
Payload	Methodology	Payload
Persistence	Methodology	Payload + technique implementation
Effect on target process	Methodology	Observation during injection
Indicators of compromise	Methodology	Evaluation metrics
Side effects	Methodology	Observation during injection

Table 5.1: Process injection design constraints, where they are taken into account during the evaluation process, and how.

a memory scan in the target process. Indeed, memory scans are computationally expensive and therefore not performed without reason. Microsoft Defender XDR will then perform an automated investigation on the target system. The exact measures taken by it, and the resulting findings, provide valuable insight into the stealthiness of the technique. Indeed, a possible outcome is that the process where the payload was found in memory is killed, but the automated analysis does not make the link back to the injector binary and leaves it untouched. This would indicate that the injection was not detected at all. Nonetheless, in both cases the exact conclusion from the test depends on the analysis of the data gathered from the EDR. This provides additional information on the stealthiness of our technique. As we discussed in section 5.1, neither of completeness, relevancy, soundness, and reproducibility, can be guaranteed when testing against a black-box security solution.

We note that the *disabling detection* constraint is an outlier. If it is implemented well, then it would be undetected by the security solution, and if it is not then it will be detected. In case of no detection of tampering, it is not possible to know if it was because there was no attempt, or if it was actually successful. Hence, this constraint can only be evaluated based on the actual design and implementation of the process injection technique.

Finally, there are some general principles to consider when implementing and evaluating process injection. The *mark-of-the-web* [90] should be removed to avoid the binary having an unnecessarily

increased detection score initially. Another suspicious characteristic of a binary is missing metadata, such as author, and the binary being unsigned. We decide to not add metadata to injector binary because it introduces too many variables into the evaluation, which would hurt soundness and reproducibility of the evaluation procedure. We also do not sign the executable, because self-signing is arguably as suspicious as not signing, and using a signing certificate issued by a trusted authority means that the certificate cannot be used again once it has been associated to malicious activity.

### 5.2.1 Methodology

During the first stage of our evaluation we assess which capabilities are detected using static analysis in our technique and implementation. We run the injector binary, containing no payload to remove any noise this introduces, against the CAPA [42] tool. The tool output is used to better understand which techniques are known, or which implementation details are considered suspicious. We then test if the standard Defender Antivirus [56] detects the injection. This is a sanity check of evasion capabilities of the technique and implementation. If it is detected at this stage, then it will also be detected by Defender for Endpoint. Before performing the injection, the Threatcheck [79] tool is used to find out if a static detection would take place, and if yes, which bytes are signed. The above steps are represented in Figure 5.1.

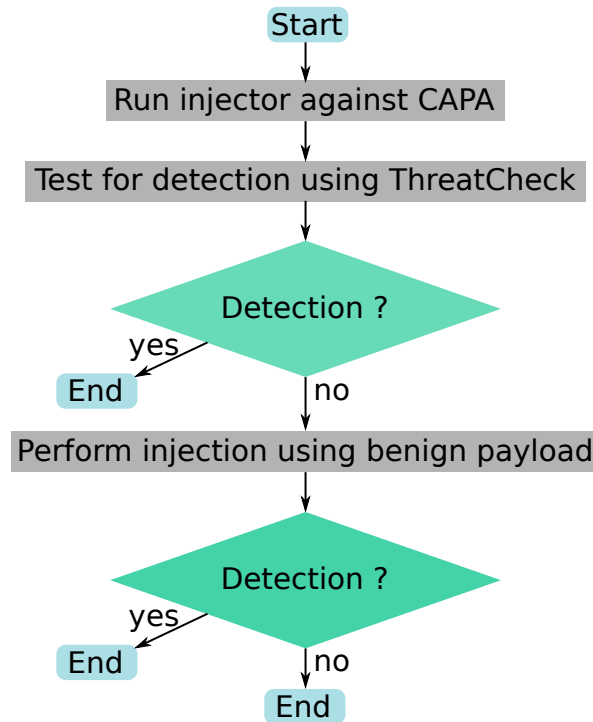


Figure 5.1: High-level overview of the first evaluation step, where the injection technique is tested against static analysis and Microsoft Defender Antivirus.

The second evaluation stage, represented in Figure 5.2, aims to evaluate detection by Microsoft Defender for Endpoint. First, the injector is placed on disk. When used with a Microsoft XDR plan, Defender Antivirus gains additional detection functionality since it is an integral part of the solution [48], though the documentation is not clear on this. Even if it was not detected statically in the first evaluation step, it might be in this one. If it was not detected on disk, we perform injection using the benign payload. This way, we can evaluate if the injection itself triggers a detection. If now the injection was not detected, we perform it again, but this time using the malicious payload. A step-by-step guide for each evaluation step can be found in the project repository.

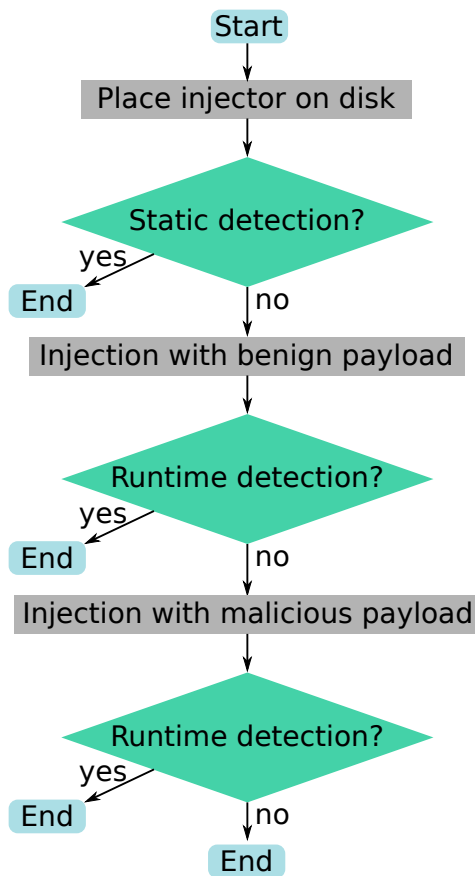


Figure 5.2: High-level overview of the second evaluation step, where the injection technique is tested against Microsoft Defender for Endpoint.

Gathering results during the first evaluation phase is straightforward. The tool output can be taken directly in the case of CAPA and Threatcheck, or, in case of a detection by Defender Antivirus, the given reason is all we have access to. The Microsoft Security Intelligence portal [45] can be used to get information on the named detection reason, though in our experience there is no interesting information published. In the second phase, more information can be extracted from the portal. We discuss this in detail in subsection 5.2.2.

### 5.2.2 Gathering Results from Defender XDR

Making sense of the logged events and how they are interpreted by Microsoft Defender XDR is confusing, as already noted by others [24]. The user facing GUI and databases appear incomplete. Finding out exactly what has been logged is not possible without exploring the internal workings of how Defender for Endpoint gathers telemetry and using third-party tools. Our detailed approach to result gathering is in this project's repository. We explain the reasoning behind it in this subsection.

Detected threats and other logged events are accessible through the Defender XDR Portal. If a threat has been detected, the actions taken by Defender and the results of the subsequent automated analysis of the affected machine can be accessed in the *Incidents & alerts* tab. An *alert* is a single detection event, and an *incident* is a grouping of alerts the EDR concluded to be related. In the incident tab, a graph visualizes the relationship between the various alerts and their assets. The involved assets, e.g., client machines, users, and files, are also listed. Each alert has its own *alert story* displaying the timeline of events and how they are related. Some justification of the detection source is given in the alert pane, such as the category, detection source, and service source. We note that this information seems arbitrary and is not very useful. We are not aware of an official resource documenting exactly what each term means and how they are related.

