UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale in Software Security

Laccolith: A Novel Approach for Anti-Detection in Adversary Emulation

Anno Accademico 2021/2022

relatore
Prof. Roberto Natella

correlatore
Ing. Vittorio Orbinato

candidato
Marco Carlo Feliciano
matr. M63001136

[Dedica]

Table of contents

1 Introduction

Offensive security is a proactive approach to protecting computer systems, networks and individuals from attacks, as opposite to “conventional security” which focuses on reactive measures. The idea of offensive security is to anticipate breaches and attacks instead of (only) reacting to them.

The “traditional” offensive security practice is penetration testing. It is still widely adopted, but it does not effectively catch all the behaviour of an attacker, because its target is to discover and identify vulnerability, with little focus on post-compromise situations [2]. Other practices used in professional red teaming engagements are threat hunting and adversary emulation, both based on threat intelligence sources [3].

Adversary emulation can be defined as "a type of red (or purple) team engagement that uses real-world threat intelligence to impersonate the actions and behaviours that your red team (or bad actors) would use in practice" [1].

Adversary emulation is important for red teams to automate their activities in a more effective way, and it is important for blue teams to train themselves to identify and fill attacks. In particular, it covers the requirement of simulating the actions an attacker would take in a post-compromise situation. There are many popular frameworks available for adversary emulation, one of the most notable being MITRE’s Caldera [4].

An important part of the attacker’s techniques is *anti-detection*. In real scenarios, skilled attackers pay attention to hide their traces, in order to avoid being detected by security teams and products (e.g., EDRs). As a result, attackers and defenders are involved in a “cat-and-mouse” game, where more robust detection techniques are continuously developed to respond to new anti-detection techniques.

The aim of this research is to study the detectability of current state-of-the-art solutions for adversary emulation, to show that they are not able to emulate a skilled attacker in presence of AV/EDR solutions, and then to propose a novel approach with anti-detection capabilities. Everything will be done considering Windows 10 as target.

The research about the detectability of current state-of-the-art solutions takes *Sandcat* as example, which is the most popular Caldera agent, and separately considers the case in which Sandcat was successfully injected (section 3.2.1) and the case in which it must be injected, trying to evade detection by Windows Defender (section 3.2.2).

It is considered Caldera as only sample because other adversary emulation tools either have an architecture similar to Caldera's one (in particular for the “agent deployment” phase) or are not complete solutions (for example, they need to be integrated with Caldera as a downloadable executable).

The following picture is a simplified view of the analyzed architecture.

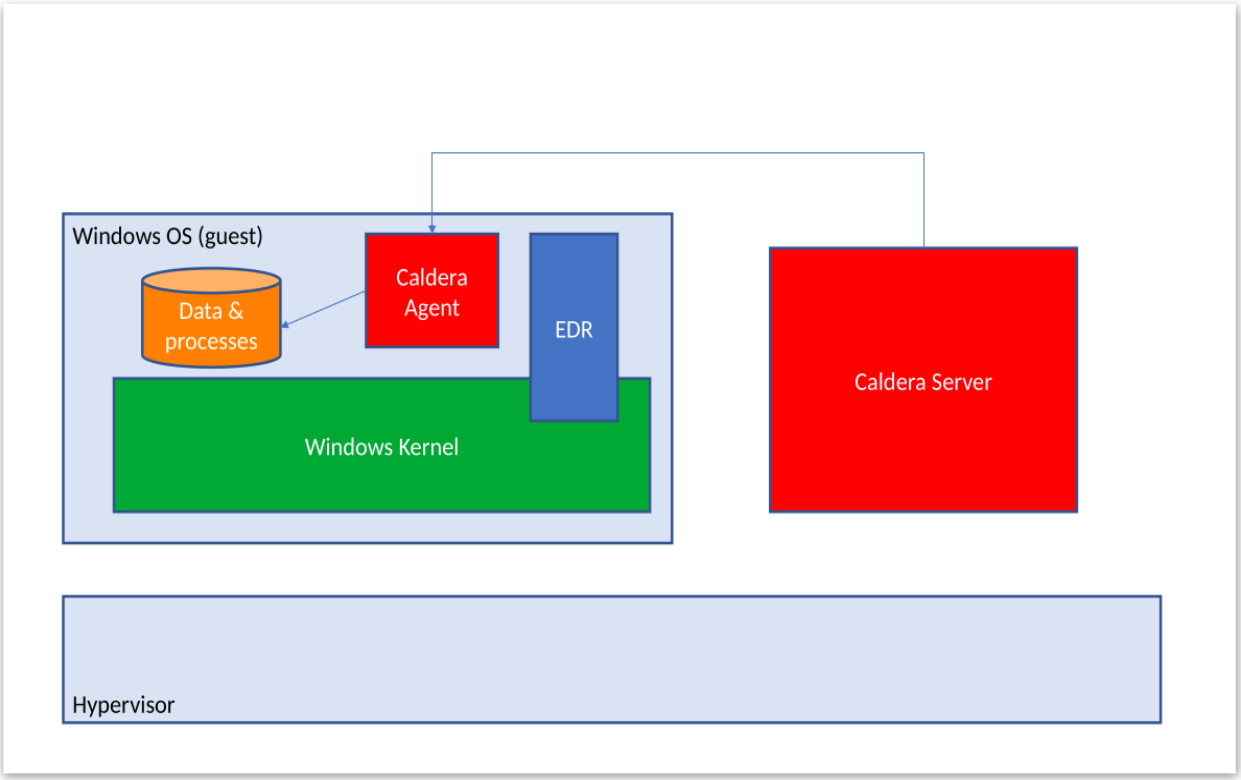

Figure 1: Experiment architecture with Caldera

After having showed that Caldera does not have real defense evasion capabilities, the next step is to design and implement the novel approach.

The novel approach is about injecting a high-privileged agent from the hypervisor, to automate the deployment phase and address the problem of the detectability. It will also be argued, in section 2.2, that doing so corresponds to a real threat, that is a hypervisor post-compromise situation, so it is an effective adversary emulation scenario. Therefore, the part of the system whose aim is to deploy an agent to the target system will usually be referred as the “attacker” and the set of channels used by the attacker to inject the agent in the target system will usually be referred as attacker’s “gained privileges”.

So, the second part of the research is about showing that it is possible to inject an agent in this situation, implement some functionalities in the agent to permit the implementation of some adversary profiles, develop a framework around it to automate its injection and to show that it is suitable for effective adversary emulation and, finally, study the detectability of this approach in presence of AV/EDR solutions, to make a comparison with the previous approach.

1.1 Thesis contribution

Thesis' contribution is the sum of two parts: experiments about state-of-the-art adversary emulation frameworks to show their lack of defense evasion capabilities, and design, implementation and experiments about the novel approach.

The next chapter is all about the second part of the research, namely the design and implementation of the previously mentioned novel approach, whereas all experiments can be found in chapter 3.

The novel approach needs three core "components" to be implemented:

- Injection: it is the first step in the post-exploitation process, in which the attacker uses the gained privileges to load an Agent capable of adversarial actions;
- Agent: it is a thread of execution which is deployed on the compromised host and connects with the C2 server to wait for commands;
- Framework: it is the infrastructure that brings together the Injection and the Agent, by automating the process and by providing the C2 server.

2 The proposed solution

In section 2.1, the requirements of the components described in section 1.1 will be defined.

Sections 2.2 and 2.3 will show the design of the solution and how the design guided the implementation.

Sections 2.4 and 2.5 provide key technical details about the implementation.

2.1 Requirements

2.1.1 Injection

The injection of the high-privileged agent must be performed from the hypervisor's privilege.

The injection should leverage full read/write access to the Windows 10 VM's RAM in real-time, like an attacker that was able to gain arbitrary read/write access after a successfully exploit. In addition to that, since the hypervisor is external to VM-specific page mechanisms, it has the R/W access bypassing page permissions.

This post-exploitation phase can also be simulated with the help of virtual machine introspection (VMI) techniques, for example with Volatility [6]. This is to have visibility of kernel symbols of the target VM.

After the successful injection of the agent, the VM must continue its operations, that is it must not crash.

2.1.2 Agent

The injected agent must have defense evasion capabilities, effective against Windows Defender and other common AV/EDR solutions. Therefore, it must run with high privilege, ideally in the kernel.

It must connect to a command-and-control (C2) server and be ready to execute commands it receives and provide output.

It must implement a set of commands complex enough to allow the implementation of some adversary profiles.

It must be able to terminate gracefully, without making the system crash, when the C2 server sends a “close” command to it.

As a plus, it would be better if it were able to execute certain actions both in a stealth way and in a less-stealth way, to offer the possibility of configuring the evasion capabilities. This is to make a specified set of activities easier to detect during a purple team engagement.

2.1.3 Framework

The proposed solution should come with a complete framework to orchestrate the injection and control of malicious actions. This framework has many functions.

First, it needs to implement the automation of the injection, using the access to the VM’s RAM and with the help of VMI techniques, like mentioned before. In the injection automation sub-system, it must be capable of changing some run-time parameters of the injected payloads. In addition to that, it must be able to automatically detect available domains (the terms “virtual machine” and “domain” will be used interchangeably) to inject, and it should be decoupled to the underlying virtualization technology.

Second, it needs to implement a C2 server to receive connections, store information about them and make them available, handle multiple connections concurrently (like Metasploit, a penetration testing framework also used for adversary emulation [7]), be able to conduct complex sequences of commands automatically (like Caldera’s “Autonomous red-team engagements” [8]).

Third, it needs to offer one or more interfaces to handle:

- Injection of domains.
- Access to C2 functions, such as listing received connections, sending a command to an agent, reading the output of a command, starting an autonomous operation.

- Customization of some parameters, for example choosing which payload to inject.
- Basic file upload and download functions: the upload can be useful for custom operations, whereas the download is useful for exfiltrated files.

These functions can be split into more than one interface, with the idea that some functions can be more user-friendly in another type of interface rather than making all of them fit together in an unique interface.

Fourth, it needs to be easy to deploy and to customize.

2.2 Architecture for the injection

The idea for the novel approach came from a Proof-of-Concept exploit by Google Project Zero [5]. The idea is to consider a scenario in which the attacker was able to compromise a hypervisor with a 0-day exploit and wants to leverage the gained arbitrary read/write access to any virtual machine's RAM to inject an agent in a virtual machine. The agent would then have a very high privilege and use it to evade defenses. The existence of this type of exploits justifies the usage of virtualization technologies for adversary emulation, by performing an injection over channels that are not usually monitored by AV/EDR solutions.

The component responsible for the injection ("injector") must have read/write access to the VM's RAM in real-time, and it should be decoupled from the virtualization technology used. Therefore, the idea is that this component should access the RAM not by using hypervisor-specific APIs, but by using common APIs.

Note that the term "hypervisor" can refer either to the Virtual Machine Manager (VMM, such as VMWare) or to the underlying operating system on which the VMM runs, depending to the sentence.

Many hypervisors offer the possibility of mapping the RAM of a VM to a file. So, the setup to do is to configure the VMM in such a way that the VM's RAM is exposed as a file, at this point the injector has "physical access" to VM's RAM: an offset in the file corresponds to a physical address.

Since there could be performance issues with this design choice, it is possible to leverage virtual file system's functions to reclaim performance. In particular, in Linux, the */dev/shm* sub-folder implements the concept of shared memory [9]: making the RAM-file reside

there is equivalent of having that file internally mapped to hypervisor's RAM.

To summarize, the VM's RAM is mapped to a file on the hypervisor's file system instead of being mapped directly to the hypervisor's RAM (which is the physical RAM, except in the case of nested virtualization), and that file is placed in a "place" in the virtual file system which is mapped to the hypervisor's RAM. So, at last, the VM's RAM is still backed by hypervisor's RAM, but it is easily accessible as a file.

it is useful to document here the actual setup made for development and for experiments. The hypervisor was *qemu-system-x86_64* [10], running on a Linux host, managed by *Libvirt* [11] and leveraging the *KVM support* offered by the CPU [12]. It was used *gnome-boxes* as front-end; it generates an XML file to describe the domain, to manage its settings and to have "pointers" to its files (for example the hard-disk *qcow2* image).

To map the VM's RAM to a file, Qemu uses the *ramfs* protocol, which is described in Linux documentation [13]. To enable it, it is compulsory to modify the generated XML file (for example with the command "virsh edit <domain>"), acting on the "memory backing" part. The official documentation of the XML format is available in the previously referenced Libvirt's website but, since this research has a *GitHub* repository associated with it, technical setup instructions can be found directly there [14].

Like highlighted in the instructions, after having correctly modified the XML file, the necessary "trick" to reclaim performance is either changing the default directory for file-backed RAM setting it to */dev/shm* or making that directory a symbolic link to */dev/shm*.

The following picture summarises the obtained situation.

Figure 2: Architecture for the injection

Since the agent is very likely to execute in the kernel, and since this framework is inspired by Caldera, it will be called Laccolith and referred to as such from now on. To maintain the parallelism with Caldera, since it gave inspiration and since it is still an adversary emulation tool, it was chosen to use a name recalling volcanic phenomenons. In fact, a "caldera" is a volcanic lake [54], whereas "a laccolith is a body of intrusive rock with a dome-shaped upper surface and a level base, fed by a conduit from below. A laccolith forms when magma (molten rock) rising through the Earth's crust begins to spread out horizontally, prying apart the host rock strata. The pressure of the magma is high enough that the overlying strata are forced upward, giving the laccolith its dome-like form." [55]. The parallelism is that the laccolith is at a "lower level" than a caldera, like Laccolith's agent that executes in the kernel as opposed to Caldera's agent that executes in user-space.

2.3 Injection chain

This section is about the design and implementation of the injection chain.

The injection chain is the set of necessary steps that, starting from arbitrary read/write access to VM's RAM, allow the execution of an agent with the previously mentioned requirements.

2.3.1 The semantic gap

After having obtained arbitrary read/write access, the next step is to obtain arbitrary code execution, in a high-privileged context.

The general way of getting arbitrary code execution using arbitrary read/write is corrupting some data structures in memory to redirect execution flow after that the code will, eventually, use these data structures.

Anyway, this approach has the problem of the "semantic gap": when dealing with raw memory, the "injector" sees physical addresses, whereas the code sees virtual addresses. In addition to that, the code does not bypass the MMU, so it always must reference valid virtual addresses. So, the injector can overwrite any data structures it wants but does not have an allocated space to place its code into.

To be clearer, let's consider an example.

When using Volatility to get the addresses of all system calls (using the plugin *windows.ssdt*), the plugin returns virtual addresses for them. This is because it starts from *ntoskrnl* module and, from there, finds the address of *KeServiceDescriptorTable* [15]; from that table, it computes the other virtual addresses. So, for system calls it only provides virtual addresses.

On the other hand, when looking for modules (using the plugin *windows.modules*) it does raw memory lookups searching for certain headers, so it provides the physical addresses; it then also associates virtual addresses to these physical addresses, using a binary search (again, Volatility's GitHub repository is the reference).

