Rope: Bypassing Behavioral Detection of Malware with Distributed ROP-driven Execution*

Daniele Cono D'Elia @
Sapienza University of Rome, Italy

Lorenzo Invidia @
Sapienza University of Rome, Italy

Abstract

Rope is a new covert design for distributed malware execution. Rope malware samples are payloads expressed in return-oriented programming and divided into chunks that pre-existing victim processes execute on behalf of the attacker. A transacted NTFS file operates as a covert communication channel for payload distribution and execution orchestration. The design of Rope aims to minimize the indicators of compromise on a victim machine and to go undetected by behavioral analyses of AV and EDR solutions. In its implementation Rope uses techniques that comply with or bypass in original ways the mitigations of Windows 10 that users might enable to harden processes.

1 Introduction

Distributed malware concepts challenge the behavioral detection of Anti-Virus (AV) and Endpoint Detection and Response (EDR) solutions by diluting the temporal and spatial features of a malicious execution across multiple processes.

Several notable malware families seen in the wild adopt a modular design with distinct features delegated to cooperating individual components. Recent research pushed this idea further by splitting the code of a sample into chunks to be run by emulators injected in multiple processes. However, the main shortcomings of these approaches are the conspicuous features and primitives they rely on, which make them an easy prey for state-of-the-art AV and EDR systems and may also conflict with operating system (OS) mitigations.

In this Black Hat USA 2021 Briefings contribution we present Rope, a new covert distributed execution technique for malware. Rope builds on transactional NTFS (TxF) as a non-inspectable covert channel for payload distribution and execution coordination, and on Return-Oriented Programming (ROP) [22] to encode the desired actions. Our technique seeks to minimize the indicators of compromise (IoCs) on the machine: for instance, it does not need any RWX region. Return-oriented programming is central for achieving the desired properties of our design and brings advantages also against code-based signatures.

For the proof-of-concept implementation of Rope we designed a stealth, usable injection primitive that temporarily hijacks a thread and ignites the distributed execution even on hardened processes. Every technique we use in Rope complies with or bypasses in original ways the mitigations presently available in Windows 10 to reduce the attack surface of applications against next-generation malware and exploitation attempts [15].

In more detail, we developed three original bypasses for the protections of Windows Defender Exploit Guard (WDEG): an injection technique that complies with Arbitrary Code Guard and Code Integrity Guard, and a scanning technique based on code reuse to elude both Export and Import Address Filtering policies that a victim process may have opted in to.

To evaluate our concept, we wrote several Rope malware samples that successfully eluded the behavioral analyses of popular state-of-the-art AV and EDR products.

* Technical report (White Paper) for the eponymous Black Hat USA 2021 Briefings presentation. A companion academic paper [8] on Rope will appear in the proceedings of ESORICS 2021.

The present White Paper complements the eponymous Black Hat talk by discussing in more detail the design and several key aspects of the implementation of Rope. Section 2 introduces prior distributed malware execution designs from the literature and discusses some of their shortcomings. Section 3 presents the design of Rope. Section 4 is the core part of this report and covers the implementation details behind Rope. Section 5 reports on the experimental findings and discusses future directions for mitigations and attack variants.

2 Background

A **distributed malware execution** takes place when the attacker partitions a malicious payload in parts to be executed individually by different processes. Designs of this kind can effectively defeat behavioral analyses of AV and EDR solutions, which identify malware by the execution footprint of the externally observable effects (such as network and filesystem activities or invoked API calls). Behavioral analyses complement conventional code-based signature mechanisms that block samples thanks to prior knowledge of their structure.

Distributed designs decouple temporal (i.e., the order in which conspicuous actions are carried out) and spatial (i.e., the entities being observed) properties of an execution by spreading them across multiple processes. The attacker can thus carry out a malicious computation through multiple coordinated processes such that no individual activity of one process matches any of the behavioral signatures of the AV/EDR security solution in place.

Malware samples can create their own set of processes to this end. An early work of Ramilli et al. [21] uses dedicated processes spawned in a way so as to avoid ancestor-descendant relationships among them, otherwise a behavioral analyzer could treat them as a whole. AutoShadow [13] explores automatic malware partitioning at compilation time. More recently, De Gaspari et al. [6] propose a mimicry technique so as to have each process resemble benign ones in the eyes of machine learning-based behavior classifiers for processes.