The alerts and incidents do not provide insight into the exact telemetry and logic that triggered the detection, except for some vague information such as *Detection source: EDR* or *Category: Suspicious activity*. Additionally, if during a test there was no detection event, it is still desirable to know what has been logged and forwarded to the backend data collection system. To get information in both of these cases, the portal offers a *Timeline* of events for each machine, alongside a queryable database of logged events through the *Advanced Hunting* tab. As noted in a blog post by Falconforce [24], the timeline and advanced hunting database do not provide access to the same data. From our own research<sup>1</sup>, we found that the advanced hunting database is a subset of the timeline data, which itself does not contain all the information. Some information is only displayed in the incidents and alerts tabs.

In our experience, the web application is a clunky and unpleasant experience. Extracting data from it is slow and requires many steps. To make the evaluation process faster and more consistent, we built an interactive CLI tool we call *CompassHarvester*. It uses the Microsoft Graph Security API [69] to collect alerts and incidents related to the current experiment, as well as performing advanced hunting queries. The REST endpoint used by the web application to get data for the timeline is queried manually to export its data. The source code of this tool is in this project's repository.

---

<sup>1</sup>The full details are in the project repository.



### 5.2.3 Setup

Even though the general architecture stays the same, the testing setup has to be adapted to take into account the various constraints and functionality targets of the evaluated technique. We summarize the testing setup as used in this project in Table 5.2. Since many requirements of the setup are related to the system configuration, *virtual machines* (VM) are used. This way each machine can be set up once, and a snapshot, i.e., a save of the machine state, can be taken. Before each test run the state of the machine can then be restored to remove any artifacts remaining from the previous test. Note that the security solution keeps track of detections independently of the machine state, meaning that any previous detections will still be present in its backend. We note that each VM has internet access, otherwise the cloud sandbox and EDR backend cannot be reached.

The victim VMs run on a x64 based machine with Windows 10 Pro. The installation is kept as default, i.e., no non-default mitigations such as WDAC are enabled, and no additional software is installed, except for 7-Zip. The first stage VM has Microsoft Defender Antivirus enabled, with cloud-based detection disabled, such as to not send our injector to the cloud unnecessarily. The victim VM for the second stage is domain joined to an Active Directory Tenant deploying Microsoft Defender for Endpoint onto the device. This tenant uses Microsoft XDR to monitor for security events.

Since our threat model assumes the adversary to have arbitrary code execution capabilities, we do not evaluate how the injector binary gets placed onto the victim machine. We run a web server on the attacker machine using Goshs [26], and download the file directly from the victim machine from there. The file binary is uploaded in an encrypted zip archive, and downloaded from the server directly by the victim machine. In both stages, it is downloaded into a folder excluded from antivirus scans. Then the archive is extracted using 7-Zip to avoid it being flagged with the mark-of-the-web<sup>2</sup>. Finally, the binary is moved outside the excluded location.

We require the target process to have ASLR, DEP, and CFG enabled. This can be verified using the *Process Explorer* tool from the *Sysinternals* suite [67].

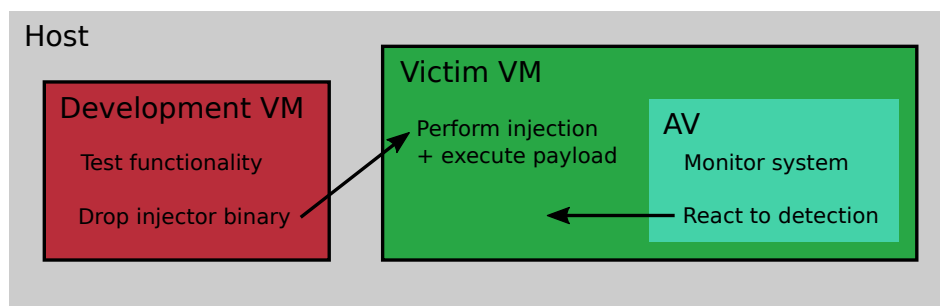


Figure 5.3: High-level overview of our testing infrastructure, as used in first phase evaluation.

<sup>2</sup>The archive file downloaded from the web server has the mark, but files extracted using 7-Zip will not.

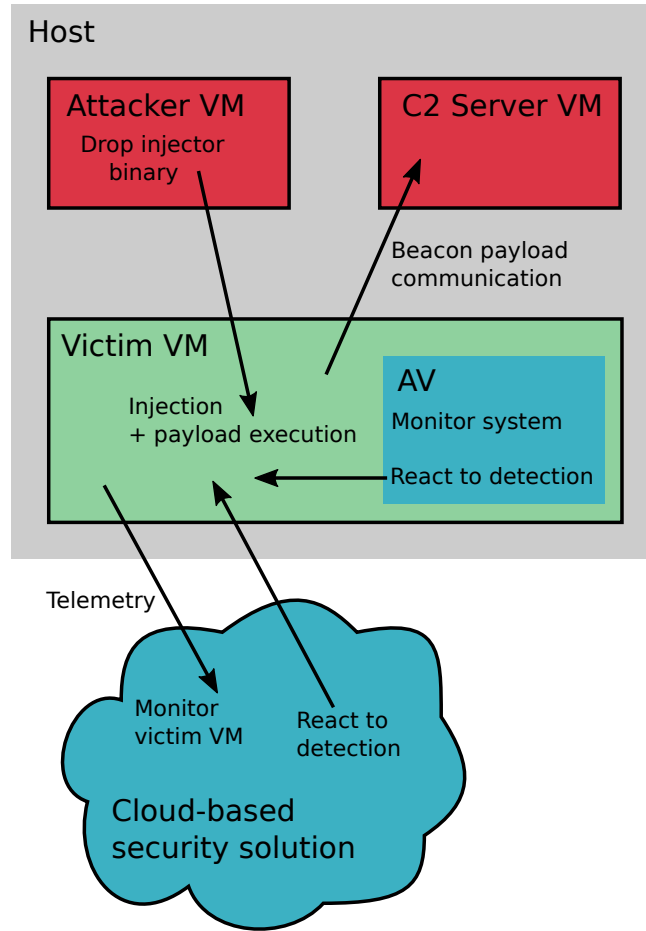


Figure 5.4: High-level overview of our testing infrastructure, including the potential information flow relevant for our evaluation, as used in second phase evaluation.

### 5.3 Results

In this section we present the results of our evaluation. As we defined in section 5.2, only the detection has to be evaluated manually. The remaining constraints are evaluated implicitly through the evaluation methodology and setup. All the reports and data collected during evaluation are stored in this project's repository. We also provide the source code, binaries, payloads, and tool outputs for reproducibility of the experiments. We note that the enclave based injection was not evaluated, since we already concluded in the design discussion at section 3.4 that it does not work.

As a benign payload, we use a shellcode which parses the PE header to locate `winbase/winexec` and then uses it to launch `calc.exe`. The malicious payload is a reverse TCP shell generated using Metasploit [78]. Both have been verified to not trigger and trigger a detection by Defender Antivirus respectively. Except if mentioned otherwise, the target process is a `notepad.exe` backed process,

Element	Product	Version
Virtualization	VMware Workstation Pro	17.5.0 build-22583795
Attacker VM	Microsoft Windows 10 Pro	10.0.19045 Build 19045
C2 server VM	Kali Linux	2024.1
Victim VM	Microsoft Windows 10 Pro	10.0.19045 Build 19045
Victim VM Hardening	default Windows OS settings	–
Antivirus	Microsoft Defender Antivirus	latest at time of test
Security Solution	Microsoft Defender XDR	latest at time of test

Table 5.2: Table containing information on the machines used for the testing setup.

which has been verified to have ASLR, CFG, and DEP enabled. We did not chose an more realistic target, such as `msedge.exe`, during this project because we do not want to rely on this for the evasion. If a technique is not detected when injecting into a process as minimalistic as `notepad`, then it can be assumed that it won't be detected when injecting into a process hosting a complex web browser.