The injector, to do its job, needs both physical and virtual addresses. In fact, it can not "write to a virtual address" because it must write to the RAM's live image at a certain offset, and this offset is a physical address. In addition to that, it must ensure that both read and writes it performs are not-cacheable.

To address this problem, it was used an utility called *shmem2* [16] (inspired by the famous *devmem2* used in Linux to implement user-space drivers). This utility was then modified and compiled as a shared object, to be used as an external API in *Python* to implement custom read/write logic.

In particular, to close part of the semantic gap on system calls, it was implemented a function to search a physical address for the given virtual address, knowing the expected bytes at that address. The idea is that, if a sufficient number of bytes of the system call's code is considered, then it is likely to be an unique sequence. It is still compulsory to know the virtual address, because the least significant 12 bytes of the virtual address are equal to the least significant bytes of the physical address and are known as *page offset*. [17]

The other shade of the semantic gap is, like previously mentioned, that the injector can not "allocate" space to use for the injection, or rather can not write to unallocated memory and pretend that a thread obtains a valid reference to that memory, because the injector bypasses all the paging mechanisms, whereas the system does not. For sure, it could do weird and complex manipulations to page tables in memory and, if needed, to process

control blocks, to obtain the effect of making the virtual addresses already used by some processes resolve to injector-controlled physical addresses. This is a different type of attack called “page table remapping attack”, and can even be used to bypass security mechanisms of “enclaved execution” [18].

But this approach is over-complicated for the purpose of code execution, in particular if the injector bypasses paging mechanisms. In fact, the semantic gap also gives the injector an advantage: it can use a different memory corruption strategy.

Usually, text areas of processes (where the assembly code is) are not-writeable pages. But, since the memory access is done from the hypervisor, any byte of memory is readable and writeable. So, it is possible to directly overwrite some code which will be eventually called and use it to inject an agent.

The overwritten code will be referred to as “injection point” from now on.

Now there are two open questions:

- which code to overwrite (with high privileges) and (sub-question) how to find it automatically to automate the injection?
- what to inject there and (sub-question) how to handle the fact that it can be called multiple times?

The next sections will answer to these questions, and the last section is about the engineering of the overall injection chain.

2.3.2 Finding an injection point (“where”)

In this section, it will be given answer to the first question.

In particular, since the problem of automatically finding a system call has already been

addressed with the combination of Volatility and the shmem2 shared object, a first target as injection point could be a generic system call.

Before diving into that, it was useful to understand the process involved when a system call is issued from user-mode [19]. Furthermore, since the injection point needs to be easily found/adapted for different Windows versions, it is also useful to have a view of Windows x86-64 system calls by version [20].

System calls are a good fit to be injection points because they get eventually called by system processes.

In addition to that, it is important that the chosen injection point is called with the right hardware priority [21]. This is because, once the agent is executing in kernel mode, to perform useful actions it is necessary to use Windows' internal kernel functions, in particular functions available to drivers [22]. From the documentation, it is easy to see that most of the functions require to be called in *PASSIVE_LEVEL*, and also by doing some reverse engineering in kernel debug it can be seen that the code of these functions performs the necessary checks to ensure that it is called with the right Interrupt Request Level (IRQL).

The hardware priority at which a code executes depends on "who" calls that code: it is "inherited". When an operating system starts, it executes at the highest priority and with the highest privileges. To change its priority and lower its privileges, it must modify certain status registers of the processor. These registers can be changed only with a certain privilege, this means that a "thread" of execution can "drop" its privileges, but can't reclaim them. So, there are special instructions that allow the call of high-privileged functions. The "handlers" of these special instructions must be setup before dropping the privileges. This is also true in Windows. In particular, on an abstract level, the module

with the highest privileges and the highest priority is the *dispatcher*, responsible for the execution of all the threads (both kernel and user). That's why its hardware priority is called *DISPATCH_LEVEL*. But there can also be other threads of execution executed at DISPATCH_LEVEL: handlers of hardware interrupts. Since this is the highest priority level, it can't be interrupted, even by the dispatcher itself (because the dispatcher runs at the same priority). Therefore, the handler of a hardware interrupt, like the routine associated to a timer, must do its duty "fast" and so can't, for example, issue blocking calls. By disassembling any Windows kernel function available to drivers, that can't be called at DISPATCH_LEVEL, it's easy to see that the first instructions perform a check about the IRQL at which the function is called, and raise a "kernel exception" if the IRQL is higher than it's expected to be. There are many intermediate IRQLs in the kernel, there are the hardware priorities for user-mode threads and there are the hardware priorities for real-time user-mode threads. So, it's clear that the dispatcher and handlers of hardware interrupts have the highest priorities, then there is a mix of priorities of kernel threads and user threads. In fact, most of the kernel code has a lower priority then user code, because the system needs to be interactive and because most of the kernel code handles I/O with file system, network and other blocking systems. The IRQL at which this kernel code runs is the previously mentioned PASSIVE_LEVEL. There are even lower levels than the passive one, but, to not wander, it's just important to know that in PASSIVE_LEVEL it's possible to do blocking calls and, so, all of the Windows' kernel API is available.

Back to system calls, it is easy to see, by putting a breakpoint on a system call while doing kernel debug, that they are called at PASSIVE_LEVEL. In fact, after the breakpoint is hit, it is possible to inspect the debugger-saved IRQL and see that its value is 0 [23]. This confirms the hypothesis that system calls are a good fit for the injection.

At this point, the next step is to choose a system call to overwrite, with "enough space" to contain the "malicious code" (*space requirements*); the malicious code also needs to

handle the case of being called multiple times and needs to somehow exit gracefully without hanging forever, because user-space processes expect the system to return after a system call (*behaviour requirements*).

it is already clear that, for space requirements and for behaviour requirements, the "malicious code" which overwrites the system call can not be the "agent" itself but will be responsible for the allocation of the code region for the agent and the asynchronous execution of the agent in that area.

For this reason, the injection is a multi-stage process, and the "malicious code" will be also referred to as "first-stage shellcode" from now on and the "agent" will be also referred to as "second-stage shellcode".

To choose the actual system call to overwrite, let's define the meaning of "linear region of code" in this context. A linear region of code is a memory region containing instructions, that can be called like a function but that does not contain other functions (so it can only be called at its starting address) and every function called from the linear region must belong to the same linear region (so that code that does not belong to the linear region does not return in the middle of the linear region); in addition to that, it must be contained all in a single memory page, without crossing page boundaries.

Ideally, the overwritten code should be a linear region of code, so it should satisfy the three mentioned requirements of a linear region of code.

The third requirement is necessary because otherwise the injection can not be done reliably (contiguous pages in physical memory do not correspond to contiguous addresses in virtual memory, in general).

The first requirement is also necessary because otherwise the code would be called “in the middle” and there would be errors like invalid opcodes or bad memory accesses.

The second requirement is a nice-to-have: the impact of a “return in the middle” is the same of the previously mentioned “call in the middle”, and its probability goes down to zero if the second requirement is satisfied; but, since there is also the possibility of race conditions (overwriting the linear region of code when it is already in use), it is okay to do not satisfy the second requirement, resulting in a “semi-linear region of code”.

The chosen injection point must be at least a semi-linear region of code, better if it is linear. In addition to that, the region of code overwritten must be big enough to contain the whole first-stage shellcode.

A good candidate found is the system call *NtQueryVirtualMemory*, or rather, to be more precise, the function it calls, located 64 bytes later (at least in different Windows updates of build 19044): *MmQueryVirtualMemory*. It is large around 3800 bytes and it is linear.

it is better to optimize the first-stage shellcode, anyway, to use as least of these bytes as possible, to be less intrusive.

After having found a conceivable injection point, let’s jump to the opposite side of the problem: instead of going bottom-up thinking about the first-stage shellcode, let’s think about the second-stage shellcode.

In fact, the second-stage shellcode is “what” needs to be injected: the actual agent, whereas the first-stage shellcode is a ring in the injection chain and provides the “how” the agent is going to be executed.

At this point, it is already clear what does the architecture for the novel approach looks like. it is depicted in the following diagram.

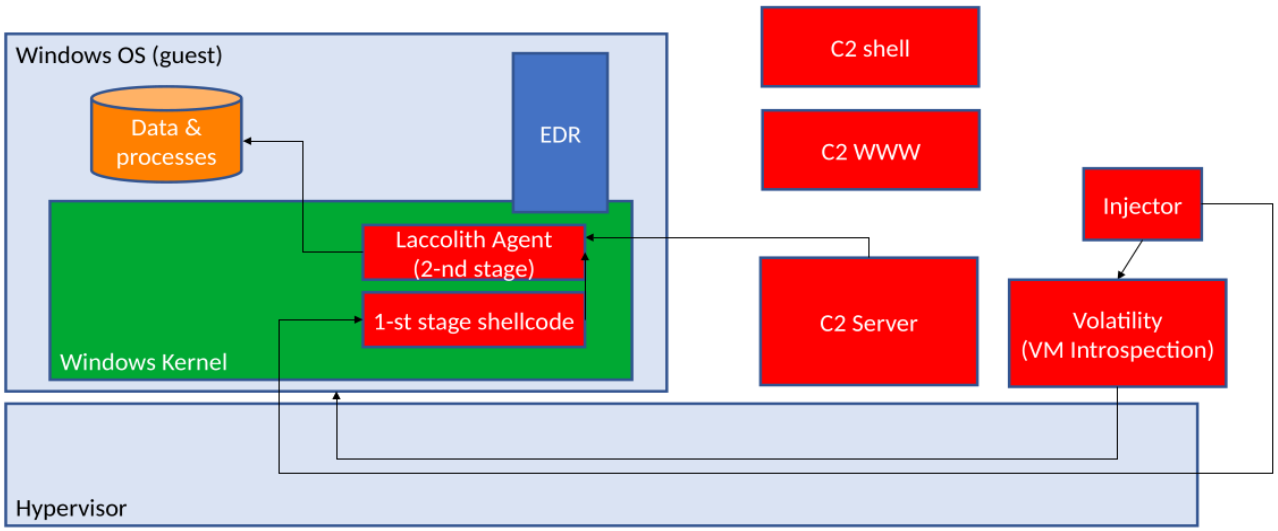

Figure 3: Experiment architecture with injection from hypervisor

2.3.3 Second-stage shellcode (“what”)

The second-stage shellcode is an agent, or rather it must open a connection to a C2 server and implement a set of commands, to perform the actions required by some adversary emulation campaigns. Furthermore, it executes in kernel mode. In this way, the second stage shellcode (but also the first stage) is very similar to a rootkit.

Much research and browsing were about finding something similar to a rootkit, but not too hard to modify and extend. With modifiability in mind, the best resource proved to be a project called *SassyKitdi* [24] by a researcher called Sean Dillon, better known as *zerosum0x0*; he was the first to reverse engineer the DoublePulsar SMB backdoor, used for EternalBlue exploit [25].

A big part of this research is about the modular development of the Agent itself; it would not have been possible without using SassyKitdi as a starting point.

What does it do?

Basically, it creates a reverse TCP socket using Windows’ Transport Driver Interface (TDI), then it sends Windows version and the memory dump of lsass.exe process over the wire; on the other end, a minidump is created out of received data.

The interesting thing, which makes this project more appealing than kernel-level backdoors like DoublePulsar, is the fact that SassyKitdi is not just an assembly payload, it is a Rust library organized in such a way to produce a position-independent shellcode.

In fact, by fine tuning the Rust compilation process (with cross-compilation to a Windows DLL), by disabling the usage of the standard library, by using a certain programming style and by using some useful abstractions (like specifying "extern" functions with "stdcall"

convention to call Windows kernel's internal functions, or by using inline assembly to implement certain functions), it is possible to use a high-level language and reliably obtain a working shellcode by extracting the startup section of the compiled DLL.

The Rust library contains definitions of Windows constants, enums, structs, signatures of functions, wrappers for complex I/O operations (in particular for IRPs [26]), some rootkit techniques rewritten in Rust (like a technique to find *ntoskrnl* base address, and the dynamic and "stealth" resolution of functions' addresses from the export table [27]), wrappers built upon the previously mentioned wrappers and the top module which compile itself as a shellcode (doing everything in the text area, without global area).

So, to build the second-stage shellcode, at this point it was necessary to understand the library, modify the code, extract it from the compiled DLL using the provided utilities, and test it.

Without having the multi-stage injection chain ready, the only way to test it was to use a driver to load kernel shellcodes. It was used a driver developed by FireEye, called kscldr [28]. In addition to that, since a kernel shellcode could crash the system, it was setup an environment with kernel debug, using KDNET. Actually, the kernel debug environment had already been previously set to study the behaviour of system calls.

The overall architecture of the test and debug environment is shown in the following diagram.

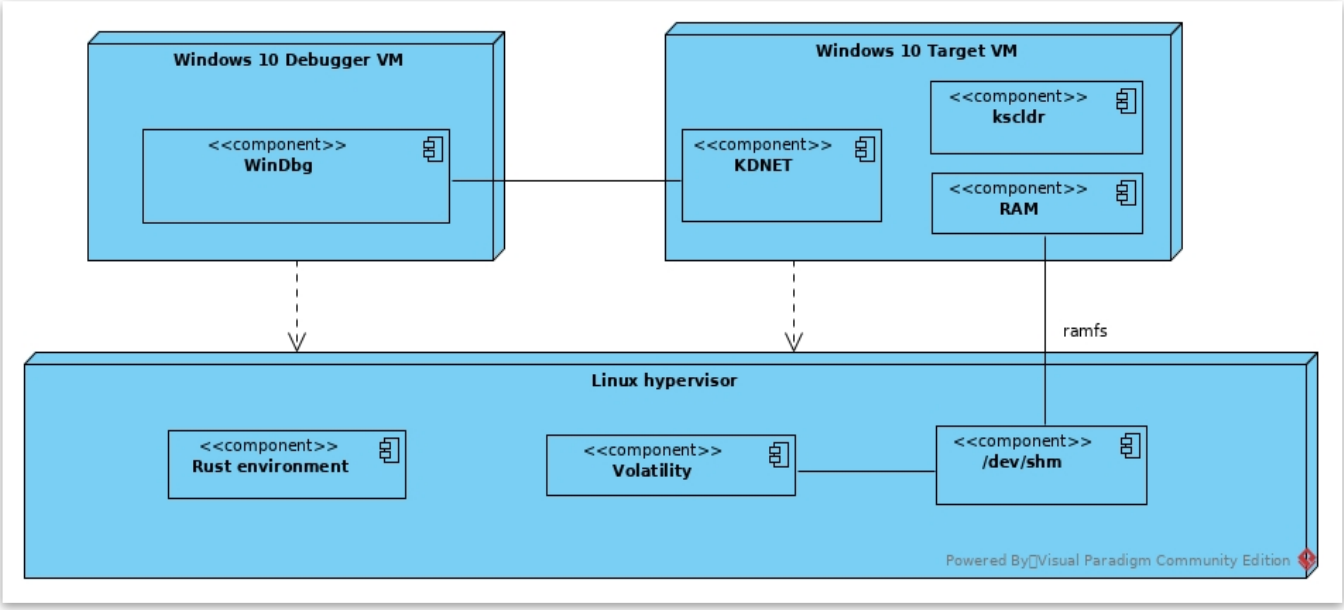

Figure 4: Test & debug environment

It took a significant effort to go from the original Rust code to a basic agent with "echo" as only command, because:

- Rust code was not friendly, so the first thing was to understand it and the tricks that made the compiled code position-independent;
- SassyKitdi was tested with KVA Shadowing [29] disabled, and its technique to find ntoskrnl base address did not work in presence of the KVA Shadowing's page guards: the solution was to use the hypervisor privilege to leak ntoskrnl base address (using Volatility's *windows.modules* plugin) and patch the shellcode bytes before loading/injecting the shellcode;
- TDI operations had a hard-coded timeout of 1 second, which is too low for something executing in PASSIVE_LEVEL: the naive solution was to increase the timeout to 15 seconds;
- the socket's receive handler was only a stub always sending ACK, so it took a bit of reverse engineering to understand how to modify it and how to develop a synchronization protocol between the shellcode's "main" and the receive handler (which executes at DISPATCH_LEVEL);

- the shellcode did not terminate gracefully.