The main drawback of these approaches is the necessary introduction of a conspicuous number of processes in the system to avoid detection (as high as 18 in [6]). Also, hardened machines may have restrictions on what new processes can do (e.g., accessing the network). Min and Varadharajan [16] propose a dynamic feature distribution approach that alters existing programs—for instance by poisoning open-source libraries or adding trampolines to binary code—and selects the most suitable victim for each desired piece of malware functionality. Static or dynamic code modifications, however, are also conspicuous.

malWASH [11] proposes a radically different design point by partitioning an existing payload in **chunks** to be distributed to running victim processes through a code injection technique, along with an emulator component that coordinates the distributed execution among such victims. Chunks have much finer-grained granularity (e.g., individual basic blocks) than previous solutions, and the partitioning happens in a functionality-agnostic way. In a later attempt to improve the transparency of the design of malWASH, D-TIME [19] replaces the original injection mechanism with remote APC calls to schedule the execution of the chunk emulator, and introduces a covert initialization strategy to set up the shared memory used for coordination without requiring administrator privileges.

Unfortunately, in these schemes the need for RWX memory regions to support the distribution and execution of the chunks, along with the conspicuous injection techniques[1] and the many shared regions they use, limit the applicability of the approach to systems with state-of-the art security products and/or hardening mitigations (we discuss them next) in place.

[1] Even more complex APC variants, like the one seen in DoublePulsar that uses kernel code to schedule APC calls, are now routinely detected by state-of-the-art AVs and EDRs.

3 Design

We propose Rope as a novel distributed malware execution paradigm. In our design, a malicious payload is arranged as a sequence of chunks, each encoded as a fragment of a return-oriented programming chain and generated in a semi-automatic manner.

Figure 1 presents the architecture and building blocks of Rope. Chunks are placed on a shared covert medium and multiple **pre-existing** victim **processes** execute them. We use the Transactional NTFS abstraction of Windows to set up a covert channel for hosting and distributing the payload and for state coordination between the victims. To ignite the execution from the initial **loader**, we designed an advanced technique to stealthily inject a **bootstrap component** while complying with the protections of WDEG [14].

Defenses in place

Our victim machine runs a state-of-the-art AV/EDR solution with behavioral detection. We also assume that the targets we abuse for injection are hardened processes, i.e., processes for which, in addition to standard system mitigations (DEP, Mandatory ASLR, Bottom-up ASLR, etc.), the user enabled application-specific opt-in mitigations of the likes of:

- *Arbitrary Code Guard* (ACG), which protects from executing dynamically generated code by preventing memory from being marked as executable[2];
- *Code Integrity Guard* (CIG), which requires that dynamically loaded modules be digitally signed by Microsoft;
- *Export Address Filtering* (EAF), which prevents code not backed by an image on disk (e.g., injected) from looking up the export address table (EAT) of loaded modules to locate API functions;
- *Import Address Filtering* (IAF), which prevents code not backed by an image on disk from looking up the import address table (IAT) of the program, e.g., to hijack API calls or more simply to locate common API functions;
- *CallerCheck*, *StackPivot*, and (32-bit-only) *SimExec*, which are ROP-specific mitigations that validate call and return sites for uses of *sensitive* APIs, i.e., those being monitored [14] by WDEG as they involve memory allocation, memory-mapped files, processes, etc.

Properties

In the design of Rope we seek to minimize the IoCs and conspicuous features of the distributed execution. The choice of ROP allows us to avoid allocating or modifying executable memories, and also defeats standard signature-based code analyses as explored in previous research [20], and even dynamic code analyses without ROP-specific provisions [10, 7].

We use a transacted file `ROP-TxF` as covert communication channel to distribute the ROP chain to victim processes and for state coordination purposes. The use of TxF implies that only processes that have a handle to the NTFS transaction can inspect the contents of the file, hence AV/EDRs would not be able to access it directly. Unlike malWASH and D-TIME, we use a single transacted file that can be mapped or copied back and forth from memory instead of multiple shared memories each having different R/W/X permissions in order to support chunk execution as in the address space of the original standalone sample.

[2] ACG presently may not prevent remote processes from allocating RWX memory (but will stop local allocations from injected code) nor can coexist with ROP mitigations, yet in Rope we anticipate further enhancements by considering a stronger ACG form capable of doing so to be in place on the victim.