### 5.3.1 Baseline

As a baseline, we use the classic process injection approach described in section 2.9, without any evasion techniques. The CAPA tool correctly identifies the process injection technique as *spawn thread to RWX shellcode* and *inject thread*. There is no static detection by Defender Antivirus, which is expected since the binary we use is our own implementation. Injection with the benign payload is not detected by Defender Antivirus, even though the classic process injection technique is well known.

Surprisingly, Defender for Endpoint does not detect the injection capabilities of the binary on disk, even though rules are known to do this, as is evident by the findings made by CAPA. The injection of the benign payload is detected during execution. Both the injector binary and the target process are correctly considered as part of the attack. As a remediation measure, the injector binary is removed, and the target process is killed. The event that is put forward by Defender is *classic\_payload\_b.exe created a thread remotely inside notepad.exe*, i.e., the `CreateRemoteThread` seems to raise the detection score above the threshold.

The evaluation of the baseline shows us that the classic process injection technique is well-known by defenders, and related behavior is monitored for. The `CreateRemoteThread` system call should be avoided, since it appears to be the API function that raises the detection score above the threshold.

### 5.3.2 Original Threadless Injection

As we found out during our baseline evaluation, the `CreateRemoteThread` function should be avoided. This is the reason given by Cedri Coburn for introducing threadless injection [10]. The original threadless injection implementation is not detected as such by CAPA. The tool does not detect the function hooking execution primitive. Nonetheless, memory allocation and writing into a remote process' virtual memory is detected. The code as it is published on GitHub is signed. Indeed, it is detected on disk by Defender Antivirus as *Trojan:MSIL/Lazy.AB!MTB*. According to Threatcheck, the signature is on the code related to command line output.

We recompiled the code after removing the command line output. This allows to perform injection with the benign payload without detection by Defender Antivirus. The injector is nonetheless detected on disk by the augmented signature database available when subscribed to a Defender XDR plan. The identification as *VirTool:Win32/ThreadlessInj.A* suggests that it is a static signature on the binary.

As an attempt to circumvent this signature, we added `nop` instructions to the loader shellcode. The binary is still detected on disk by Defender for Endpoint, but now as *Trojan:MSIL/Lazy.AB!MTB*. Our hypothesis, supported by the malware naming scheme [51], is that the loader shellcode has been signed manually as an essential component of threadless injection, and that through usage in real-world malicious activity, the C# implementation has been signed automatically as well.

### 5.3.3 Improved Threadless Injection

We implemented our own version of threadless injection in C as described in section 3.3. The CAPA tool detects the memory allocation and writing capabilities into a remote process' virtual address space. It also detects that we parse the PE header to locate DLLs in memory. As expected, since it is our own implementation, the injector is not detected as malicious on disk by static signatures. We can perform injection undetected by Defender Antivirus.

In the second evaluation phase, the injector is again not detected on disk. Injection of the benign payload into the notepad process was performed successfully. The only relevant events that appear in the security portal timeline are that the injecting process changed the memory protection of a region in the address space of the target process. This was evidently not enough to raise the detection score above the required threshold.

When performing injection of the malicious payload, an alert was generated. The injection was successful, and the reverse shell connected to the attacker VM. The shell executes in a child process of the notepad process. An outgoing tcp connection from this process raised the detection score of the target notepad process above the threshold. A memory scan of its address space led to the detection of the unencrypted payload in memory. Since it is signed, a *Behavior:Win32/Meterpreter.gen!A*

was detected as expected. As a mitigation, the notepad process was terminated. The EDR did not make the link with the injection, i.e., it just detected the payload in memory, but not the running reverse shell. The reverse shell stays active, and the injector binary remains on disk.

For reasons unknown to us, this time the memory protection changes from before are not logged. This means that there are no events available in the security portal that provides the necessary information to analyst to make the link between the injector binary and the remote shell. Only the execution of an unknown binary, i.e., the injector, executing chronologically before the detection of the payload, could be a lead to follow.

We argue that this means that our threadless injection technique is undetected by Defender XDR according to our evaluation methodology. Its detection score is low enough that it does not trigger a detection on its own. In combination with our loader shellcode design, even after detection due to a heavily signed payload, it is not considered in the automated investigation.

To detect this technique, the function hook needs to be detected. This can be done by monitoring write access on function entry points of modules loaded into the virtual address space. The whitelisted CFG addresses used by CFG could be used to know which addresses to monitor. A memory forensic tool such as Moneta [75] can also be used to detect the allocated memory in the target process.

## Chapter 6

# Related Work

In section 6.1, we present the process injection techniques we identified during our preparatory research for this project. We provide a systematization on the issues we encountered when developing our own technique using memory enclaves in section 6.2

### 6.1 Known Techniques

There is no peer-reviewed literature on process injection for Windows that we know of. This does not mean that there is little work done on it. There are many GitHub repositories with proof of concepts, or fully fledged tools, as well as talks at conferences, and blog posts. By nature of process injection, there is an inherent incentive to not publish production ready implementations to avoid it being signed. We also assume that sophisticated techniques are kept secret by malicious or state actors for their own purposes. Many of the known techniques are published by independent researchers or companies active in information security, and then presented by others in their own blog posts or implementations. This produces a lot of noise on existing techniques, their names, and effectiveness. Only slight variations in the implementation of conceptually the same technique might lead to it being known under two different names. Additionally, process injection is composed of three primitives (read section 2.9) leading to more confusion on classification. Most techniques are presented as functional, with hypothetical evasion capabilities. We believe that this is because of the issues of testing injection techniques against security solutions we presented in section 5.1.

As recognized by Klein et al. [37], there exists no exhaustive and centralized collection of known techniques. They provide one in their whitepaper related to their talk at BlackHat 2019 [37]. Another list of techniques is given by MITRE [77]. We group injection techniques by their main category, i.e., which element of the operating system they rely on. We then classify them into the three primitives, i.e., memory allocation, memory writing, and execution. If a technique includes all three, then it is

considered as an *all-in-one* technique. This classification is atomic. If a technique is classified as allocation and writing primitive, then it cannot be used for either allocation or writing separately. We provide references to the original source of where we learned of these techniques. Nonetheless, due to authors often not giving references, we do not know if the actual inventors of the techniques are referenced here. An overview is given at Table 6.1. This list is non-exhaustive and does not cover all the known techniques.

### 6.1.1 DLL

Dynamic Link Libraries are designed to be loaded into processes, hence it seems logical that functionality related to them can be leveraged for process injection. *DLL Hollowing* [87] overwrites the `.text` section of a legitimate DLL on disk, and then loads it into the target process. *Phantom DLL Hollowing* [76] improves on this by overwriting the DLL during the loading phase without touching its image on disk. Once the path to a DLL containing malicious code is available in the target process, execution can be redirected to it using *Classic DLL Injection Execution* [37]. Instead of loading a DLL from disk, the *Reflective DLL* [17] technique implements its own loader functionality and loads the DLL directly from the injecting process into the target. The *Unmap+Overwrite*[37] technique enables execution hijacking by unmapping an already loaded dll in the address space of the target, and overwriting its address space using any allocation and writing primitive.

### 6.1.2 Callback

A *callback* is a function registered to run after some action occurs. Either by registering a callback, or by hooking an existing callback, they can be used as an execution primitive by using them to redirect execution to the injected payload. They typically require another action to be performed for the callback to trigger and execute. *Ctrl-Inject* [36] hooks the callbacks related to `Ctrl` signals in console applications. Observed in real-world malware, *KernelControlTable* [49] registers a kernel callback. Hexacorn's *PROPagate* [27] is part of the *Shatter-like* [89] family of techniques leveraging messages controlling GUI windows. The *Instrumentation callback* [96] can be hooked, and is triggered every time kernel hands back control to the process after a system call.