After having solved all the previous problems, it came the implementation of some commands in the second-stage shellcode, either by re-using/extending code in some "common" crates (a "crate" is a Rust package) or by creating new crates (for example for the management of the file system and of the system registry).

It was also compulsory to implement some functions for a basic parsing of received commands, always with the requirement of the position-independent compilation (so it was not possible to use strings, because they would have gone in the global area).

For the configuration of the evasion capabilities, it was also necessary to implement a command to execute some code in user-mode, i.e., a "ring0-to-ring3" function; this is a common problem in rootkits, not because they want to be detected but because they want to have access to higher-level functions available in user-mode [30]. Many approaches were explored (see the chapter "Future developments"), but at last the related command was left as stub, because it needs to be portable and reliable (as defined in the related section).

The following diagram shows the list of implemented commands in the second-stage shellcode.

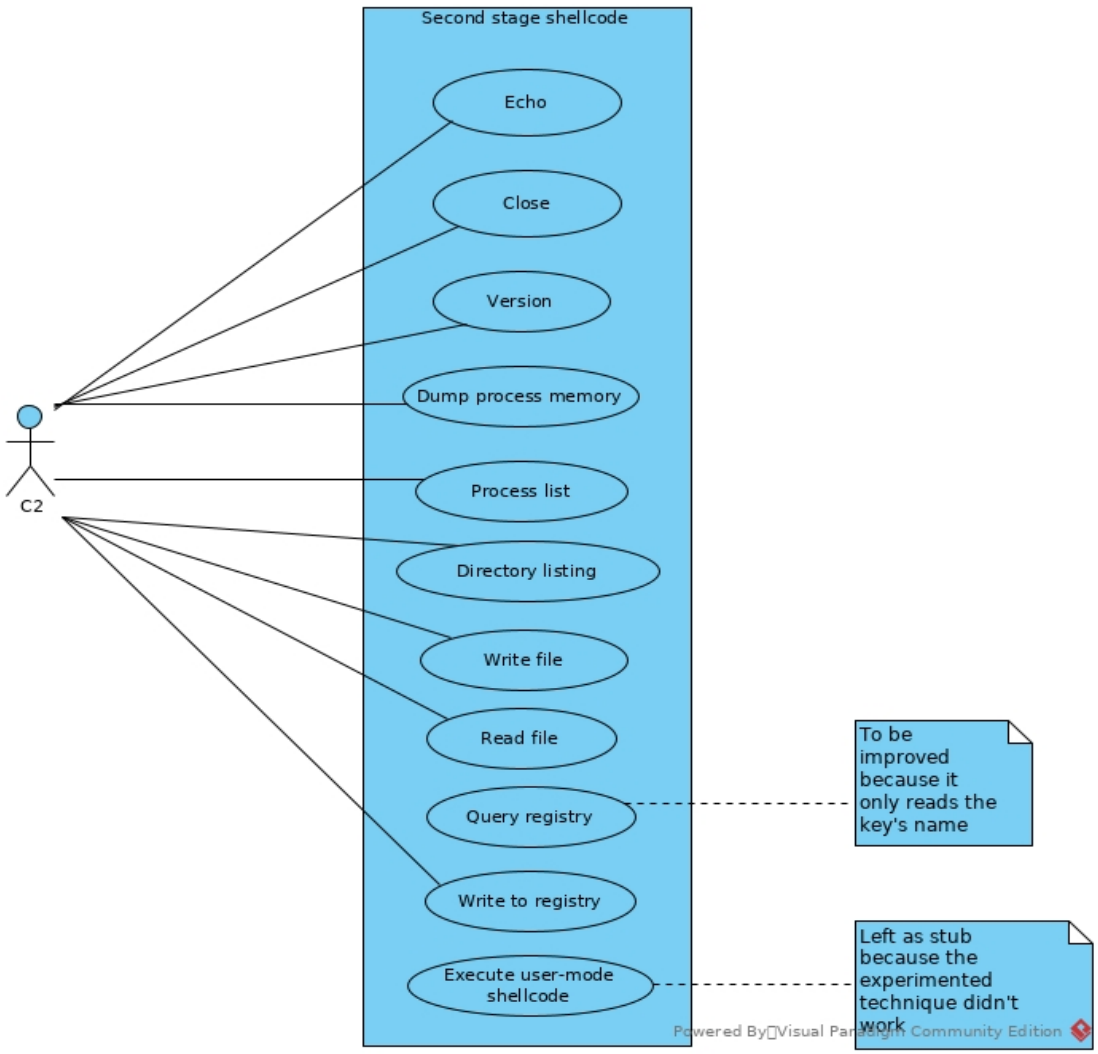

Figure 5: Commands implemented in second-stage shellcode

At this point, the actual agent has been implemented, but it has yet to be loaded in an automated way using the hypervisor's privilege, rather than with a driver.

So, the next step is the implementation of the first-stage shellcode, executed after the injection from the hypervisor and responsible for the reliable loading of the second-stage shellcode.

2.3.4 First-stage shellcode ("how")

Let's summarise, from a high-level perspective, the technical details of the first-stage shellcode.

The first-stage shellcode will overwrite a system call for a short period of time, long enough to "communicate" to the hypervisor that the area for the second-stage shellcode is ready; the hypervisor will write the second-stage shellcode to that area, then the first-stage shellcode will start a "system thread" to execute the second-stage shellcode and will return.

Other concurrent calls to the system call will immediately return, using some synchronization pattern. After the start of the second-stage shellcode, the original system call code can be restored.

For convenience, the first-stage shellcode is developed using the Rust library, as well.

In the following diagram there is the hierarchical structure of the resulting “kernel shellcode library”, in which the leaves are the top-level modules that get compiled, and the other nodes are the “common crates”.

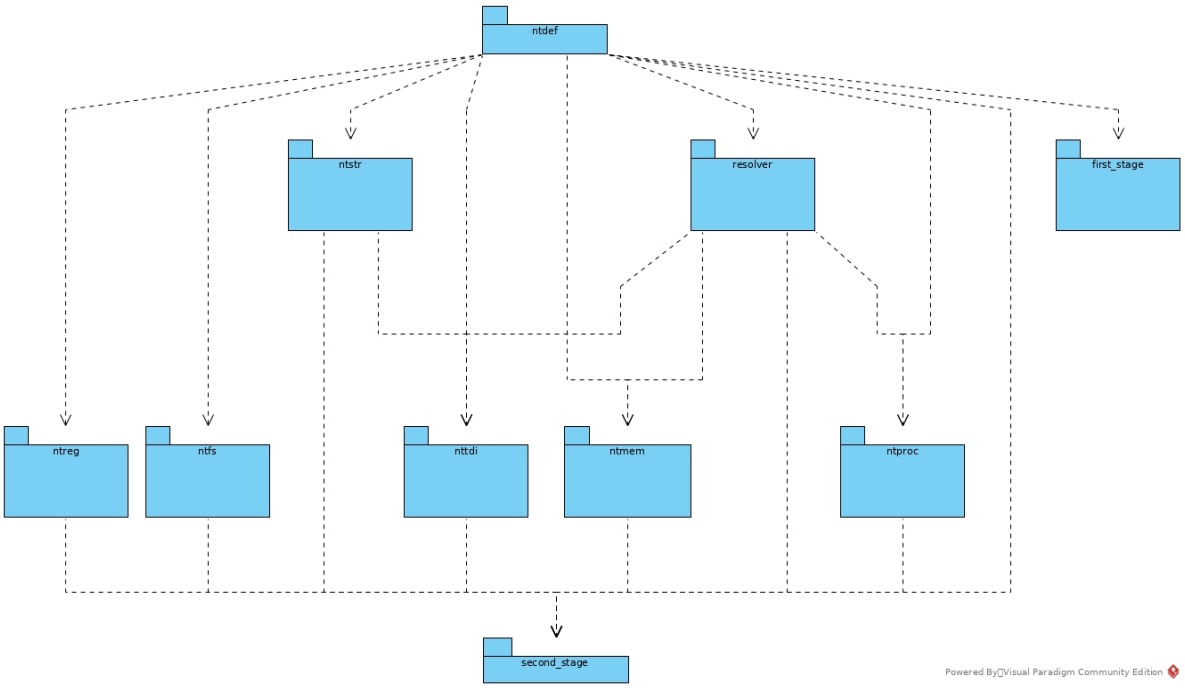

Figure 6: Kernel shellcode library package diagram

Now, all the technical perks of the first-stage shellcode will be described, since it was developed from scratch.

The first-stage shellcode needed to be much shorter than the second-stage, so instead of finding ntoskrnl's exported functions at run-time by iterating on the module's export table (like the second-stage shellcode does), it uses hard-coded offsets to ntoskrnl base address. The portability will be easily implemented in the automation tool with a "version" configuration parameter and hard-coded offsets for a set of versions, to patch shellcode bytes at run-time before injection or to raise an exception if it is selected a version for which the portability is missing.

Since the injector sees physical addresses and the "final" second-stage shellcode is bigger than the size of a page (~10 KB against 4 KB), the first-stage shellcode needs to allocate contiguous memory for the second-stage; this is accomplished using the function *MmAllocateContiguousMemory*, from WDK [22].

So, it allocates 16 KB of contiguous memory, and writes a "very random cookie" to it, whose value is known by the injector. This "cookie" is called "egg" because the injector must search it and overwrite it with the second-stage shellcode, which is bigger, so "the egg hatches".

The first-stage shellcode periodically reads the memory in which the egg is located to see if it has changed; if it has changed, then the second-stage shellcode has been written there, so the first-stage shellcode allocates memory for a thread handle and uses *PsCreateSystemThread* function specifying the address of the area in which the second-stage shellcode is located as *StartRoutine* parameter.

The only other thing the first-stage shellcode must implement is the synchronization pattern to handle concurrent calls: it must ensure that its "main" is executed one and only one time.

To do that, it implements a fast function to "try to acquire" a spinlock, using *xchg* instruction in inline assembly [31]. “Try to acquire” means that the function does not wait on the spinlock: it returns True if it managed to enter the critical section, otherwise it returns False. When it returns True, then the first-stage shellcode’s main must be executed, otherwise it immediately returns.

But where is the spinlock located?

Usually, when multiple threads need synchronization, the thread which starts them allocates for them the necessary synchronization objects: in fact, if each thread allocated objects on its own, they would be different objects. To solve this problem, since there are not pre-allocated synchronization objects, it is necessary to use an allocation primitive which is able to return pointers to the same memory area. So, physical addresses must be specified as input parameters of this allocation primitive. In Windows the function satisfying these requirements is *MmMapIoSpace*.

Which physical address to use?

A good candidate is physical address 0x0, because it is used for early boot operations and left "uninitialized" after that; in fact, the platform firmware usually allows operating systems to use the first KB of physical memory as "physical memory pool" available to the OS for internal operations [32].

So, the solution is to map the spinlock to physical address 0x0, and the xchg operation is

done by using a "cookie" (0xDEAD) that is unlikely to be the value at physical address 0x0 when the first-stage shellcode is called for the first time. It is used a cookie because the value is uninitialized, so the code for synchronization must not check that the value exchanged was equal to a certain value, it must check that the value was different from 0xDEAD: if it was different, then the code is allowed to enter the critical section.

The first test of the injection chain was done by using the sequence "*int3; ret*" as second-stage shellcode, and the chain worked (the breakpoint was hit).

Since the second-stage shellcode is started with *PsCreateSystemThread*, it must terminate with *PsTerminateSystemThread:* this is an "integration" step between the first-stage shellcode and the second-stage shellcode.

Like suggested in "Finding an injection point" section, this approach does not have a 100% probability of success. Factors of unreliability and the related empirical study to estimate the success rate are available the Experiments chapter.

2.3.5 Automation

This section will introduce the "prescriptive" architecture of *Laccolith*, which guided further developments of the framework and was necessary to implement the automation of the injection chain, along with the C2 server and the portability of shellcodes implemented using run-time patching of their bytes.

It will then show two sequence diagrams, one for the first-stage shellcode and the other one for the second-stage shellcode, to better explain how they work.

The following diagram is the "prescriptive" architecture of Laccolith, hand-written on a blank paper.