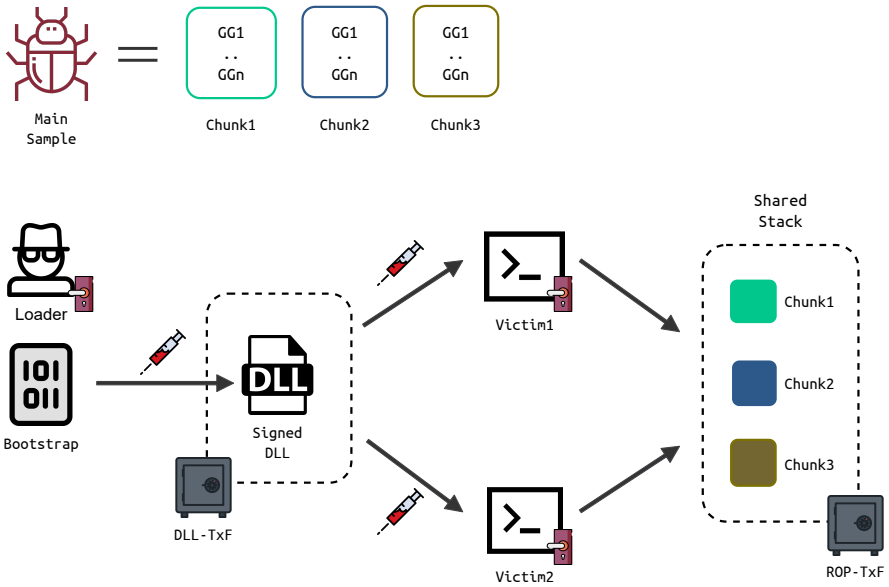

Figure 1 Architecture of Rope. The payload is arranged in chunks that we place on a shared stack, the `ROP-TxF`, accessed by victim processes. We use another transacted file, the `DLL-TxF`, to deliver the bootstrap component. The loader component carries out the injection on each victim.

We envision two **execution modes**: one with explicit coordination of the chunks provided by the Rope runtime, and a simpler one where chunks execute independently (similarly to feature-distributed malware and early multi-process malware designs, and possibly leveraging characteristics of the victim processes) and coordinate on their own if needed.

A key aspect to make our approach practical is a new usable and stealth injection primitive to bootstrap the Rope execution in the context of victim processes. We detail this and other implementation aspects throughout the next section.

4 Implementation

This section covers key aspects of our implementation, including original bypasses for WDEG mitigations. We **responsibly disclosed** our findings to the Microsoft Security Response Center in February 2021 along with actionable suggestions for countermeasures. The vendor concluded that none of the three techniques that we reported met their bar for servicing.

We start with the description of the injection primitive and the bootstrap component that we place in victim processes. We then cover the generation of ROP code and how we use the `ROP-TxF` covert communication channel to distribute the chain. Next, we move to the problems of API resolution in the context of the victim and of hook evasion to further hamper AV/EDR detections; in the process, we also detail how to make API calls in a "friendly" manner for the ROP mitigations of WDEG. We conclude the section by presenting to our readers further practical considerations for the implementation of Rope malware instances.

4.1 Bootstrap Component and Injection Primitive

The execution of the ROP chunks, together with any explicit synchronization actions, is scheduled by a **bootstrap component** running in the context of each injected victim process. This pivotal element of Rope has the role of mapping the ROP stack, fetching the next chunk, and setting up the execution.

The bootstrap component can be expressed as a standalone ROP chain or, depending on the capabilities of the injection primitive, even as a short shellcode. The injection technique that we designed allowed us to implement either scenario in a covert fashion, hiding the component within a signed Microsoft DLL in a perfectly CIG-compliant manner. Furthermore, we hijack a thread only temporarily and promptly restore its normal operation once we scheduled the bootstrap component as an internal application thread.

For the injection, we revamped and extended the Phantom DLL Hollowing technique originally proposed by Forrest Orr[3], which is currently blocked by WDEG when attempting to execute `NtCreateSection` on a TxF-ed DLL in the context of a CIG-enabled process.

In particular, we create a transacted file `DLL-TxF` on such a DLL (e.g., we may pick one at random from those shipped with Windows), make the changes to its contents to host the bootstrap component as code (i.e., shellcode inside the `.text` section) or data (i.e., ROP chain), and force the victim to load it using a bypass technique for Windows mitigations.

To get around WDEG, we use the loader component for creating the `DLL-TxF` transacted file and a Section object over it with `NtCreateSection`, and then duplicate a handle to the Section for the victim. Next, we make the victim execute a ROP chain that calls `NtMapViewOfSection` on the duplicated handle, effectively loading the modified signed DLL in its own address space in compliance with all WDEG mitigations. In fact, at the moment of writing `NtMapViewOfSection` is not among the sensitive APIs monitored by WDEG. The loader subcomponent of the loader identifies suitable victim processes according to some attacker-defined policy (e.g., network access permissions, actions that the distributed payload needs to carry). Then for each of them we execute the following steps:

- The injector enumerates the victim's threads and suspends one of them, saving the CPU state in a `CONTEXT` object for later restore;
- The injector updates the value of the stack pointer and places on the victim's stack area the saved `CONTEXT` object and a short ROP chain that is responsible for making the victim load the `DLL-TxF` with `NtMapViewOfSection`;
- The injector then updates also the instruction pointer in the new CPU state so as to refer to the first gadget in the chain. Finally, it resumes the thread;
- The chain loads the transacted DLL, then executes a sequence that spawns the bootstrap component. Here we simply use `CreateThread` to instantiate a separate execution unit;
- Upon termination of the sequence, the chain restores the old `CONTEXT` from the stack with `NtContinue`, and the victim thread resumes its work from where it left off.