### 6.1.3 Other

The most well known primitives are the ones used for classic process injection, namely the three Windows API functions `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`. This classic allocation and writing primitive is used in most techniques in combination with a different execution primitive. For example, *Inject-Me* [97] uses multiple calls to `SetThreadContext` to execute the payload in chunks. Two other classic execution techniques are *Thread Hijacking* [37], where

Technique	Allocation	Writing	Execution
VirtualAllocEx	X		
WriteProcessMemory		X	
CreateRemoteThread			X
Thread Hijacking			X
APC Execution			X
Ctrl-Inject			X
KernelControlTable			X
Shatter-like			X
Instrumentation Callback			X
DLL Hollowing	X	X	
Phantom DLL Hollowing	X	X	
Classic DLL Injection Execution			X
Reflective DLL	X	X	X
Unmap+Overwrite			X
Atom Bombing		X	
GhostWriting		X	X
Inject-Me			X
memset/memmove		X	
Existing Shared Memory	X	X	
NtMapViewOfSection		X	
StackBomber		X	X
Pool Party			X
COM	X	X	X

Table 6.1: Non-exhaustive overview of known process injection techniques.

the target thread is suspended, its context changed, before resuming its execution, and *APC Execution* [37], which uses `kernel32!QueueUserAPC` to queue an asynchronous process call in the target thread. *Pool Party* [41] uses the thread pool available in all Windows processes to schedule execution. *Existing Shared Memory* can be abused to host the payload [37]. Similarly, `NtMapViewOfSection` can be used to copy the payload from the injector into the target. The *GhostWriting* [6] technique uses return oriented programming with a gadget writing one register to the address of another register to manipulate the memory of the target process. By splitting the payload into 255 byte long strings, *Atom Bombing* [5] copies a payload loaded into the Atom table string store [43] into the target process. Another writing primitive is *memset/memmove* [37] uses `NtQueueApcThread` to make the target process copy the payload over from the injecting process using `memset` and `memmove`. The *StackBomber* [37] technique manipulates the stack to host and execute the payload. Finally, the Windows Component Object Model can be leveraged in multiple ways to perform process injection [19, 20].



## 6.2 Systematization

Our idea to use memory enclaves for process injection, as we presented in section 3.4, did not come to fruition. We provide here a systematization of the issues we encountered during the research and development processes.

### 6.2.1 Explore Underutilized Operating System Functionality

The memory enclaves API [47] initially caught our interest because it exposes functionality analogous to the one used for classic process injection. Indeed, it provides system calls for all three process injection primitives: `CreateEnclave` for allocation, `LoadEnclaveData` for writing, and `CallEnclave` for execution. At the same time, there is no public minimum working example of an enclave managed using this API that we know of.

Identifying OS functionality or APIs previously unused for process injection, and that expose, or can be abused for, at least one of the three primitives, constitutes a valid approach to discovering new techniques. For example, Pool Party [41] used the documented thread pools [68] functionality for process injection before others did. Since every Windows process has a thread pool, it can be abused as an execution primitive. The allocation and writing primitives can be the same as in classic process injection. As with Threadless Injection [10], this change is sufficient to evade detection.

### 6.2.2 Documentation Quality of Operating System Functionality

Nonetheless, even if a functionality is documented, such as the existence and basic usage of the memory enclaves API, does not imply that it is documented thoroughly. There might still be usage quirks or undocumented parts to it. In our case, this manifested in long and tedious reading of the official Intel documentation to understand how the Windows enclave API should be used with Intel SGX enclaves. Similarly, how exactly a VBS enclave DLL should be compiled is not documented.

Poorly documented functionality provides the advantage that it is likely not monitored for by security solutions. This comes at the cost of the time necessary to get it to work, if it is at all possible. Whilst it did not work out for us, there are cases where it did. For example, PROPagate [27] identified a GUI window property that acts as a hook on window messages. An attacker can use the documented `SetProp` API to replace the hook code and trigger execution of injected code. The author of this technique notes that there are other properties that can be abused in the same way. All of these properties are undocumented, but can be abused following some reverse engineering to uncover how.

### **6.2.3 Account for Implications of the Underlying Technology**

Most of the native Windows functionality is implemented traditionally, i.e., there is a userspace data structure and API that can be used to interact with some functionality implemented in the kernel. For example, the `WriteProcessMemory` API can be used to write into a remote process. To guarantee the expected functionality, the kernel verifies that the target location is indeed allocated and writable. As is made evident by the classic process injection technique, this check does not prevent an attacker of abusing the API.

Nonetheless, there are circumstances where a functionality check also prevents abuse. In our case, the hardware implemented nature of the underlying SGX technology is what makes it impossible to abuse the enclave API for process injection. If a functionality is implemented using hardware support, or even fully implemented in hardware as with SGX, special care has to be given to consider the possible implications.

### **6.2.4 Understand the Complexity of the System**

Another consequence of the implementation details of the functionality is eventual limitations that are not directly apparent. The complexity of the underlying technology has to be taken into account when deciding how to do research. For example, COM based techniques require knowledge of a notoriously bloated and complex technology to research correctly. Despite this, with enough knowledge it is possible to find ways to leverage it for process injection [19, 20].

Other times, the complexity of a system can lead to issues appearing late in the research process. `LoadEnclaveData` seems to support loading arbitrary data into the enclave. Whilst this is true in theory, in practice not all instructions are available from inside the enclave. The complexity of Intel SGX has been underestimated in our case when deciding to research injection with memory enclaves, which lead to spending a lot of time trying to implement enclaves, understanding the underlying technology, to finally arrive at the conclusion that it is not possible to use SGX enclaves and the related API for injection.

## Chapter 7

# Conclusion

The modern security landscape is constantly evolving. Threat simulations play an important part in establishing the security stance of an organization. Process injection is a useful and commonly used technique to achieve objectives whilst staying undetected during such assignments.

In our project, we presented the background required to work with, and implement, injection techniques. By clearly defining the design conditions and constraints of process injection in the context of this project, we give a starting point for red teaming operators to reason about process injection in their assignments.

We explored the potential of using the Windows memory enclaves API for process injection. Whilst we did not find a novel technique, we implemented an SGX enclave which gets closer to a working example using this API than what is publicly available. Through this, and reverse engineering of its implementation, we have shown that the Windows enclave API cannot be used for process injection with SGX enclaves. We also explored the potential of VBS enclaves with the same API, and proposed a design which might lead to a new technique, though poor documentation prevented us from implementing a proof of concept. Since extensive reverse-engineering would be required to write an implementation to test our proposed design, we leave this as future work once Microsoft publishes more thorough documentation on this technology.

As a consequence of our research on memory enclaves for process injection, we provided a systematization of issues that can surface when performing such research. This provides useful advice for researchers to identify similar issues earlier than we did.

The design considerations of malware are dependent on the security of the system it is intended to be used on. We discuss the related evaluation challenges and how to approach them. Using our evaluation methodology, we have shown that our improved threadless injection variant successfully evades detection by Microsoft Defender XDR. In combination with the CompassHarvester tool, our evaluation methodology enables operators to efficiently evaluate tools against Defender XDR.