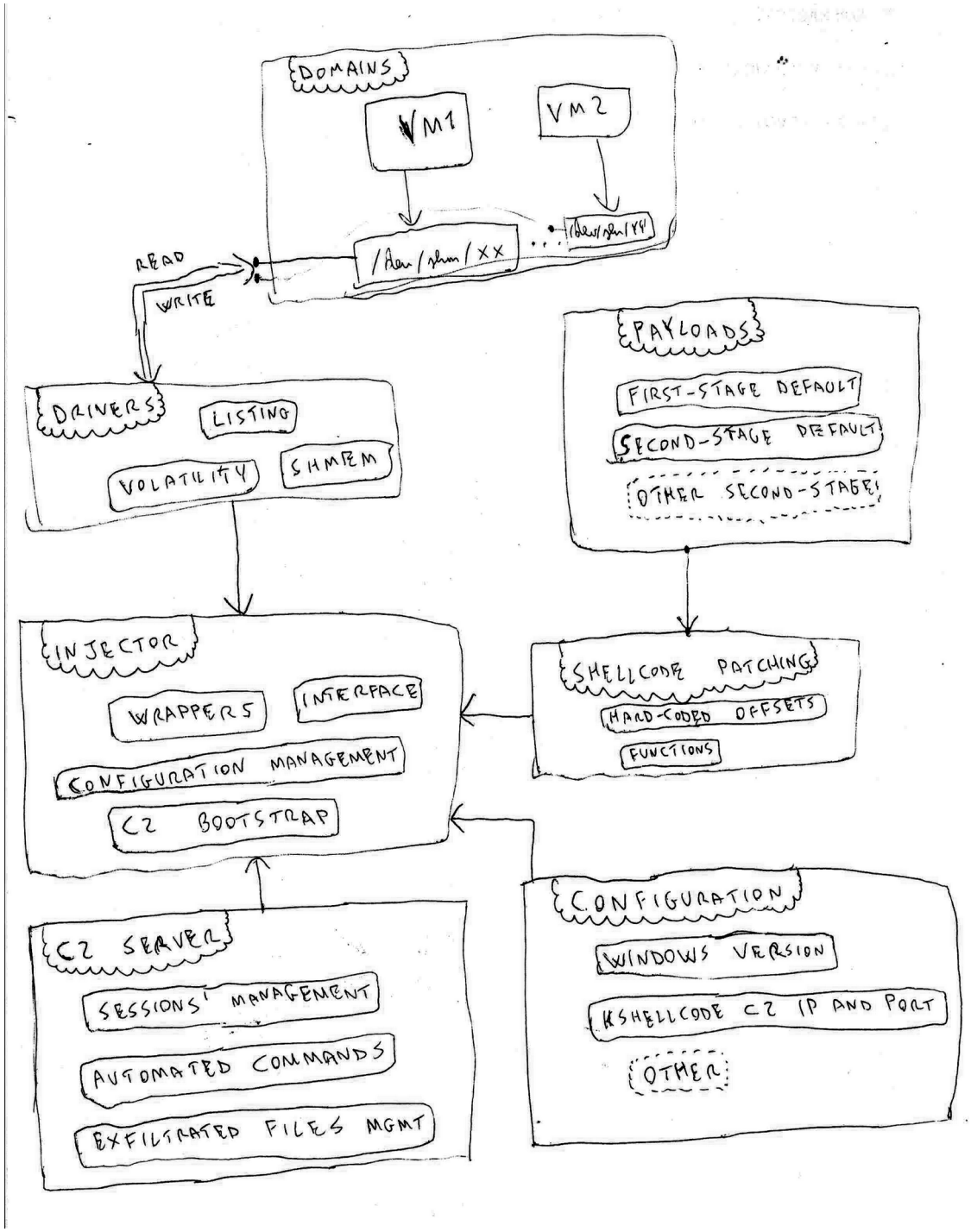

Figure 7: Laccolith prescriptive architecture

It has all the components previously expressed as requirements.

It is a mix of a package diagram and of a deployment diagram, because it highlights both how the package Drivers (for VMs) is going to have access to VM's RAM, both a flexible structure of packages, where there is a top-module which, when started, bootstraps the C2 server and becomes ready on one or more interfaces to serve requests. Its commands use the other available packages, implementing the requirements.

The portability is in two different packages:

- The drivers have to implement portability to be able to perform the injection against different Windows versions.
- The shellcode patching package has to implement portability to patch some version-specific code, for example in the first-stage shellcode the hard-coded offsets to ntoskrnl base address of WDK functions.

The full injection chain can be seen with this sequence diagram, which summarises the operations performed by the first-stage shellcode.

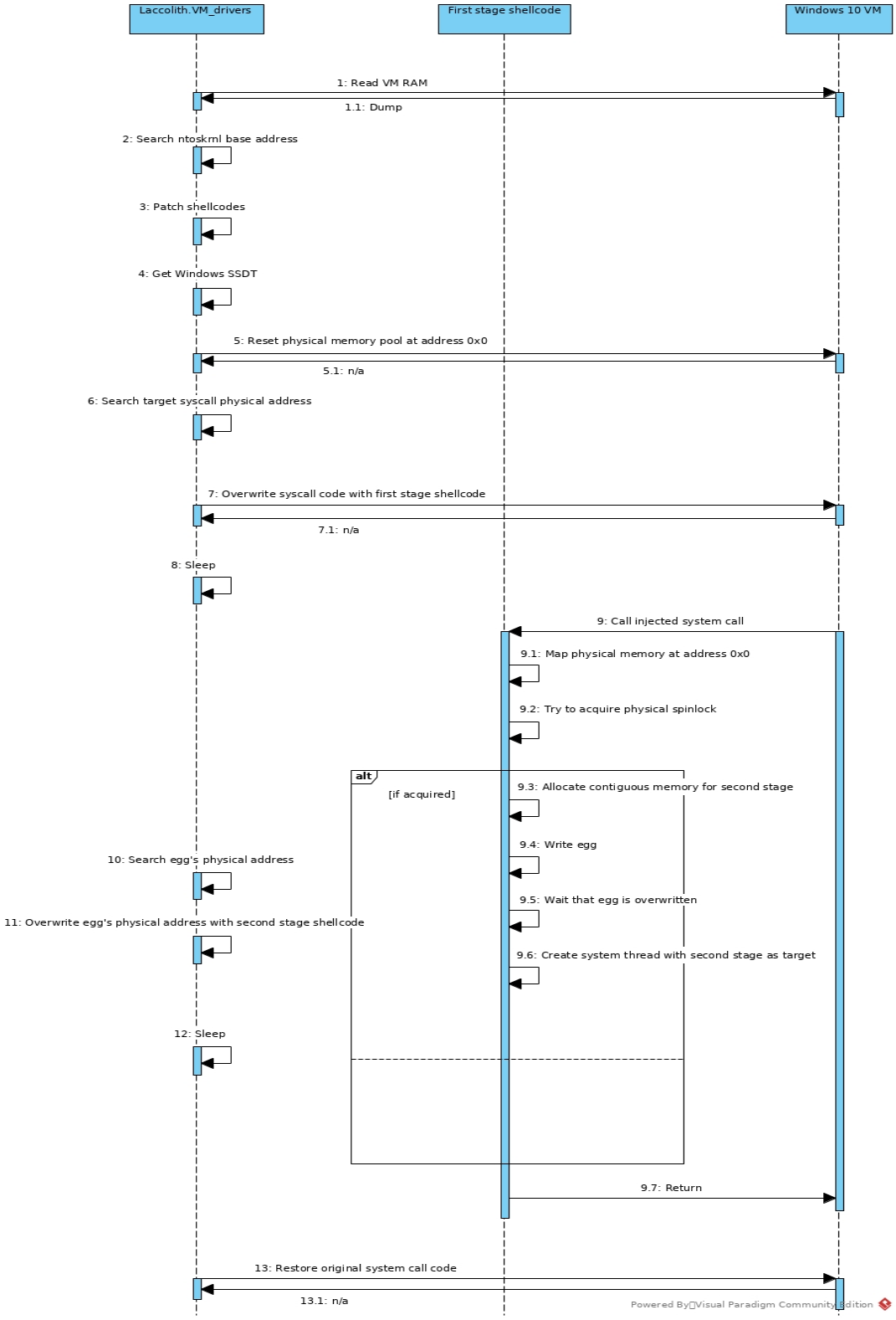

Figure 8: Injection chain sequence diagram

Note that there are two sleeps; it is a bad synchronization pattern, but it is hard to implement an exact synchronization in this context. Anyway, the first sleep is called “first stage injection latency” and the second is called “second stage injection latency”. The amount of sleep can be configured for both.

It is also possible to choose if the original system call code must be restored or not; by default, it is restored.

It’s also useful to show another diagram to better describe the injection chain. The following diagram will put emphasis on memory address spaces involved during the injection chain.

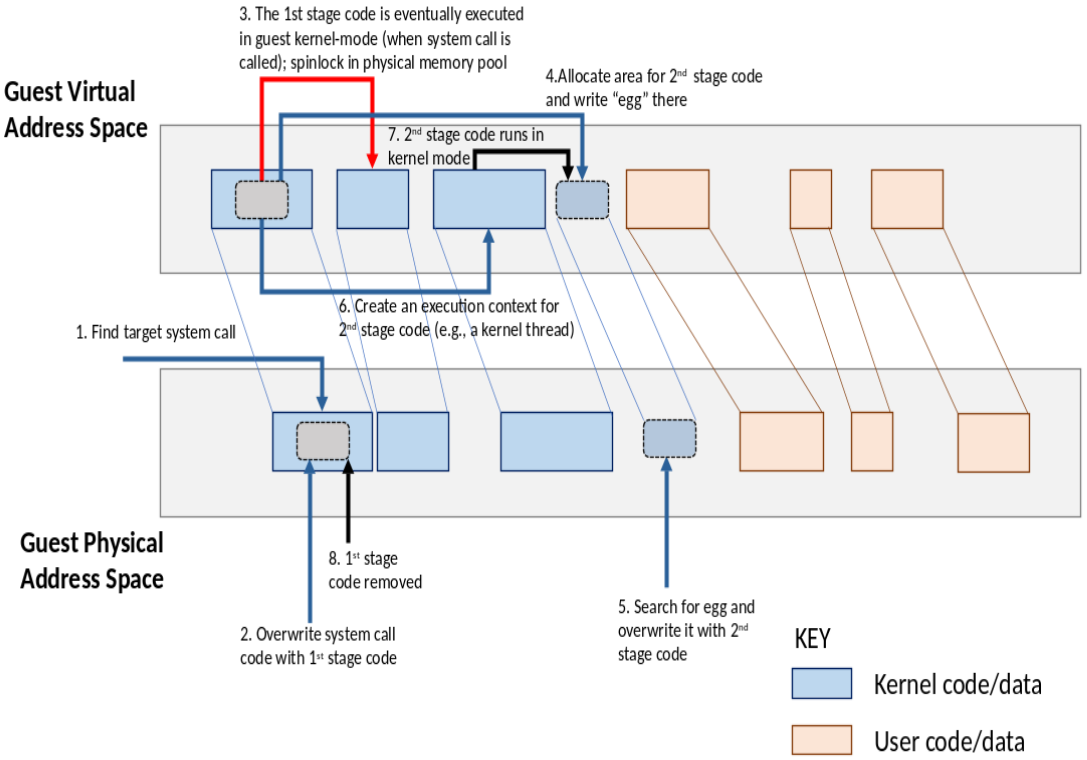

Figure 9: Injection chain with emphasis on memory layout

On the other hand, the following diagram summarises the generic sequence followed by the second-stage shellcode connecting to the C2 server, with an user, connected to the system, which wants to issue commands to a bot.

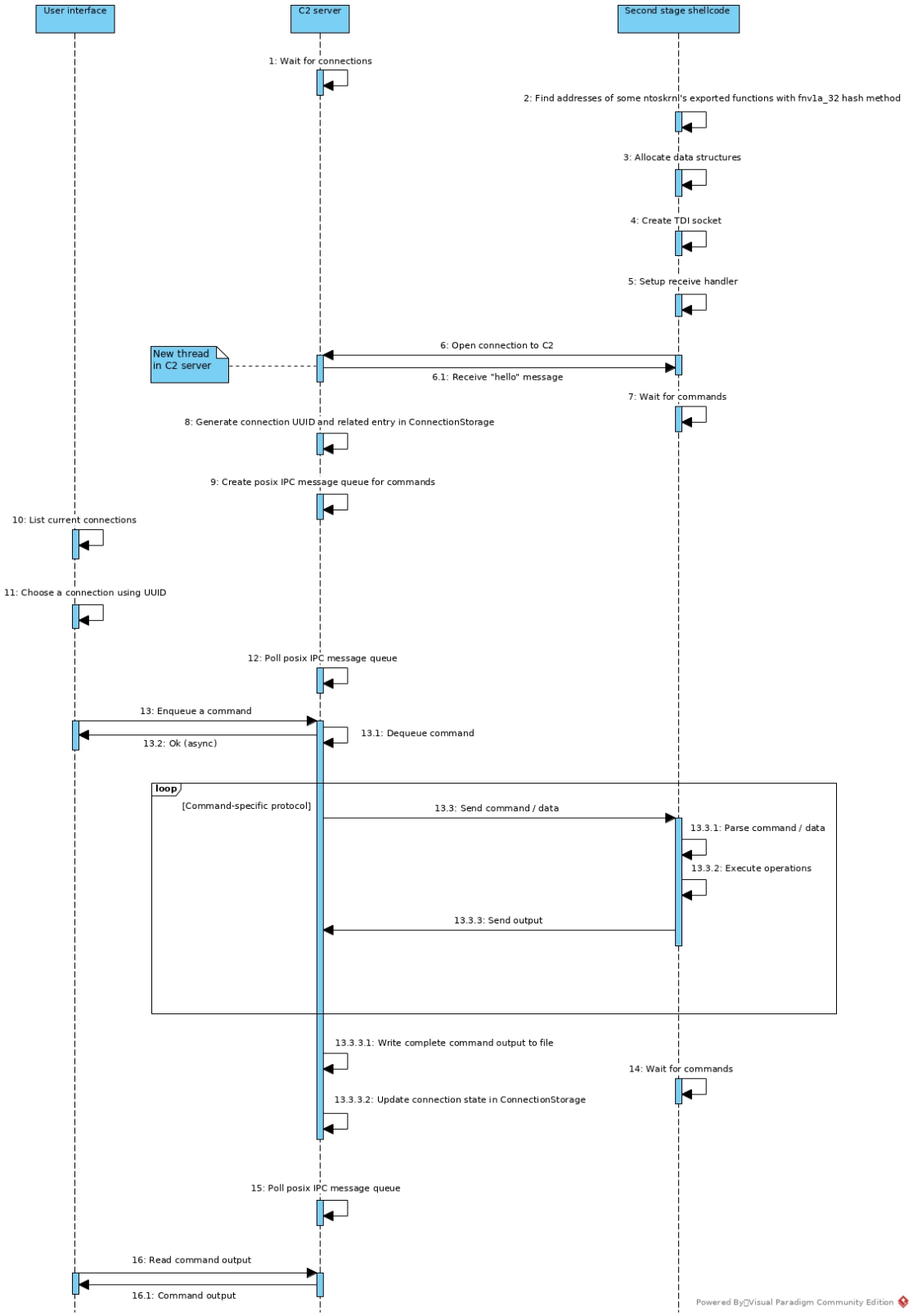

Figure 10: Second-stage shellcode sequence diagram

2.4 Implementation

This section will provide more details about the implementation of the framework.

Its main language is Python, it calls external tools like Volatility and the shmem2 shared object (using Python's CDLL library), uses the compiled payloads from Rust as artifacts and gets access to virtual domains using the previously mentioned */dev/shm* approach. It provides a web interface and a text user interface (TUI) over network; it also provides the interface of the C2 server, to which agents connect, of course.

The TUI is multi-process, so each connection to it has its own copy of variables; this is useful to customize some parameters of the injection in an instance-wise taste.

A design choice is about how the C2 server makes possible to send commands to different bots at the same time. Let's consider the case of a simple *netcat* waiting for the connection of the agent. In this case, the user interfacing with the "C2 server" would be able to interact with at most one agent at a time, because the server is in "interactive mode": standard streams are duplicated to the socket's file descriptor [33].

To handle more agents at the same time, each one of them must be handled by a separate thread; the thread must "poll" for commands and handle the connection with the agent, by forwarding the commands and by receiving and logging the output.

So, in general, in presence of multiple connected agents, the communication from the interface to a generic agent is asynchronous.

How commands flow from the interface to the connection-handling threads?

In general, the data structure to use is a queue. It could have been a simple multithreaded queue for each connection, built-in with Python, but it would not have handled the case of the multi-process TUI. Therefore, the design choice was to use Posix IPC queues [34], since they are an efficient mean of exchanging messages among processes. In addition to that, it is easy to get a handle to the right queue, by using named queues.

In fact, when the C2 server receives a connection, it automatically assigns an UUID to that connection, and updates a persistent object called *ConnectionStorage.* This object can be read using the TUI. If the user connected to the TUI wants to send a command to a connected agent, it must specify the connection UUID and the command itself: the UUID is used to get the handle of the named queue on which the connection-handling thread waits for commands, and the command is enqueued. This behaviour is also highlighted in the sequence diagram which describes the generic sequence of the second-stage shellcode, the very last of section 2.3.5.

Other low-level design choices are easier, so in the next sections the framework will be described on an architectural level rather than on a design level, with the exception of the automation of commands (separated concept from the automation of the injection). The maximum flexibility for the automation of commands is, of course, scripting with the interface, but it would not be user-friendly, and would not have a Caldera-like taste.

Therefore, it was implemented something similar to the “autonomous red-team engagements” of Caldera. In fact, since the target is also to implement some adversary profiles and test the detectability of the agent against some AV/EDR solutions there was also the implementation of Actions, Operations, Parsers and Facts, like shown in the "descriptive" package diagram (2.4.1).

An Action is similar to a Caldera "Ability", whereas an Operation is the equivalent of a

Caldera "Adversary profile" (refer to the terminology [35]).

A Parser is associated to an Action, takes as input the commands' raw outputs and produces Facts out of them, which are high-level information available to the user interface and required as input for some other Actions.

The lifecycle of an Action can be seen in the diagram at 2.4.4, which summarizes the flow of a generic action, that is customized by specifying a list of commands and by specifying a “concrete” Parser.

2.4.1 Packages

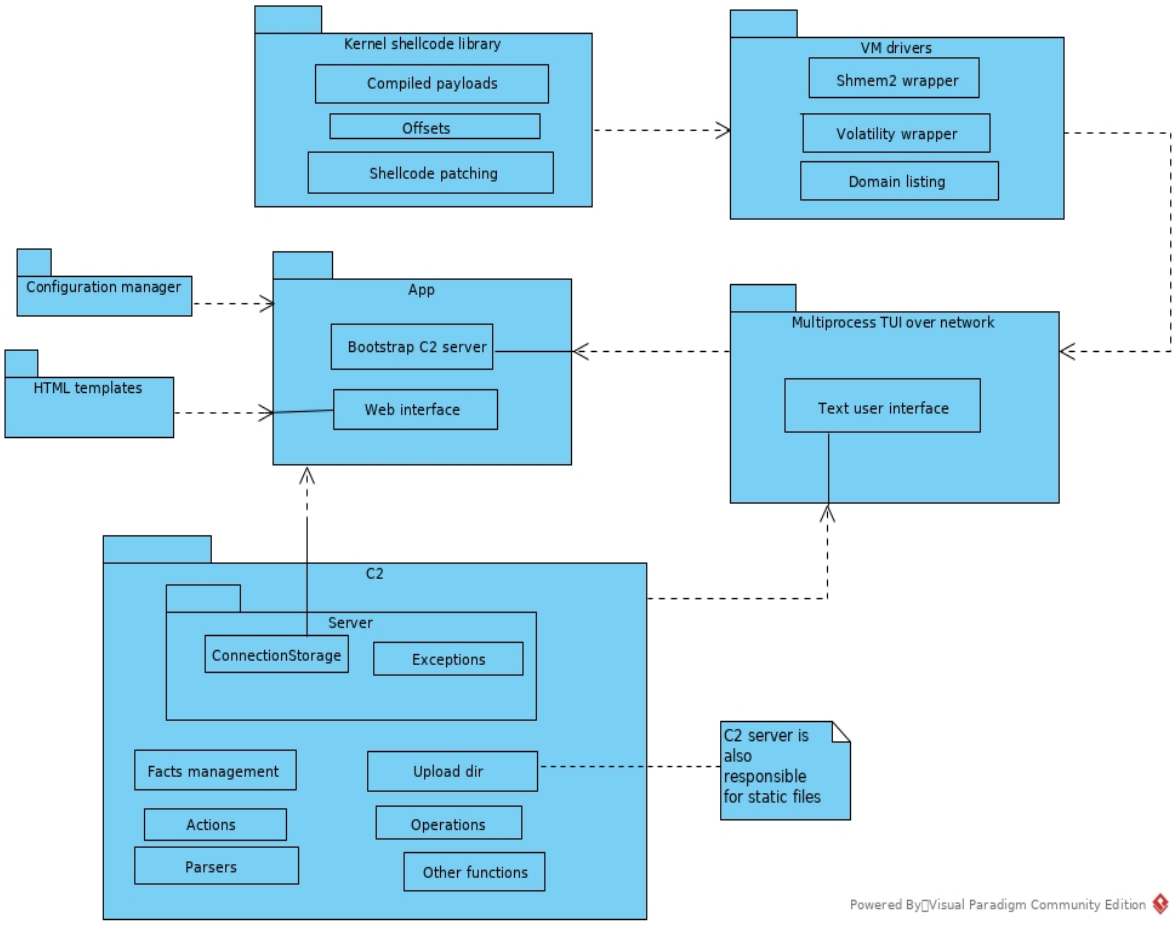

Figure 11: Laccolith descriptive architecture

There is not much left to say about packages.

The *ConnectionStorage* object provides wrappers methods and exposes data as a dictionary, so the internal implementation can be replaced with any DBMS technology. At the time of writing, it was not used a real DBMS; it was handled with a JSON file protected by a named mutex. For an easy mapping of the object (which is a JSON-serializable dictionary), it is suggested to modify this by using MongoDB as backend [36].

Like previously said (in section 2.3.5), the portability of the whole injection chain is implemented in the “Shellcode patching” module and in the “VM drivers” package.

The C2 server is also responsible for static files because it makes connection files available for download on the web interface, so the static files necessary for the web interface are placed in a directory sub-tree in which there are both the static files for the web interface and the connection files; in addition, there is the folder of uploaded files.

2.4.2 Deployment

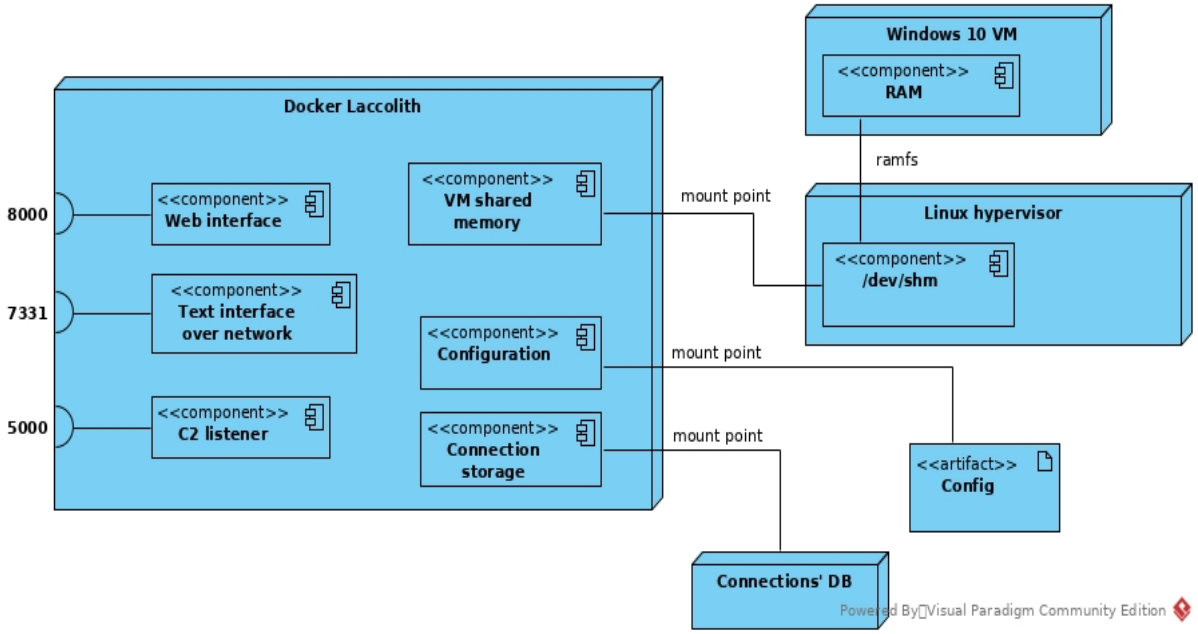

Figure 12: Laccolith deployment diagram

The framework is deployed as a Docker container. In this way, it is easy to build and deploy the container itself, it just needs the proper integration with the environment on which it is deployed, in particular for the *mount points*.

There are three mount points:

- Data for ConnectionStorage, which must be persistent and must not be erased if the container image is re-built.
- The directory in which there is the configuration file: even if some configuration parameters can be changed using the interface, others cannot be, and changing them must not require re-building the container image.
- The shared memory file system, which is */dev/shm*, which is the (crucial) way for accessing virtual domains' RAM; note that this assumes having Linux as hypervisor.

In addition, like already said, there are three interfaces:

- a web interface, listening on port 8000;
- a text user interface over network, listening on port 7331, this is the richer one and can be used to control the agents;
- the C2 port on which agents must connect, listening on port 5000.

Both web and TUI interfaces are user-friendly and, in particular, the TUI interface gives much information about each command's behaviour.

The functions implemented in the interfaces are summarized in the diagram in the next section.

2.4.3 Use cases covered by the interfaces

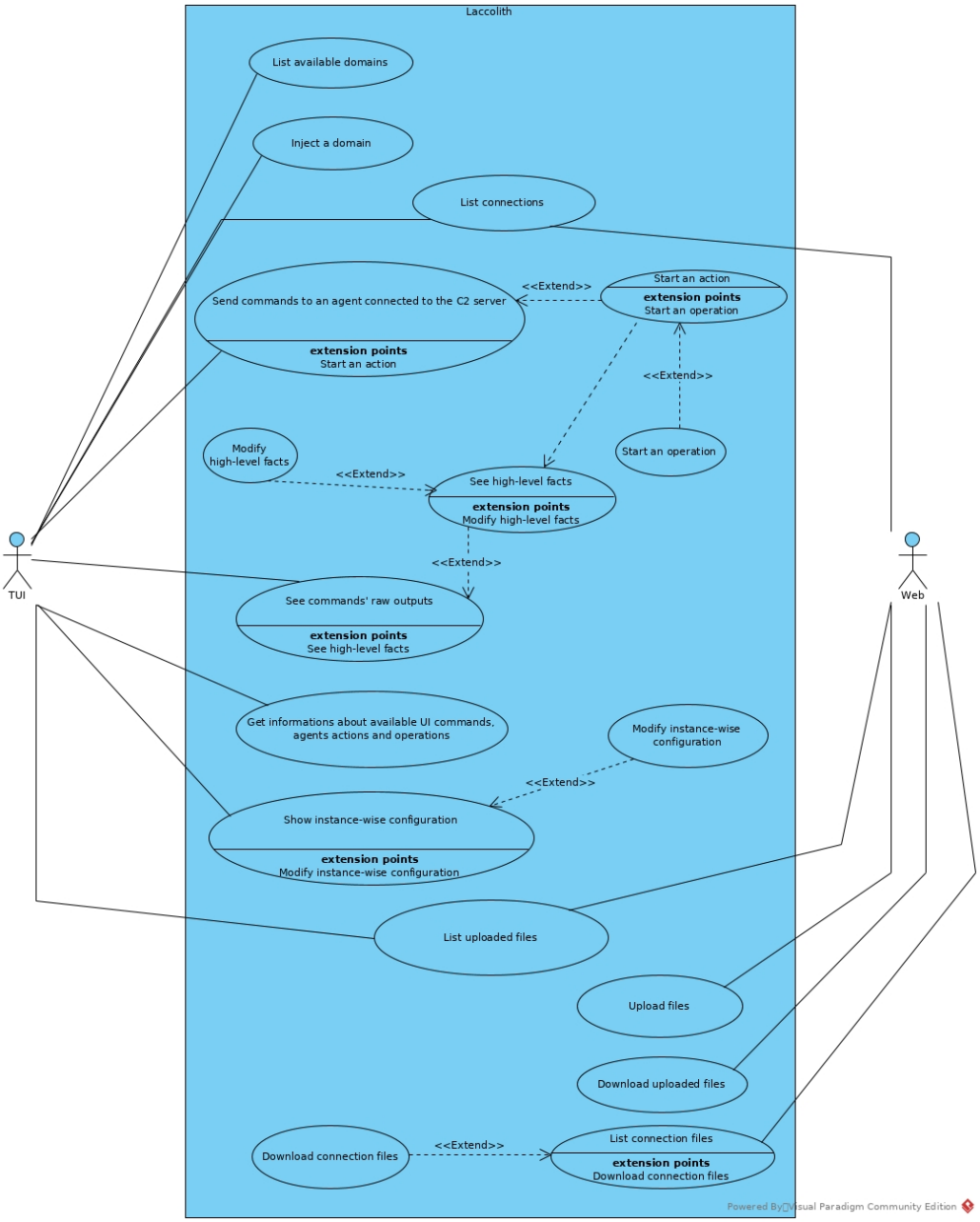

Figure 13: Use cases implemented in the interfaces

It may seem complex, but it covers all the previously mentioned requirements and should be easier to understand after having read the exposed architectural and design choices.