How to evaluate process injection techniques is not discussed in depth in current research. Our reasoning on how to evaluate against modern security solutions, and our application thereof, raises the current standard. We take into account soundness, relevancy, completeness, and reproducibility to the extent possible with a black-box, cloud-based detection logic. Further research into evaluation against state-of-the-art detection solutions is necessary in our opinion. More sophisticated approaches performing a man-in-the-middle between sensors and the agent, and between the agent and the cloud backend, could provide valuable insight into what is detected and logged. For this, major reverse engineering efforts are required for every EDR product. We believe that the current state of threat simulation in general would benefit greatly from better formalization and classification of techniques such as process injection. A better understanding of how state-of-the-art security solutions are implemented is necessary for an efficient deployment of them by defenders. If blind spots are understood by red teaming operators, they can be used during assignments to prove that relying only on detection is not sufficient to provide security to an organization.

# Bibliography

- [1] Andrea Allievi, Alex Ionescu, Mark E. Russinovich, and David A. Solomon. *Windows Internals, Part 2*. Seventh edition. Pearson Education, 2021. ISBN: 978-0-13-546240-9.
- [2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O’Keeffe, Mark L Stillwell, David Goltzsche, David Eyers, Rudiger Kapitza, Peter Pietzuch, and Christof Fetzer. “SCONE: Secure Linux Containers with Intel SGX”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Nov. 2016), pp. 689–703.
- [3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.10. Arpaci-Dusseau Books, Oct. 2023. URL: <http://www.ostep.org>.
- [4] Andrew Baumann, Marcus Peinado, and Galen Hunt. “Shielding Applications from an Untrusted Cloud with Haven”. In: *ACM Transactions on Computer Systems* 33.3 (Aug. 2015), 8:1–8:26. ISSN: 0734-2071. DOI: 10.1145/2799647. (Visited on 02/06/2024).
- [5] *BreakingMalwareResearch/Atom-Bombing*. BreakingMalwareResearch. Feb. 2024. URL: <https://github.com/BreakingMalwareResearch/atom-bombing> (visited on 02/28/2024).
- [6] *C0de90e7/GhostWriting*. Feb. 2024. URL: <https://github.com/c0de90e7/GhostWriting> (visited on 02/29/2024).
- [7] Brian Catlin, Jamie E. Hanrahan, Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 1*. Seventh edition. Redmond: Microsoft Press, 2017. ISBN: 978-0-7356-8418-8.
- [8] Adam Chester. *Hiding Your .NET - ETW*. URL: <https://blog.xpnsec.com/hiding-your-dotnet-etw/> (visited on 01/31/2024).
- [9] Ryan Cobb. *Cobbr/SharpSploit*. Jan. 2024. URL: <https://github.com/cobbr/SharpSploit> (visited on 01/31/2024).
- [10] Ceri Coburn. *Threadless Process Injection*. Dec. 2023. URL: <https://github.com/CCob/ThreadlessInject> (visited on 12/09/2023).
- [11] *ComodoSecurity/Openedr*. Comodo Cyber Security. Feb. 2024. URL: <https://github.com/ComodoSecurity/openedr> (visited on 02/09/2024).

- [12] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. 2016. URL: <https://eprint.iacr.org/2016/086> (visited on 02/05/2024).
- [13] Cybereason. *Cybersecurity Software | Cybereason*. URL: <https://www.cybereason.com> (visited on 02/09/2024).
- [14] Benjamin Delpy. *Gentilkiwi/Mimikatz*. Jan. 2024. URL: <https://github.com/gentilkiwi/mimikatz> (visited on 01/30/2024).
- [15] *Elastic/Detection-Rules*. elastic. Jan. 2024. URL: <https://github.com/elastic/detection-rules> (visited on 02/01/2024).
- [16] *Elastic/Protections-Artifacts*. elastic. Feb. 2024. URL: <https://github.com/elastic/protections-artifacts> (visited on 03/01/2024).
- [17] Stephen Fewer. *Stephenfewer/ReflectiveDLLInjection*. Feb. 2024. URL: <https://github.com/stephenfewer/ReflectiveDLLInjection> (visited on 02/28/2024).
- [18] James Forshaw. *COM in Sixty Seconds! (Well Minutes More Likely)*. Aug. 2017. URL: <https://www.youtube.com/watch?v=dfMuzAZRGm4> (visited on 01/31/2024).
- [19] James Forshaw. *Injecting Code into Windows Protected Processes Using COM - Part 1*. Oct. 2018. URL: <https://googleprojectzero.blogspot.com/2018/10/injecting-code-into-windows-protected.html> (visited on 02/28/2024).
- [20] James Forshaw. *Injecting Code into Windows Protected Processes Using COM - Part 2*. Nov. 2018. URL: <https://googleprojectzero.blogspot.com/2018/11/injecting-code-into-windows-protected.html> (visited on 02/28/2024).
- [21] *Gramineproject/Gramine*. The Gramine Project. Feb. 2024. URL: <https://github.com/gramineproject/gramine> (visited on 03/01/2024).
- [22] Matt Hand. *Evading EDR: The Definitive Guide to Defeating Endpoint Detection Systems*. San Francisco, CA: No Starch Press, 2024. ISBN: 978-1-71850-334-2.
- [23] Matt Hand. *Matterpreter/OffensiveCSharp*. Jan. 2024. URL: <https://github.com/matterpreter/OffensiveCSharp> (visited on 01/31/2024).
- [24] Olaf Hartong. *Microsoft Defender for Endpoint Internals 0x04 — Timeline*. Oct. 2023. URL: <https://medium.com/falconforce/microsoft-defender-for-endpoint-internals-0x04-timeline-3f01282839e4> (visited on 02/13/2024).
- [25] *Hash Functions*. URL: <http://www.cse.yorku.ca/%7B%5Ctextasciitilde%7Ddoz/hash.html> (visited on 02/07/2024).
- [26] Patrick Hener. *Patrickhener/Goshs*. Jan. 2024. URL: <https://github.com/patrickhener/goshs> (visited on 02/19/2024).
- [27] Adam Hexacorn. *PROPagate – a New Code Injection Trick*. Oct. 2017. URL: <https://www.hexacorn.com/blog/2017/10/26/propagate-a-new-code-injection-trick/> (visited on 02/28/2024).

- [28] *I Want to Create a Vbs Enclave Module, but LoadEnclaveImageW Return 557, How Can i Get the Digital Certificate?* URL: <https://learn.microsoft.com/en-us/answers/questions/1289713/i-want-to-create-a-vbs-enclave-module-but-loadencl?page=1> (visited on 02/07/2024).
- [29] Intel. *64 and IA-32 Architectures Software Developer's Manual Volume 3D: System Programming Guide, Part 4*. URL: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html> (visited on 02/05/2024).
- [30] Intel. *SGX Product Licensing*. URL: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sgx-product-licensing.html> (visited on 02/05/2024).
- [31] Intel. *Software Guard Extensions (Intel® SGX) Debug and Build Configurations*. URL: <https://www.intel.com/content/www/us/en/content-details/671525/intel-software-guard-extensions-intel-sgx-debug-and-build-configurations.html> (visited on 02/05/2024).
- [32] Intel(R) *Software Guard Extensions Developer Guide*. Jan. 2024.
- [33] *Intel/Confidential-Computing-Zoo*. Intel Corporation. Feb. 2024. URL: <https://github.com/intel/confidential-computing-zoo> (visited on 02/29/2024).
- [34] *Intel/Linux-Sgx*. Intel Corporation. Feb. 2024. URL: <https://github.com/intel/linux-sgx> (visited on 02/07/2024).
- [35] Jonathan Johnson. *Uncovering Windows Events*. Mar. 2023. URL: <https://jsecurity101.medium.com/uncovering-windows-events-b4b9db7eac54> (visited on 01/29/2024).
- [36] Rotem Kerner. *Ctrl-Inject*. URL: <https://web.archive.org/web/20180513201535/https://blog.ensilo.com/ctrl-inject> (visited on 02/28/2024).
- [37] Amit Klein and Itzik Kotler. "Windows Process Injection in 2019". In: (2019). URL: <https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All-wp.pdf> (visited on 02/27/2024).
- [38] Kubilay Ahmet Küçük, Steve Moyle, Andrew Martin, Alexandru Mereacre, and Nicholas Allott. "SoK: How Not to Architect Your Next-Generation TEE Malware?" In: *Proceedings of the 11th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '22. New York, NY, USA: Association for Computing Machinery, Sept. 2023, pp. 35–44. ISBN: 978-1-4503-9871-8. DOI: 10.1145/3569562.3569568. (Visited on 02/05/2024).
- [39] Paul L. (@am0nsec). *Am0nsec/HellsGate*. Feb. 2024. URL: <https://github.com/am0nsec/HellsGate> (visited on 02/06/2024).
- [40] *Lazarus Group*. URL: <https://attack.mitre.org/groups/G0032/> (visited on 02/28/2024).
- [41] Leviev, Alon. *Process Injection Using Windows Thread Pools*. URL: <https://www.safebreach.com/blog/process-injection-using-windows-thread-pools/> (visited on 02/28/2024).