2.4.4 Action lifecycle

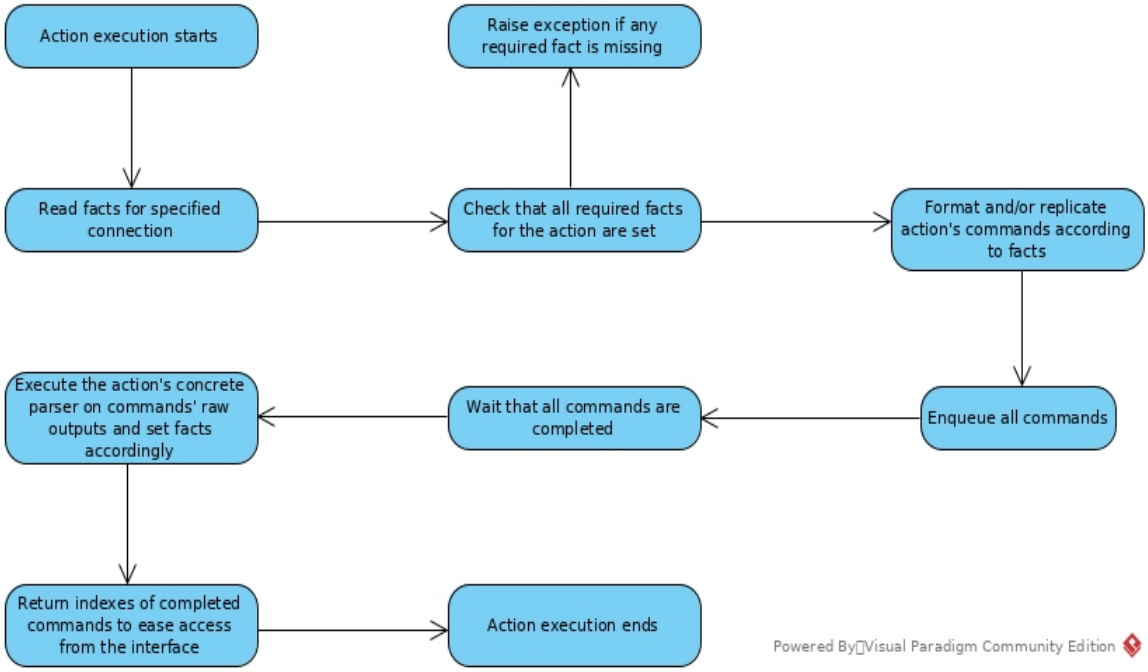

Figure 14: Action lifecycle

This diagram should be self-explanatory after having understood the terminology, introduced at last of 2.4 section.

2.5 Versions used in development & tests

Since, like any research project, this work aims to be repeatable, it is useful to document the versions used in development & tests; what is not documented here is otherwise included in build files (*Dockerfile*, *docker-compose.yml*, *requirements.txt* and so on [37]).

Table 1: Versions used in development & tests

Tool / system	**Version and additional notes**
Windows VM	10 Home, build 19044, KVA Shadowing enabled, kernel debug with KDNET, libvirt XML modified to have file-backed RAM on a link to /dev/shm (file-mapped on RAM) and to have e1000 NIC (for compatibility with KDNET)
Rust	rustc 1.63.0-nightly (12cd71f4d 2022-06-01)
Cargo	cargo 1.63.0-nightly (38472bc19 2022-05-31)
qemu-system-x86_64	qemu-5.2.0-9.fc34
Libvirt, virsh & libvirtd	7.0.0
Linux host	5.12.14-200.fc33.x86_64

3 Experiments

This section will provide results for the following experiments:

- Detectability of popular tools, taking Caldera's HTTP agent (Sandcat) as sample and also trying to integrate it with a Windows anti-detection tool called Inceptor [38].
- Detectability of the proposed approach, implementing some adversary profiles and considering the detectability from the very first steps of the injection to the end of the adversary profile.
- Reliability of the injection method, since in section 2.3.2 it was argued that the memory corruption necessary for the injection could make the system crash.

The experiments took place in a fresh Windows 10 machine, different from the one used in development & tests, but with the same build number.

The detectability will be documented by reporting the *adversary profile execution progress*, which represents the progress of the execution for each adversary profile, in terms of executed abilities / high-level actions with respect to the total number of abilities / actions of the profile itself. This metric can be represented as a fraction:

$$N_{EA}/N_{PA} \quad (1)$$

Where N_{EA} is the number of executed actions of an adversary profile and N_{PA} is the number of "profile actions", or rather the total number of actions of that adversary profile.

The "atomic" high-level actions of which an adversary profile is composed are called "abilities" for the Caldera agent and "actions" for the Laccolith agent.

Experiments were conducted in presence of different AV/EDR solutions, described in the next section.

3.1 Chosen AV/EDR solutions

Windows Defender has always been active (unless otherwise specified), since it is active by default in Windows and AV/EDR solutions are designed to be compatible with it. Other solutions deployed were the free versions of Avast, AVG, Kaspersky (Security Cloud), Avira.

They were not deployed all together, because it is not common to have many AVs active together and they "stamp each other's toes": in fact, they detect "incompatible software" and they run in "passive mode" if the incompatible software is not removed.

So, to get the best from each AV, they were deployed one at a time.

3.2 Detectability of popular tools

The detectability of the Caldera agent was tested considering many different adversary profiles, since they are available.

The profiles are executed in “atomic order”, that is each ability is executed sequentially. Therefore, if an ability is detected, the abilities that would be executed after that ability will not be executed, even if they do not have as pre-requisites the output of the stopped ability.

A thing to note is that the deployment of the Caldera agent itself is always detected by Windows Defender, so, since this defense is always active, it must be considered the pre-injection of the agent as hypothesis to avoid a null score for each adversary profile and for each AV/EDR solution.

that is why it is also considered the case of the integration of the Caldera agent with the previously mentioned anti-detection tool, Inceptor.

In addition to that, for the experiments of the adversary profiles with the Caldera agent, Windows Defender has been disabled when deploying another AV/EDR solution: since the Caldera agent is not “stealth” and Windows Defender already blocks its deployment, it was more useful to consider the execution progress of its campaigns against only one solution at a time and not against the combination of Windows Defender and another AV.

In the next two sections these situations will be explored.

3.2.1 Detectability of campaigns assuming the agent was successfully injected

The following table shows the result obtained after executing different adversary profiles using the pre-injected agent, in different AV/EDR scenarios; it is worth noting again that

in this case Windows Defender is not active in combination with other AVs.

They were selected twelve among the default Caldera's campaigns, of which the first eight are just *reconnaissance & information gathering* and the other four are *advanced*.

The first category includes profiles which perform basic operations: user identification, process enumeration, antivirus discovery, screenshot capture, and file search.

On the other hand, the advanced profiles perform more invasive operations, like process injection, lateral movement, and malicious payload execution. As a consequence, their activities will be noisy and more likely to be flagged by the antiviruses.

The activities made by each profile can be summarized as follow:

- *Discovery*: discovers host details, such as local users, user processes, admin shares, and antivirus program;
- *Hunter*: performs Discovery operations, then tries to exfiltrate files from the working directory;
- *Check*: checks details of the platform configuration, like Chrome, Go and Python installation and network interface configuration;
- *Collection*: collects information from the host, like company emails, IP addresses and files;
- *Enumerator*: enumerates different types of processes, such as WMIC, PowerShell, and SysInternals processes;
- *Nosy Neighbor*: finds preferred Wi-Fi networks and tries to disrupt the connection;
- *Signed Binary Proxy Execution*: performs signed binary proxy execution;
- *Super Spy*: monitors the active user by capturing screenshot, copying clipboard, scanning preferred Wi-Fi networks;
- *Undercover*: swaps from built-in PowerShell to PowerShell Core to stop

PowerShell processes;

- *Stowaway*: injects Sandcat into a process;
- *Worm*: runs PowerKatz to steal credentials and then moves laterally in any possible way;
- *You Shall (Not) Bypass*: bypasses User Account Control (UAC).

In addition to that, to make a "stricter" comparison with the proposed approach, it was estimated the execution progress of a "Ransomware" profile. This profile was designed ad-hoc, looking at how other profiles are implemented.

In fact, its first 4 abilities re-use abilities of other profiles; the full list is:

- Find files;
- Stage sensitive files;
- Compress staged directory;
- Exfil staged directory;
- Encrypt sensitive files.

For the last ability, it uses a powershell script [56] to encrypt the previously found sensitive files, so it silently skips the files it does not have the permission to encrypt.

Using a powershell script with standard APIs for encryption makes the process more "legitimate" and, therefore, more difficult to detect; on the other hand, it also makes the encryption process weaker, because powershell may log commands and, with them, information that could be used to recover the key. In this way, it is expected to be as detectable as the other *reconnaissance & information gathering* adversary profiles.
The intelligence summary of this profile:

- *Ransomware*: find sensitive files in users' folders, exfiltrate them and encrypt them, skipping the files for which permission is denied.

Results are reported in the following table.

Table 2: Adversary profiles execution progress for Caldera

Profile	**Windows Defender**	**Avast**	**AVG**	**Kaspersky**	**Avira**
Discovery	9 / 9	9 / 9	9 / 9	9 / 9	9 / 9
Hunter	14 / 14	14 / 14	14 / 14	14 / 14	14 / 14
Check	6 / 6	6 / 6	6 / 6	6 / 6	6 / 6
Collection	2 / 2	2 / 2	2 / 2	2 / 2	2 / 2
Enumerator	5 / 5	5 / 5	5 / 5	5 / 5	5 / 5
Nosy Neighbor	7 / 7	7 / 7	7 / 7	7 / 7	7 / 7
Signed Binary Proxy Execution	3 / 3	3 / 3	3 / 3	3 / 3	3 / 3
Super Spy	11 / 11	11 / 11	11 / 11	11 / 11	11 / 11
Undercover	1 / 2	1 / 2	1 / 2	1 / 2	2 / 2
Stowaway	1 / 2	1 / 2	1 / 2	1 / 2	2 / 2
Worm	1 / 9	1 / 9	1 / 9	1 / 9	9 / 9
You Shall (Not) Bypass	2 / 4	2 / 4	2 / 4	1 / 4	1 / 4
Ransomware	5 / 5	5 / 5	5 / 5	5 / 5	5 / 5

It can be seen that most profiles are executed without problems, but they do not do harmful actions. In fact, it is useful to inspect which are the detected abilities for each profile.

Table 3: Detected abilities for Caldera's profiles

Profile	**Detected Ability**
Undercover	Install PowerShell Core 6
Stowaway	Inject Sandcat into Process
Worm	Run PowerKatz
You Shall (Not) Bypass	Wow64log DLL Hijack
You Shall (Not) Bypass	Bypass UAC Medium

Caldera's agent, in the pre-injection situation, is able to execute most Discovery / Collection / Exfiltration techniques, but it is easily blocked when it tries to do Credential Access / Privilege Escalation.

3.2.2 Trying to integrate the agent with anti-detection solutions

According to its documentation, Inceptor is a template-driven AV/EDR evasion framework.

A template is a generic, customizable "loader" which has placeholders for evasion techniques and for the actual payloads. There are many templates for three different types of payloads: .NET, powershell and native.

There is also the possibility of chaining encoding techniques (such as *Shikata-Ga-Nai* and others) to overcome static analysis.

Other "source code writing" techniques are pluggable to try to evade AMSI's dynamic analysis [39].

Anyway, AMSI also analyse in-memory artifacts, for example text areas the code is going to jump to, so there is also the implementation of some AMSI bypass techniques, if applicable [40].

Experiments done here tried to inject a Caldera agent into the Windows 10 VM, with only Windows Defender active, trying many different combinations of the Inceptor's techniques, after having extracted a "raw payload" to inject.

Since all combinations except one were always blocked either with just static analysis or after dynamic analysis, it was not considered useful to report here all failed tries; it is reported only the "almost successful combination".

This combination used: a native (binary) template, an encoding chain composed of Shikata-Ga-Nai and xor encoding, 120 seconds of execution delay, unhook technique for EDR bypass, and the resulting binary was signed by Microsoft using CarbonCopy [41].

The research showed that binary templates used to perform better; the encoding chain is useful to bypass static analysis, whereas the execution delay and unhook technique heuristically helps to bypass some dynamic analysis. But the crucial addition was the binary signature. In fact, like highlighted in the previously referenced Inceptor's presentation, if the binary has a valid signature, then the amount of dynamic analysis performed is lower.

With this combination, approximately 1 time on 4 the agent is successfully injected, which is a 25% probability (with some margin of error). But it was not done a complete empirical study on that, and the injection is considered only "almost successful", because, even when not detected by Windows Defender, there is still the UAC (Microsoft SmartScreen).

In fact, the binary is executed only after UAC approval, and after UAC approval it has still an approximate probability of being executed as low as 25%. The UAC approval mitigates

the severity of this injection technique, because it either requires user interaction or a previous access which implemented UAC bypass and execution techniques.

3.3 Detectability of the proposed approach

After having showed that popular tools are easily detected when trying to do privileged actions and are hard to inject, there come the experiments with the novel approach.

3.3.1 Implemented profiles

There were implemented three adversary profiles, called *Thief*, *Op-2* and *Ransomware.*

Since many actions performed by Caldera's adversary profiles required a download-and-execute chain, it was not easy to do a 1:1 mapping to these adversary profiles. Instead, implemented profiles are similar to some Caldera profiles in terms of high-level requirements, and their actions reference techniques used by real world APTs (*threat-informed adversary emulation*).

Furthermore, the experiments done with these three profiles cover all the tactics covered by the experiments done with Caldera's profiles, expect Privilege Escalation, because the privilege is already high (but the the technique used for Persistence is for high-privileged actions in user-space). The *ransomware* profile also includes Impact, that is not even covered by Caldera's default profiles; in fact, this profile is a "provocative" profile to argue that complex operations can be implemented with the proposed approach.

More details about the profiles are available in the following table and in section 3.2.3, which contains a more low-level description of the profiles.

Table 4: Adversary profiles for Laccolith

Profile	**Description**	**Tactics**	**Commands used**	**High-level actions**	**Referenced APTs (examples)**
Thief	Exfiltrate files from local user desktop	Discovery, Collection, Exfiltration	Directory listing (one time to find local users and another time to list desktop files), Read file	(1) Find local users, (2) List user desktop, (3) Exfiltrate a list of staged files	APT1, OilRig, APT3
Op-2	Upload a powershell script in a system folder and install a scheduled task that executes that script at boot, get system version and dump memory of LSASS	Persistence, Credential access	Write file, Write to registry (many times), Version, Dump process memory	(1) Write file on remote file system, (2) Install a scheduled task on the remote Windows target, (3) Get system version, (4) Dump lsass credentials	Remsec (Strider), Ke3chang

	process				
Ransomware	Discover and exfiltrates sensitive files, encrypt them and leave a message	Discovery, Collection, Exfiltration, Impact	Directory listing (multiple times), Read file (many times), Write file (many times plus one)	(1) Find local users, (2) Find sensitive files, (3) Exfiltrate a list of staged files and encrypt them, (4) Encrypt remote files, (5) Write ransom message	APT3, Bad Rabbit (multiple APTs)

3.3.2 Adversary profiles execution progress

The following table shows the adversary profile execution progress for the three implemented profiles, for all the chosen AV/EDR situations. it is worth noting that in this case, as opposite to the case of Caldera, Windows Defender is always active.

In addition to that, this table also takes into account the injection phase, not just the progress of adversary profiles.

Table 5: Adversary profiles execution progress for Laccolith

Profile	**Windows Defender**	**Avast**	**AVG**	**Kaspersky**	**Avira**
Thief	3 / 3	3 / 3	3 / 3	0 / 3	3 / 3
Op-2	4 / 4	4 / 4	4 / 4	0 / 4	4 / 4
Ransomware	5 / 5	5 / 5	5 / 5	0 / 5	5 / 5

About Kaspersky, it does not detect the attack; it uses the system in a more aggressive way, somehow lowering the success rate of the injection. In fact, it makes the system reboots and asks to send a crash dump to Kaspersky servers, but it does not show any log of the attack; also because the connection rarely reaches the C2 server, so the actual Operation never starts.