- [42] *Mandiant/Capa*. MANDIANT. Jan. 2024. URL: <https://github.com/mandiant/capa> (visited on 01/30/2024).
- [43] Microsoft. *About Atom Tables*. May 2021. URL: <https://learn.microsoft.com/en-us/windows/win32/dataxchg/about-atom-tables> (visited on 02/28/2024).
- [44] Microsoft. *CreateThread Function (Processthreadsapi.h)*. July 2023. URL: <https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread> (visited on 02/06/2024).
- [45] Microsoft. *Cyberthreats, Viruses, and Malware - Microsoft Security Intelligence*. URL: <https://www.microsoft.com/en-us/wdsi/threats> (visited on 02/25/2024).
- [46] Microsoft. *Download the Windows Driver Kit (WDK)*. Jan. 2024. URL: <https://learn.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk> (visited on 01/31/2024).
- [47] Microsoft. *Enclaveapi.h Header*. Feb. 2024. URL: <https://learn.microsoft.com/en-us/windows/win32/api/enclaveapi/> (visited on 02/05/2024).
- [48] Microsoft. *Extended Detection and Response (XDR)*. URL: <https://www.microsoft.com/en-us/security/business/solutions/extended-detection-response-xdr> (visited on 02/06/2024).
- [49] Microsoft. *FinFisher Exposed: A Researcher's Tale of Defeating Traps, Tricks, and Complex Virtual Machines*. Mar. 2018. URL: <https://www.microsoft.com/en-us/security/blog/2018/03/01/finfisher-exposed-a-researchers-tale-of-defeating-traps-tricks-and-complex-virtual-machines/> (visited on 02/28/2024).
- [50] Microsoft. *GetProcAddress Function (Libloaderapi.h)*. Feb. 2024. URL: <https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress> (visited on 02/07/2024).
- [51] Microsoft. *How Microsoft Names Malware*. Feb. 2024. URL: <https://learn.microsoft.com/en-us/microsoft-365/security/defender/malware-naming?view=o365-worldwide> (visited on 02/26/2024).
- [52] Microsoft. *IMAGE\_ENCLAVE\_CONFIG64 (Winnt.h)*. Feb. 2024. URL: [https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image%5C\\_enclave%5C\\_config64](https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image%5C_enclave%5C_config64) (visited on 02/08/2024).
- [53] Microsoft. *Install WinDbg*. Dec. 2023. URL: <https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/> (visited on 02/07/2024).
- [54] Microsoft. *Isolated User Mode (IUM) Processes*. Jan. 2021. URL: <https://learn.microsoft.com/en-us/windows/win32/procthread/isolated-user-mode--ium--processes> (visited on 03/11/2024).



- [55] Microsoft. *LoadLibraryW Function (Libloaderapi.h)*. Feb. 2023. URL: <https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibraryw> (visited on 02/07/2024).
- [56] Microsoft. *Microsoft Defender Antivirus in Windows*. Jan. 2024. URL: <https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/microsoft-defender-antivirus-windows?view=o365-worldwide> (visited on 02/20/2024).
- [57] Microsoft. *Microsoft Defender for Endpoint Documentation*. URL: <https://learn.microsoft.com/en-us/microsoft-365/security/defender-endpoint/?view=o365-worldwide> (visited on 01/30/2024).
- [58] Microsoft. *Microsoft Defender Portal*. Jan. 2024. URL: <https://learn.microsoft.com/en-us/microsoft-365/security/defender/microsoft-365-defender-portal?view=o365-worldwide> (visited on 02/20/2024).
- [59] Microsoft. *Microsoft Learn: Build Skills That Open Doors in Your Career*. URL: <https://learn.microsoft.com/en-us/> (visited on 01/30/2024).
- [60] Microsoft. *Mitigate Threats by Using Windows 10 Security Features*. Mar. 2023. URL: <https://learn.microsoft.com/en-us/windows/security/threat-protection/overview-of-threat-mitigations-in-windows-10> (visited on 01/25/2024).
- [61] Microsoft. *New Low-Level Binaries*. Apr. 2021. URL: <https://learn.microsoft.com/en-us/windows/win32/win7appqual/new-low-level-binaries> (visited on 01/18/2024).
- [62] Microsoft. *Overview of .NET Framework*. Mar. 2023. URL: <https://learn.microsoft.com/en-us/dotnet/framework/get-started/overview> (visited on 01/31/2024).
- [63] Microsoft. *Overview of Apps on Windows Client Devices*. Aug. 2023. URL: <https://learn.microsoft.com/en-us/windows/application-management/overview-windows-apps> (visited on 01/22/2024).
- [64] Microsoft. *PE Format*. Mar. 2023. URL: <https://learn.microsoft.com/en-us/windows/win32/debug/pe-format> (visited on 01/25/2024).
- [65] Microsoft. *SetProcessMitigationPolicy Function (Processthreadsapi.h)*. Nov. 2022. URL: <https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-setprocessmitigationpolicy> (visited on 01/25/2024).
- [66] Microsoft. *Symbols for Windows Debugging*. Dec. 2022. URL: <https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/symbols> (visited on 01/31/2024).
- [67] Microsoft. *Sysinternals*. Feb. 2024. URL: <https://learn.microsoft.com/en-us/sysinternals/> (visited on 02/26/2024).
- [68] Microsoft. *Thread Pools*. Jan. 2021. URL: <https://learn.microsoft.com/en-us/windows/win32/procthread/thread-pools> (visited on 03/13/2024).

- [69] Microsoft. *Use the Microsoft Graph Security API*. June 2023. URL: <https://learn.microsoft.com/en-us/graph/api/resources/security-api-overview?view=graph-rest-1.0> (visited on 02/25/2024).
- [70] Microsoft. *VirtualProtect Function (Memoryapi.h)*. Feb. 2024. URL: <https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect> (visited on 02/06/2024).
- [71] Microsoft. *Windows SDK - Windows App Development*. URL: <https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/> (visited on 01/31/2024).
- [72] Microsoft. *X64 ABI Conventions*. Apr. 2022. URL: <https://learn.microsoft.com/en-us/cpp/build/x64-software-conventions?view=msvc-170> (visited on 01/26/2024).
- [73] Peyton Smith and Mitchell Moser. *7 Common Microsoft AD Misconfigurations That Adversaries Abuse*. Feb. 2021. URL: <https://www.crowdstrike.com/blog/seven-common-microsoft-ad-misconfigurations-that-adversaries-abuse/> (visited on 01/31/2024).
- [74] *Openenclave/Openenclave*. Open Enclave. Feb. 2024. URL: <https://github.com/openenclave/openenclave> (visited on 02/07/2024).
- [75] Forrest Orr. *Forrest-Orr/Moneta*. Mar. 2024. URL: <https://github.com/forrest-orr/moneta> (visited on 03/12/2024).
- [76] Forrest Orr. *Masking Malicious Memory Artifacts – Part I: Phantom DLL Hollowing*. URL: <https://www.forrest-orr.net/post/malicious-memory-artifacts-part-i-dll-hollowing> (visited on 02/28/2024).
- [77] *Process Injection, Technique T1055*. URL: <https://attack.mitre.org/techniques/T1055/> (visited on 02/09/2024).
- [78] *Rapid7/Metasploit-Framework*. Rapid7. Mar. 2024. URL: <https://github.com/rapid7/metasploit-framework> (visited on 03/01/2024).
- [79] *Rasta-Mouse/ThreatCheck*. Jan. 2024. URL: <https://github.com/rasta-mouse/ThreatCheck> (visited on 01/29/2024).
- [80] *ReactOS*. URL: <https://reactos.org/> (visited on 01/31/2024).
- [81] *ReactOS Documentation*. URL: <https://doxygen.reactos.org/> (visited on 01/31/2024).
- [82] Wang Ruofan. *AI-driven Adaptive Protection in Microsoft Defender for Endpoint*. URL: <https://techcommunity.microsoft.com/t5/microsoft-defender-for-endpoint/ai-driven-adaptive-protection-in-microsoft-defender-for-endpoint/ba-p/2966491> (visited on 03/11/2024).
- [83] Joanna Rutkowska. *Thoughts on Intel's Upcoming Software Guard Extensions (Part 1)*. Aug. 2013. URL: <https://theinvisiblethings.blogspot.com/2013/08/thoughts-on-intels-upcoming-software.html> (visited on 02/05/2024).

- [84] Joanna Rutkowska. *Thoughts on Intel's Upcoming Software Guard Extensions (Part 2)*. Sept. 2013. URL: <https://theinvisiblethings.blogspot.com/2013/09/thoughts-on-intels-upcoming-software.html> (visited on 02/05/2024).
- [85] K A Scarfone, W Jansen, and M Tracy. *Guide to General Server Security*. Tech. rep. NIST SP 800-123. Gaithersburg, MD: National Institute of Standards and Technology, 2008, NIST SP 800-123. DOI: 10.6028/NIST.SP.800-123. (Visited on 01/31/2024).
- [86] Michael Schwarz, Samuel Weiser, and Daniel Gruss. "Practical Enclave Malware with Intel SGX". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Roberto Perdisci, Cl  mentine Maurice, Giorgio Giacinto, and Magnus Almgren. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 177–196. ISBN: 978-3-030-22038-9. DOI: 10.1007/978-3-030-22038-9\_9.
- [87] SECFORCE. *DLL Hollowing*. URL: <http://www.secforce.com/blog/dll-hollowing-a-deep-dive-into-a-stealthier-memory-allocation-variant/> (visited on 02/28/2024).
- [88] *SGX 101*. URL: <https://sgx101.gitbook.io/sgx101> (visited on 02/07/2024).
- [89] *Shatter Attacks - How to Break Windows*. Sept. 2006. URL: <https://web.archive.org/web/20060904080018/http://security.tombom.co.uk/shatter.html> (visited on 02/28/2024).
- [90] Matt Susannah. *Why so, ISO? Mark-of-the-Web, Explained*. URL: <https://redcanary.com/blog/iso-files/> (visited on 02/19/2024).
- [91] Claudiu Teodorescu, Igor Korkin, and Andrey Golchikov. "Veni, No Vidi, No Vici: Attacks on ETW Blind EDR Sensors". In: *Black Hat Europe* (2021).
- [92] *The CrowdStrike Falcon   Platform*. URL: <https://www.crowdstrike.com/falcon-platform/> (visited on 02/09/2024).
- [93] "The Lazarus Heist: How North Korea Almost Pulled off a Billion-Dollar Hack". In: *BBC News* (June 2021). URL: <https://www.bbc.com/news/stories-57520169> (visited on 02/28/2024).
- [94] *thefLink/RecycledGate*. Feb. 2024. URL: <https://github.com/thefLink/RecycledGate> (visited on 02/06/2024).
- [95] *Virtualization Based Security (VBS) Enclave - Image Requirements*. URL: [https://learn.microsoft.com/en-us/answers/questions/690951/virtualization-based-security-\(vbs\)-enclave-image](https://learn.microsoft.com/en-us/answers/questions/690951/virtualization-based-security-(vbs)-enclave-image) (visited on 02/07/2024).
- [96] *Weaponizing Mapping Injection with Instrumentation Callback for Stealthier Process Injection*. URL: <https://splintercod3.blogspot.com/p/weaponizing-mapping-injection-with.html> (visited on 02/28/2024).
- [97] Alon Weinberg. *Inject Me X64 Injection-less Code Injection*. July 2019. URL: <https://www.deepinstinct.com/blog/inject-me-x64-injection-less-code-injection> (visited on 02/28/2024).

- [98] *Welcome to YARA's Documentation!* URL: <https://yara.readthedocs.io/en/stable/index.html> (visited on 02/12/2024).

## Appendix A

### ETW-TI Events

Below are the keys of events logged by the ETW-TI provider, as retrieved from running `logman.exe query providers Microsoft-Windows-Threat-Intelligence`. Their names provide insight into what actions are monitored by security solutions utilizing the provider. We can see that many are closely related to process injection. Events with *ALLOCVM\_REMOTE* in their key likely refer to memory allocations into a remote process using the `VirtualAllocEx` API call. Similarly, *PROTECTVM\_REMOTE* likely refers to `VirtualProtectEx` calls. Notice that some events contain *KERNEL\_CALLER* as a suffix, indicating that not only the user mode functions, such as `ntdll!NtAllocateVirtualMemory`, are monitored, but also their kernel mode counterparts. This capability makes telemetry sources such as DLL hooks in user space redundant, since monitoring kernel calls crosses a security boundary.

- `KERNEL_THREATINT_KEYWORD_ALLOCVM_LOCAL`
- `KERNEL_THREATINT_KEYWORD_ALLOCVM_LOCAL_KERNEL_CALLER`
- `KERNEL_THREATINT_KEYWORD_ALLOCVM_REMOTE`
- `KERNEL_THREATINT_KEYWORD_ALLOCVM_REMOTE_KERNEL_CALLER`
- `KERNEL_THREATINT_KEYWORD_PROTECTVM_LOCAL`
- `KERNEL_THREATINT_KEYWORD_PROTECTVM_LOCAL_KERNEL_CALLER`
- `KERNEL_THREATINT_KEYWORD_PROTECTVM_REMOTE`
- `KERNEL_THREATINT_KEYWORD_PROTECTVM_REMOTE_KERNEL_CALLER`
- `KERNEL_THREATINT_KEYWORD_MAPVIEW_LOCAL`
- `KERNEL_THREATINT_KEYWORD_MAPVIEW_LOCAL_KERNEL_CALLER`