Therefore, in any case the injection and the execution of the Actions is fully undetected, just in the case of Kaspersky the injection fails a little time after the start of the second stage shellcode.

Another thing to mention is that, after the dump of the *LSASS* process, the detach from the process makes the system crash, probably it is an invalid detach.

The following picture recalls the architecture of the experiment, putting emphasis on the presence of the *LSASS* process.

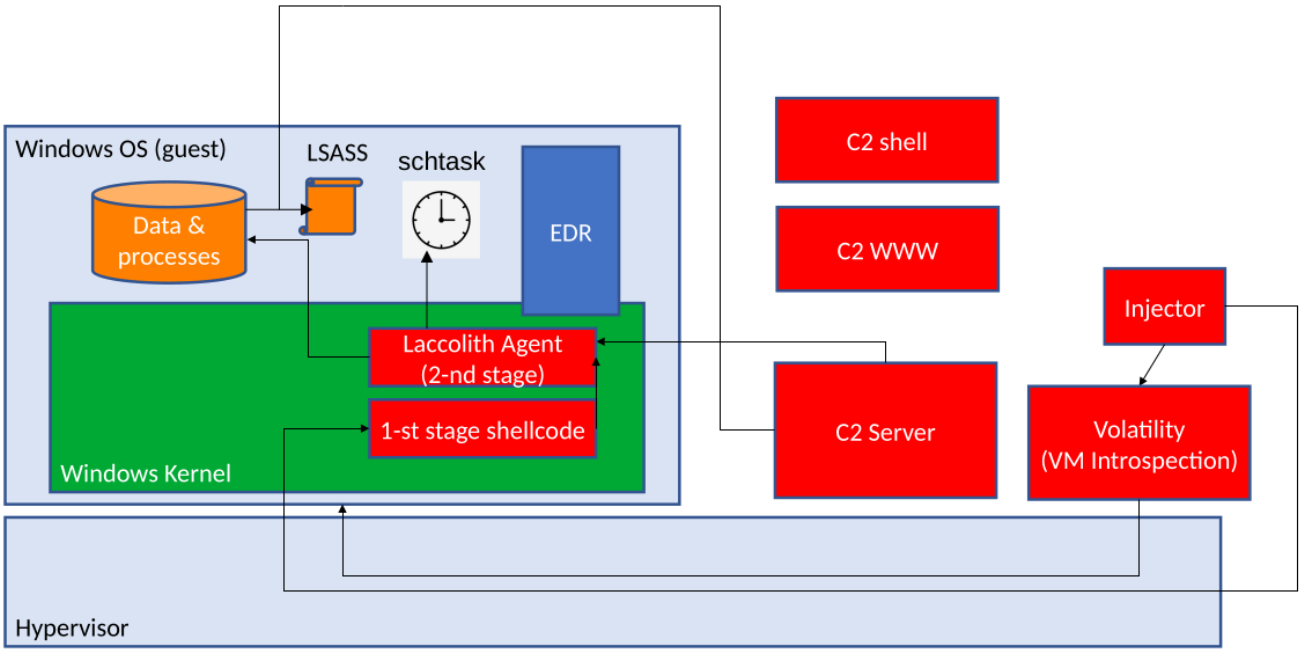

Figure 15: Experiment architecture with injection from hypervisor, emphasis on Op-2

3.3.3 Commands' coverage

This section provides details about which second-stage shellcode's commands are involved in each profile, to provide a "test coverage" of the second-stage shellcode with respect to implemented adversary profiles.

it is worth noting that the "*usermode*" command is still counted in the entire list of commands even if it was left as stub.

Table 6: Test coverage of second-stage commands with adversary profiles

Profile or aggregation	**Commands coverage**	**Commands**
Thief	2 / 11 (18.2 %)	dir, read
Op-2	4 / 11 (36.3 %)	write, setkey, version, dump
Ransomware	3 / 11 (27.3 %)	dir, read, write
Overall profiles	6 / 11 (54.5 %)	dir, read, write, setkey, version, dump
All commands	11 / 11 (100 %)	echo, close, version, dump, pslist, dir, write, read, queryvalkey, setkey, usermode

Of course, commands not included in this coverage metrics were still tested, but they are used in high-level actions, not in fully-fledged operations.

3.4 Reliability of the injection method

3.4.1 What can go wrong

Let’s recall concepts from sections 2.3.2, about possible failures of the injection method.

The injection relies on overwriting the **code** of a system call, or of a function called by a system call. It is done in such a way that the first-stage shellcode is executed only one time, and the other calls just return, by using a "physical mutex". After the loading of the second-stage shellcode, the original code is restored.

It can go wrong in two ways.

3.4.1.1 Way 1

The code is restored concurrently with a thread that is executing that function, which could encounter invalid opcodes due to unalignment, or code that is valid but that gives errors because of invalid values in registers.

Result: bug check in kernel.

3.4.1.2 Way 2

A critical user-space process (such as a SYSTEM svchost.exe) crashes because of the faulty system call which does not exhibit the expected behaviour.

Result: the system reboots.

With experiments, it was found that the probability of these events is higher in the first minutes after the boot; afterwards, the injection becomes more reliable.

Therefore, the success probability of the injection method with respect to elapsed time after boot could be modelled as an exponential distribution.

It would require many experiments to verify it, so the next section will report results of an empirical study of success probability of the injection only considering injections performed “a certain time after boot”, better described in next section.

3.4.2 Empirical study

To study the success probability of the injection method, there are different approaches.

The most precise one would be to leverage the knowledge about the possible failures and obtain the probability of success as the opposite of the probability of failure.

The probability of failure is the sum of the probability of “way 1” and the probability of “way 2”, described in the previous section, if they are independent failures. An example of starting point to measure these probabilities is to measure the ratio at which the injected system call is called, or the fraction of time that the system spends executing that system call. it is not easy to measure the exact probability of failure from there, and a lower fraction of time does not automatically mean a lower probability of failure, because it could be a low fraction of time of critical processes as opposed to a higher fraction of time of “common” processes, which do not make the system reboot if they crash.

Therefore, it was preferred to do a black-box empirical study, just by performing the injection multiple times and counting how many times it was successful.

There were conducted 20 experiments for each AV/EDR situation, re-booting the VM each time to have independent samples. The previously mentioned “certain time after boot” was chosen to be one minute after that the login prompt appears. For each injection, to verify that it was actually successful, it was sent an “*echo*” command and a “*close*”, to verify that the agent is able to execute commands and to terminate gracefully. The parameters “first-stage injection latency” and “second-stage injection latency”, described

in section 2.3.5, were set to 2 seconds each. If the connection is received, but it is not possible to perform the echo and/or to close the connection gracefully, then the injection is considered failed. On the other hand, if there is some minor error in the GUI, which does not lead to system reboot / agent crash, like “unknown exception in explorer.exe”, the injection is considered successful.

The following table summarizes the experimental results.

Table 7: Empirical reliability of the injection technique

Windows Defender	Avast	AVG	Kaspersky	Avira	Overall without Kaspersky (percentage)
15 / 20	16 / 20	17 / 20	0 / 20	16 / 20	64 / 80 (80%)

Note that for Kaspersky the probability of success is 0%, or otherwise very low; it is hard to explain why, perhaps the best way to go is to improve the injection method, for example mitigating “way 2” of failure and seeing if it still fails.

The other situations exhibit a statistically similar behaviour, even more similar by recalling that Windows Defender is active in any situation. So, to have a higher number of repetitions they can be considered belonging to the same “class”: in this way, it is possible to sum the repetitions of the different situations and extract from them an “overall probability”, after removing all “points” from Kaspersky’s situation, which are *outliers*. The expected value for the overall probability is 80%, then.

it is possible to approximate the margin of error as $\frac{1}{\sqrt{N}}$ (2), where N is the number of repetitions [42].

By doing so, the probability of success of the injection method ranges from 68.8% to 91.2%, that is a broad range but centred around 80% which is high enough to say that this approach is much more reliable than “Caldera+Inceptor” approach, which had approximately a 25% probability of success but also required UAC bypass.

4 Conclusions

In this work it was evidenced the lack of defense evasion capabilities of one of the most popular adversary emulation tools (Caldera) in Windows 10 environment, arguing that it is not able to fully emulate the capabilities of a skilled adversary, both for the way the agent is executed and for the techniques used to perform the actions of the adversary profile.

To address this issue, a novel approach with anti-detection capabilities was proposed, starting with the hypothesis of injection from hypervisor. This hypothesis represents a real threat in cloud computing systems, since it is a hypervisor post-compromise situation [43].

The proposed approach was simulated by considering an arbitrary read/write access to a Windows 10 VM's RAM and was implemented after performing research about Windows' kernel internal data structures, solving different complex technical issues and developing an automated injection chain, an agent to be executed in Windows' kernel and a framework ("Laccolith") to put all together and to provide a front-end.

Then, experiments were conducted to show that the injected agent is able to perform high-privileged actions without being detected, as opposed to Caldera's agent; furthermore, the injection chain, although not being fully reliable, proved to be still much more reliable than the execution of the Caldera's agent even when this one is integrated with powerful anti-detection tools.

Therefore, this approach can be expected to be used for purple team engagements, in which red teams can operate easily thanks to the defense evasion capabilities, and blue teams can train harder for the same reason.

In addition, it can be used to better analyze the impact of a hypervisor's compromise.

It can also be expected that this approach will have future developments. Some possible future developments are described in the next chapter.

5 Future developments

This chapter will describe different development lines that can be taken to contribute to this research.

Section 5.1 is more generic and refers to issues that are also documented on project's GitHub repository [37]; some of these issues only require effort on Python code, others only on Rust code, and others on the whole structure of the project.

Section 5.2 is about portability of the injection chain and of the agent, it requires a proper setup but a little effort on the code (adding key-values to two dictionaries in Python).

Section 5.3 refers to the open problem of implementing the approach for other operating systems, so it may require a great research and engineering effort on all the project's structure and even on the structure of the kernel shellcode library.

Section 5.4 is about the implementation of the user mode transition in the second-stage shellcode, which is a nice-to-have requirement; it requires research, but the effort is mostly focused on Rust code related to the second-stage shellcode.

Section 5.5 proposes experiments for situations in which there is hardware-assisted protection for virtual domains that do not trust the hypervisor.

5.1 Handle existing issues

5.1.1 Increase the reliability of the injection method

In section 3.4.1, it is documented that the injection can go wrong in two ways; "way 1" is very hard to address but has a lower probability, whereas "way 2" is still quite hard and has a higher probability, as shown in the experiments with Kaspersky.

"Way 2" could be addressed by designing the first-stage shellcode such that it does not return without doing the system call's duty, but somehow "hooking" the system call with the help of the hypervisor, using a technique similar to the one used to inject the second-stage shellcode, but synchronous.

5.1.2 Refactoring the project

When developing the framework “Laccolith”, the project maintained the original structure it had during the initial research, when different “things” were being tried each one separately from the others.

The integration among the pieces of the puzzle was done using Python code; some packages were developed brand new, such as the C2 server, but other pieces of code, such as the shellcode patching part (which was originally a command-line utility) were used as they were. So, there are source code directories mixed with other kinds of directories.

This structure should be refactored to have a clearer view of the project when looking at the project’s tree.

5.1.3 Minor bug fixes

There are minor bug fixes to do.

One of them is the timeout handling in Actions and, therefore, in Operations. In fact, each Action executes sequentially its list of commands, and waits for the completion of each

command by polling the *ConnectionStorage*. In this way, if the agent crashes during a command, the interface which started the Action will hang forever and will not give information about the partial state of the Action. This bug must be fixed in such way that partial state's information is returned.

A more complex issue is the performance of the write protocol between C2 server and second-stage shellcode. In fact, writing large files using the protocol implemented in the second-stage shellcode is very slow. This is because the communication with the C2 server uses a synchronization pattern which allows the reliable execution of only one "command" at a time and needs the response to go on. Each "command", which can be either an actual command or data for the command-specific protocol, is at most 1460 bytes. Therefore, writing a large file requires to write it 1460 bytes at a time, with a complete handshake for each data unit. The handshake is completed in the "main" of the second-stage shellcode, which executes at *PASSIVE_LEVEL*, so it takes a few seconds for each "command". So, to write a file of 1 MB, it can take up to half an hour.

The write protocol, the synchronization pattern between the receive handler and the main of the second-stage shellcode, and the synchronization pattern between the second-stage shellcode and the C2 server must be reviewed to increase the performance of the "write_file" action. This will need development in both the C2 part (Python) and the second-stage shellcode part (Rust).

5.2 Implement portability for other Windows versions

The portability needs to be implemented for each Windows update, because the injection chain leverages Windows "deep internals". On a high-level perspective, the portability part is described in section 2.3.5.

This section will provide practical details on how to implement the portability.

The schema used is *"build_number.minor_update"*, like "19044.2006". Implemented versions are in two dictionaries:

- injections points: *injection_points_by_ver* in *vm_drivers.py*;
- PI addresses: *funcs_pi_addresses_by_ver* in *kernel_shellcode_library/shellcode_patching.py*.

In the first dictionary, for each version there is a list of injection points, each one with:

- *syscall_name*, like shown in Volatility's *windows.ssdt* plugin, which returns the virtual address of that system call, used to get the page offset (virtual page offset = physical page offset);
- *expected_bytes* is a list of possible byte arrays to look for, it is used to "translate" the virtual address to a physical address by performing a search on every physical memory page at the previously obtained page offset: it contains the code of the system call, and must be enough long to be unique, but enough short to have a low probability of crossing the page's boundary;
- *injection_offset* is the offset at which the injection of the first-stage shellcode is performed with respect to the target system call: for example, in the case of *NtQueryVirtualMemory* system call, the actually overwritten function is *MmQueryVirtualMemory*, at offset 64, because the system call "stub" was too short.

In the second dictionary, for each version there is a list of offsets, which are position-independent addresses of functions in *ntoskrnl* virtual address memory range (the names of the functions must match with the names specified as sub-keys of *hardcoded_function_offsets* sub-key in *kernel_shellcode_library/offsets.json*). This dictionary is used for shellcode run-time patching before injection, so there is the concept of "offset within the shellcode" and the concept of "function offset to *ntoskrnl* base address" (a bit like "return to libc" methodology).

The two dictionaries are easily filled for new versions after setting up kernel debug.

How can you find the needed information?

The first thing you have to do with the kernel debugger is reloading kernel symbols, using "*.reload*" command (in *WinDbg*). Then, for the first dictionary you make the debugger show the address of the system call, in this case with the command "*x ntoskrnl! NtQueryVirtualMemory*", then you disassemble that address and verify that the offset to *MmQueryVirtualMemory* is still 64, and at last you use "*db*" command to display bytes at the system call's virtual address and copy them to build the expected bytes; 240 bytes are a good trade-off, because they are less than 1/8 of 4 KB, so the probability of crossing a page boundary when searching for the injection point, assuming an uniform distribution of the injection point's page offset, is less than 12.5%.

For the second dictionary it is easier: you get *ntoskrnl* base address, then you use the display symbol command (like "*x ntoskrnl!MmAllocateContiguousMemory*") to obtain the virtual address of the function and get the offset by subtracting the first address to the second address.

If you understood everything, including the concept of linear region of code in section 2.3.2, you can even find your own injection points and/or add more hard-coded functions to the first-stage shellcode (but in this case you also have to work on the "offsets within the shellcode").

Another possible future development could be to obtain the required information for the two dictionaries by only using Volatility/shmem, without setting up kernel debug.

5.3 Implement payloads and injection chain for other operating systems

It was very interesting to delve into Windows internals to develop the injection chain and the kernel-level shellcodes.

At this point, it could also be interesting to do the same with other operating systems. The attack scenario resembles the scenario of a cloud environment, in which the hypervisor's exploit results in a virtual domains' compromise.

Therefore, considering the case of mobile operating systems could still be useful for adversary emulation, but may not represent the scenario of a real threat.

So, a possible family of target operating systems may be Linux distributions used in cloud [44], like Ubuntu, Debian, RHEL, Amazon Linux 2.

It may also be interesting to explore the case of MacOS, although it, like Windows, would require a significant effort of reverse engineering.

5.4 Implement ring0-to-ring3 technique in a portable and reliable way

Like previously said, having the possibility of executing code in user-mode is useful both for customizing the defense evasion capabilities and to have access to higher-level functions.

Because of processor privileges, kernel mode is also called "ring 0", whereas user-mode is also called "ring 3" [26].

The "ring0-to-ring3" technique must be:

- *reliable*: it must be deterministic, or rather the code must eventually execute and the technique itself must evade detection;
- *portable*: the resulting shellcode must easily be patched automatically before injection, and the information needed for patching must easily be obtained for different Windows version or otherwise a script to automatically obtain the required information must be implemented.

Several approaches were explored by browsing "unofficial rootkit documentations" and by brainstorming about Windows internal functions. it is useful to group them in the following categories.

5.4.1 Indirect user-mode

This is not a real "ring0-to-ring3" technique. The idea is to use existing execution techniques in user-space and trigger them by using kernel functions.

For example, a working technique is to replace a legitimate executable (or a DLL) that is executed periodically with a malicious one. But this is not reliable: the malicious binary is easily flagged as malicious by Windows Defender, in first instance because it is not signed. The reliability of this technique can be enhanced by combining it with other techniques to impair defenses, like setting keys in the system registry to allow UAC

bypass, and so on. But it would still have a low reliability because there is the open question "when the binary will be executed".

Another technique, experimented in this research, is about the scheduled tasks [45]. To register a scheduled task, there are user-space APIs, which require high privileges. But, after a scheduled task is registered, there is a related XML file in *C:\Windows\System32\Tasks* directory tree; it is only used for integrity checks. In fact, the actual task's information is in system registry [46]. Anyway, using the kernel agent to modify the registry by adding new tasks or by changing some already registered tasks seems to not have effects on the actual scheduled tasks: the newly created task can be seen using the scheduled tasks' GUI, but never gets executed. This is maybe because the GUI queries the registry, but the task scheduler does not, it needs to be notified. So, this technique can work at most for persistence, unless a raw way to notify the task scheduler is found.

Much brainstorming can be performed for the "indirect user-mode" strategy; it is hard to find a reliable technique with this strategy, but for sure it will not need any portability efforts.

5.4.2 Raw context switch

The idea is to reverse engineer the way in which Windows performs the context switch from kernel-mode to user-mode and use it in a separate thread to execute an user-mode shellcode.

An attempt is left commented in *kernel_shellcode_library/SassyKitdi-master/src/payloads/second_stage/src/lib.rs*, see the user_mode_transition function and the usermode_cmd branch.

Basically, the context switch will always end with the privileged instructions *swapgs* and *sysret* (detailed documentation for these instructions is available online [31]).

The reverse engineering part is about setting the appropriate values in registers before calling these instructions. The only one certain is RCX register, whose value will be copied to RIP after *sysret*, so it must point to the area in which there is the user-mode shellcode... but, is the area allocated from kernel-mode using *ExAllocatePool* executable in user-mode? This question is left open for future developments about this technique.

5.4.3 Undocumented exported functions (like a rootkit)

There are listed here some legacy methods, hard to be made portable, and maybe with low reliability because EDRs could have hooked them because of their usage in past rootkits:

- *KeInitializeApc* + *KeInsertQueueApc* [47], it should be one of the legit ways that Windows uses to start new threads in active processes;
- *RtlCreateUserThread* [48];
- *KeUserModeCallback* [49], it is a "reverse system call" resulted from the refactoring of Windows code for graphics from kernel mode to user mode.

5.4.4 Heresy's gate (undocumented and not-exported functions)

Another time that zerosum0x0 made awesome research.

According to his article [50], "*you can think about Heresy's Gate as the generic methodology to dynamically generate and execute kernel-mode syscall stubs that are not exported by ntoskrnl.exe*".

It is called like that because it allows for the usage of kernel functions which are not exported by the kernel itself, i.e., internal kernel functions: if the kernel is the Hell, Heresy is at a lower realm [51].

After getting access to not-exported kernel functions, it takes some reverse engineering to choose the approach to use.

zerosum0x0 chose to use worker factories [52], which are Windows' thread pools. More details are described in his article.

This approach needs to be studied, it is not a fast solution but can achieve the maximum reliability, whereas the portability could be hard to implement.

5.5 Test the attack against micro-architectural based protections like Intel TDX

The injection relies on the fact that the hypervisor can violate integrity and confidentiality of the virtual domain.

Technologies like *Intel TDX* [53] try to ensure that the hypervisor can only violate availability of a virtual domain (a hypervisor can always deny resources to a virtual machine), but not integrity or confidentiality.

They do that with hardware support for cryptographic operations, such that data in RAM is always encrypted and authenticated, and only the processor sees raw data. So, in theory, the injection should be impossible in presence of such technology, but in practice evidence is always needed:

- to test if the current injection process is blocked;
- to test if there is some trick that the injector can use to fool the checks made by TDX (for example if some cryptographic digests are stored in memory, try to exploit the cryptographic protocol to modify the data while staying undetected).

The second situation can be simulated without having the proper Intel TDX hardware: it could be possible to abstract the cryptographic protocol and express the problem as a more generic cryptographic attack, keeping an eye on hardware limits that would make the actual implementation violate the ideal conditions of the cryptographic protocol.

In other words, the simulation should be enough precise to catch the impact of hardware limitations on the robustness of the cryptographic protocol implemented in Intel TDX, testing if it possible to crack the protocol abusing the arbitrary read/write on the RAM.

6 References

[1] Plextrac, https://plextrac.com/what-is-adversary-emulation-adversary-simulation/, 28/09/2022

[2] Andy Applebaum, Doug Miller, Blake E. Strom, Chris Korban, and Ross Wolf. Intelligent, automated red team emulation. Proceedings of the 32nd Annual Conference on Computer Security Applications, 2016

[3] Oakley, J.G. (2019). The State of Modern Offensive Security. In: Professional Red Teaming. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-4309-1_3

[4] Mitre, https://caldera.mitre.org/, 28/09/2022

[5] Google Project Zero Blog, https://googleprojectzero.blogspot.com/2017/04/pandavirtualization-exploiting-xen.html, 28/09/2022

[6] Volatility Foundation, https://www.volatilityfoundation.org/, 28/09/2022

[7] Metasploit, https://www.metasploit.com/, 28/09/2022

[8] Caldera Docs (getting started), https://caldera.readthedocs.io/en/latest/Getting-started.html, 28/09/2022

[9] Cyberciti, https://www.cyberciti.biz/tips/what-is-devshm-and-its-practical-usage.html, 28/09/2022

[10] Qemu, https://www.qemu.org/, 28/09/2022

[11] Libvirt, https://libvirt.org/, 28/09/2022

[12] Linux-KVM, https://www.linux-kvm.org/page/Main_Page, 28/09/2022

[13] Linux kernel documentation, https://www.kernel.org/doc/html/latest/filesystems/ramfs-rootfs-initramfs.html, 28/09/2022

[14] Project's GitHub repository (VM setup instructions), https://github.com/Shotokhan/adversary-emulation/tree/main/vm, 28/09/2022

[15] Volatility's GitHub repository (ssdt plugin implementation), https://github.com/volatilityfoundation/volatility/blob/master/volatility/plugins/

ssdt.py#L42-L46, 29/09/2022

[16] Shmem2's GitHub repository, https://github.com/rick-heig/devmem2/blob/master/shmem2.c, 29/09/2022

[17] Connor McGarr's blog, https://connormcgarr.github.io/paging/, 29/09/2022

[18] Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., & Strackx, R. (2017). Telling Your Secrets without Page Faults: Stealthy Page {Table-Based} Attacks on Enclaved Execution. In *26th USENIX Security Symposium (USENIX Security 17)* (pp. 1041-1056).

[19] Alice Climent-Pommeret's blog, https://alice.climent-pommeret.red/posts/a-syscall-journey-in-the-windows-kernel/, 29/09/2022

[20] j00ru, https://j00ru.vexillium.org/syscalls/nt/64/, 29/09/2022

[21] Microsoft Docs (hardware priorities), https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/managing-hardware-priorities, 29/09/2022

[22] Microsoft Docs (WDK), https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/, 29/09/2022

[23] Microsoft Docs (debugger), https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/-irql, 29/09/2022

[24] zerosum0x0 blog (SassyKitdi), https://zerosum0x0.blogspot.com/2020/08/sassykitdi-kernel-mode-tcp-sockets.html, 29/09/2022

[25] zerosum0x0 blog (DoublePulsar), https://zerosum0x0.blogspot.com/2017/04/doublepulsar-initial-smb-backdoor-ring.html, 29/09/2022

[26] Pavel Yosifovich, Mark E. Russinovich, Alex Ionescu, David A. Solomon. Windows Internals, Part 1: System architecture, processes, threads, memory management, and more, 7th Edition. Microsoft Press Store, 2017.

[27] mez0, https://mez0.cc/posts/dynamic-api-fnv/, 29/09/2022

[28] Flare kernel shellcode loader's GitHub repository, https://github.com/mandiant/flare-kscldr, 29/09/2022

[29] Microsoft Security Response Center's blog, https://msrc-blog.microsoft.com/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/, 29/09/2022

[30] D. Fu, S. Zhou and C. Cao, "A Windows Rootkit Detection Method Based on Cross-View," 2010 International Conference on E-Product E-Service and E-Entertainment, 2010, pp. 1-3, doi: 10.1109/ICEEE.2010.5660871.

[31] x86 and x64 auto-generated instruction reference (xchg instruction), https://www.felixcloutier.com/x86/xchg#description, 29/09/2022

[32] Unix Stack Exchange, https://unix.stackexchange.com/questions/461689/what-does-physical-address-0-in-x86-linux-contain/461774#461774, 29/09/2022

[33] Linux manpages (dup2), https://man7.org/linux/man-pages/man2/dup2.2.html, 01/10/2022

[34] Atlidakis, Vaggelis, et al. "POSIX abstractions in modern operating systems: The old, the new, and the missing." Proceedings of the Eleventh European Conference on Computer Systems. 2016.

[35] Caldera Docs (terminology), https://caldera.readthedocs.io/en/latest/Learning-the-terminology.html, 01/10/2022

[36] Arora, Rupali, and Rinkle Rani Aggarwal. "Modeling and querying data in mongodb." International Journal of Scientific and Engineering Research 4.7 (2013): 141-144.

[37] Project's GitHub repository (root), https://github.com/Shotokhan/adversary-emulation, 01/10/2022

[38] Inceptor's GitHub repository, https://github.com/klezVirus/inceptor, 03/10/2022

[39] Rubin, Amir, Shay Kels, and Danny Hendler. "Amsi-based detection of malicious powershell code using contextual embeddings." arXiv preprint arXiv:1905.09538 (2019).

[40] Inceptor – Bypass AV-EDR solutions combining well-known techniques, https://github.com/klezVirus/inceptor/blob/main/slides/Inceptor%20-%20Bypass%20AV-EDR%20solutions%20combining%20well%20known%20techniques.pdf, 03/10/2022

[41] CarbonCopy's GitHub repository, https://github.com/paranoidninja/CarbonCopy,

03/10/2022

[42] Brian Caffo, Statistical inference for data science, Leanpub, 2016, 57-60

[43] L. Turnbull and J. Shropshire, "Breakpoints: An analysis of potential hypervisor attack vectors," 2013 Proceedings of IEEE Southeastcon, 2013, pp. 1-6, doi: 10.1109/SECON.2013.6567516.

[44] Bell Software, https://bell-sw.com/announcements/2022/06/29/linux-distributions-for-server-and-cloud-overview/, 03/10/2022

[45] Microsoft Docs (scheduled tasks), https://learn.microsoft.com/en-us/windows/win32/taskschd/task-scheduler-start-page, 03/10/2022

[46] Cyber WTF blog (Windows Registry Analysis – Scheduled Tasks), https://cyber.wtf/2022/06/01/windows-registry-analysis-todays-episode-tasks/, 03/10/2022

[47] Wikileaks (CIA Vault 7), https://wikileaks.org/ciav7p1/cms/page_7995529.html, 03/10/2022

[48] Unknown Cheats Forum, https://www.unknowncheats.me/forum/general-programming-and-reversing/305615-pssetcontextthread-correct-usage.html, 03/10/2022

[49] StormShield, https://www.stormshield.com/news/how-to-run-userland-code-from-the-kernel-on-windows/, 03/10/2022

[50] zerosum0x0 blog (Heresy's Gate), https://zerosum0x0.blogspot.com/2020/06/heresys-gate-kernel-zwntdll-scraping.html, 03/10/2022

[51] Wikipedia (Dante's Inferno), https://en.wikipedia.org/wiki/Inferno_(Dante)#Sixth_Circle_(Heresy), 03/10/2022

[52] Microsoft Docs (thread pools), https://learn.microsoft.com/en-us/windows/win32/procthread/thread-pools, 03/10/2022

[53] Intel (Trust Domain Extension), https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 03/10/2022

[54] Wikipedia (Caldera), https://en.wikipedia.org/wiki/Caldera, 05/10/2022

[55] Wikipedia (Laccolith), https://en.wikipedia.org/wiki/Laccolith, 05/10/2022

[56] Collection of Powershell scripts' GitHub repository, https://github.com/fleschutz/PowerShell/blob/master/Scripts/encrypt-file.ps1, 06/10/2022