- KERNEL\_THREATINT\_KEYWORD\_MAPVIEW\_REMOTE
- KERNEL\_THREATINT\_KEYWORD\_MAPVIEW\_REMOTE\_KERNEL\_CALLER
- KERNEL\_THREATINT\_KEYWORD\_QUEUEUSERAPC\_REMOTE
- KERNEL\_THREATINT\_KEYWORD\_QUEUEUSERAPC\_REMOTE\_KERNEL\_CALLER
- KERNEL\_THREATINT\_KEYWORD\_SETTHREADCONTEXT\_REMOTE
- KERNEL\_THREATINT\_KEYWORD\_SETTHREADCONTEXT\_REMOTE\_KERNEL\_CALLER
- KERNEL\_THREATINT\_KEYWORD\_READVM\_LOCAL
- KERNEL\_THREATINT\_KEYWORD\_READVM\_REMOTE
- KERNEL\_THREATINT\_KEYWORD\_WRITEVM\_LOCAL
- KERNEL\_THREATINT\_KEYWORD\_WRITEVM\_REMOTE
- KERNEL\_THREATINT\_KEYWORD\_SUSPEND\_THREAD
- KERNEL\_THREATINT\_KEYWORD\_RESUME\_THREAD
- KERNEL\_THREATINT\_KEYWORD\_SUSPEND\_PROCESS
- KERNEL\_THREATINT\_KEYWORD\_RESUME\_PROCESS
- KERNEL\_THREATINT\_KEYWORD\_FREEZE\_PROCESS
- KERNEL\_THREATINT\_KEYWORD\_THAW\_PROCESS
- KERNEL\_THREATINT\_KEYWORD\_CONTEXT\_PARSE
- KERNEL\_THREATINT\_KEYWORD\_EXECUTION\_ADDRESS\_VAD\_PROBE
- KERNEL\_THREATINT\_KEYWORD\_EXECUTION\_ADDRESS\_MMF\_NAME\_PROBE
- KERNEL\_THREATINT\_KEYWORD\_READWRITEVM\_NO\_SIGNATURE\_RESTRICTION
- KERNEL\_THREATINT\_KEYWORD\_DRIVER\_EVENTS
- KERNEL\_THREATINT\_KEYWORD\_DEVICE\_EVENTS

## Appendix B

# ThreatCheck

The ThreatCheck tool [79] tests whether Defender Antivirus [56] identifies a binary as malicious. If yes, it splits the binary into chunks to pinpoint the sequence of bytes that triggered the detection. If we analyze the very popular Mimikatz tool [14] using ThreatCheck, we see that the file is signed. Indeed, the bytes at 0x319 trigger a detection, as visible in Listing B.1.

```
> .ThreatCheck.exe -e Defender -f .\mimikatz.exe
[+] Target file size: 1355264 bytes
[+] Analyzing...
[!] Identified end of bad bytes at offset 0x319
00000000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2E
00000010  74 65 78 74 00 00 00 09 F7 0C 00 00 10 00 00 00
00000020  F8 0C 00 00 04 00 00 00 00 00 00 00 00 00 00 00
00000030  00 00 00 20 00 00 60 2E 72 64 61 74 61 00 00 68
00000040  77 06 00 00 10 0D 00 00 78 06 00 00 FC 0C 00 00
00000050  00 00 00 00 00 00 00 00 00 00 00 40 00 00 40 2E
00000060  64 61 74 61 00 00 00 50 78 00 00 00 90 13 00 00
00000070  6A 00 00 00 74 13 00 00 00 00 00 00 00 00 00 00
00000080  00 00 00 40 00 00 C0 2E 70 64 61 74 61 00 00 10
00000090  68 00 00 00 10 14 00 00 6A 00 00 00 DE 13 00 00
000000A0  00 00 00 00 00 00 00 00 00 00 00 40 00 00 40 2E
000000B0  72 73 72 63 00 00 00 E8 3F 00 00 00 80 14 00 00
000000C0  40 00 00 00 48 14 00 00 00 00 00 00 00 00 00 00
000000D0  00 00 00 40 00 00 40 2E 72 65 6C 6F 63 00 00 40
000000E0  25 00 00 00 C0 14 00 00 26 00 00 00 88 14 00 00
000000F0  00 00 00 00 00 00 00 00 00 00 00 40 00 00 42 00
```

Listing B.1: Output of ThreatCheck version 1.0.0 when run against Mimikatz version 2.2.0 20220919 with Defender security intelligence version 1.404.324.07.

As another example, we provide in Listing B.2 the output of the tool when run against the original

Threadless Injection implementation. In this case, it is clearly visible that the strings intended to be output in the command line are signed. This means that by removing them, detection due to this signature can be prevented.

```
[+] Target file size: 60416 bytes
[+] Analyzing...
[!] Identified end of bad bytes at offset 0xC7C4
00 3A 00 78 00 7D 00 00 41 5B 00 21 00 5D 00 20  ··x·}·A[·!·]·
00 46 00 61 00 69 00 6C 00 65 00 64 00 20 00 74  ·F·a·i·l·e·d· ·t
00 6F 00 20 00 77 00 72 00 69 00 74 00 65 00 20  ·o· ·w·r·i·t·e·
00 70 00 61 00 79 00 6C 00 6F 00 61 00 64 00 3A  ·p·a·y·l·o·a·d·:
00 20 00 7B 00 30 00 7D 00 00 3B 5B 00 21 00 5D  · ·{·0·}··;[·!·]
00 20 00 46 00 61 00 69 00 6C 00 65 00 64 00 20  · ·F·a·i·l·e·d·
00 74 00 6F 00 20 00 70 00 72 00 6F 00 74 00 65  ·t·o· ·p·r·o·t·e
00 63 00 74 00 20 00 30 00 78 00 7B 00 30 00 3A  ·c·t· ·0·x·{·0·:
00 78 00 7D 00 00 7B 5B 00 2B 00 5D 00 20 00 53  ·x·}··{[·+·]· ·S
00 68 00 65 00 6C 00 6C 00 63 00 6F 00 64 00 65  ·h·e·l·l·c·o·d·e
00 20 00 69 00 6E 00 6A 00 65 00 63 00 74 00 65  · ·i·n·j·e·c·t·e
00 64 00 2C 00 20 00 57 00 61 00 69 00 74 00 69  ·d·,· ·W·a·i·t·i
00 6E 00 67 00 20 00 36 00 30 00 73 00 20 00 66  ·n·g· ·6·0·s· ·f
00 6F 00 72 00 20 00 74 00 68 00 65 00 20 00 68  ·o·r· ·t·h·e· ·h
00 6F 00 6F 00 6B 00 20 00 74 00 6F 00 20 00 62  ·o·o·k· ·t·o· ·b
00 65 00 20 00 63 00 61 00 6C 00 6C 00 65 00 64  ·e· ·c·a·l·l·e·d
```

Listing B.2: Output of ThreatCheck version 1.0.0 when run against the original Threadless Injection [10] implementation with Defender security intelligence version 1.405.477.00.



## Appendix C

### CAPA

CAPA [42] uses static analysis to detect capabilities in executable files. It leverages its own rule set described using Yara [98]. A truncated example output of running CAPA against Mimikatz [14] is given at Table C.1. The detection capabilities it provides is not based on byte signatures, but on behavior signatures. This means that the exact binary content does not matter, since it considers code structure, imported functions, and other elements instead.

Capability	Namespace
acquire credentials from Windows Credential Manager	collection
gather chrome based browser login information	collection/browser
parse credit card information	collection/credit-card
reference SQL statements (3 matches)	collection/database/sql
get domain trust relationships	collection/network
resolve DNS	communication/dns
connect network resource (2 matches)	communication/http
create pipe	communication/named-pipe/create
listen for remote procedure calls (2 matches)	communication/rpc/server
validate payment card number using luhn algorithm	data-manipulation/checksum/luhn
compress data via WinAPI	data-manipulation/compression
decode data using Base64 via WinAPI (6 matches)	data-manipulation/encoding/base64
encode data using Base64 via WinAPI (2 matches)	data-manipulation/encoding/base64
encode data using XOR (20 matches)	data-manipulation/encoding/xor
...	...
parse PE header (2 matches)	load-code/pe
resolve function by parsing PE exports (24 matches)	load-code/pe
persist via Windows service	persistence/service

Table C.1: Truncated output of running CAPA 6.1.0 against Mimikatz version 2.2.0 20220919.