Whenever having an additional thread in the context of the victim process turns out to be suspicious due to the nature of the specific application, one can alternatively schedule an APC call or use other methods. Note that in general internal threads and self APCs are not suspicious elements for AV/EDR solutions, while their remote counterparts very likely are.

The bootstrap component is in turn responsible for loading the `ROP-TxF` contents. According to the TxF interface, a process can inspect a transacted file only if it has a handle to it. As such handles are process-exclusive, we have to perform TxF handle initialization and duplication from the loader component, so the bootstrap component of each victim will operate on its own handle with a workflow similar to the one used for the injection. Listing 1 sketches the main steps for the creation and duplication of the `ROP-TxF` handles.

We defer to Section 4.3 the discussion of how the bootstrap component can solve the API needed to load the `ROP-TxF` contents. The upcoming section instead will detail how we handle the shared `ROP-TxF` stack contents.

[3] `https://www.forrest-orr.net/post/malicious-memory-artifacts-part-i-dll-hollowing`.

```
char TxfPath[MAX_BUFF]  = {0};
HANDLE hTxf, hTxfFile;  // hTxfFile is our ROP-TxF
OBJECT_ATTRIBUTES ObjAttr = { sizeof(OBJECT_ATTRIBUTES) };

NTSTATUS NtStatus = NtCreateTransaction(
        &hTxf,
        TRANSACTION_ALL_ACCESS,
        &ObjAttr,
        nullptr, nullptr, 0, 0, 0, nullptr, nullptr);

hTxfFile = pCreateFileTransacted( // API solved covertly
        TxfPath,
        GENERIC_WRITE | GENERIC_READ,
        0, nullptr,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL,
        nullptr,
        hTxf,
        nullptr, nullptr);

for (int i = 0; i < victims; i++) {
  HANDLE hTxfRemote, hTxfFileRemote;
  if (!pDuplicateHandle(
       (HANDLE)-1,                // loader process
       hTxf,
       proc[i].hProcess,          // victim process
       &hTxfRemote,
       0,                         // ignored
       FALSE,                     // no inheritance
       DUPLICATE_SAME_ACCESS) error();
  if (!pDuplicateHandle(
       (HANDLE)-1,
       hTxfFile,
       proc[i].hProcess,
       &hTxfFileRemote,
       0, FALSE, DUPLICATE_SAME_ACCESS) error();

  /* proceeds with injection and other preparatory tasks */
  [...]
}
```

Listing 1 Creation and duplication of `ROP-TxF` handles

4.2 Shared Stack and ROP Payload Generation

Our injection technique allows us to covertly load an altered signed DLL into processes hardened with ACG and CIG, yet there is a small price to pay: unlike classic DLL injection attacks, the `DLL-TxF` will get loaded at possibly different addresses in each victim process.

This implies that in order to execute a chunk of the ROP chain, each victim process will have to patch it first so as to relocate the gadget addresses according to the base address of the `DLL-TxF` in its address space. The bootstrap component[4] is responsible for this task.

An alternative would be to borrow gadgets from a Windows DLL already in use by all victim processes: while our design does not prevent this possibility, we believe that the flexibility that a modifiable DLL can grant is higher (for instance, we can add missing or handy ROP gadgets in code caves of the `DLL-TxF`).

The bootstrap component has two alternatives in order to lay out in memory the ROP chain containing the chunks we wish to execute:

[4] When implemented as a ROP chain instead of shellcode, also such chain would require relocation. With modifications to relocation information of the PE header of the `DLL-TxF` we can have the loader do it.

(1) Mapping the `ROP-TxF` transacted file in memory and executing the chain directly from there, with any changes readily visible to other processes that also have a memory-mapped `ROP-TxF`. The patching happens directly inside the transacted file, and the bootstrap component annotates each involved chunk with the offset used for patching (i.e., the difference between preferred and actual DLL base address), so that any process that may need to execute such a chunk later can account for it during their own patching.
(2) Placing the chain on a heap region, filling it by reading the desired chain portions from the `ROP-TxF`, and writing them back when permanent changes are needed (e.g., to transfer program values between processes). In this case, the patching happens only in the own memory of the victim process, and is reverted upon any write-back operation to the file.

The two strategies have different trade-offs in terms of efficiency and conspicuousness, and the bootstrap component may follow either. Due to the nature of ROP payloads, we can use a single region to host any kind of storage for the payload, as read and write permissions will suffice for both "ROP code" and data.

Another key aspect is how to generate ROP code for the chunks, which can be a daunting prospect when done entirely manually [1]. We assume as working scenario a payload written in C. To ease the translation to ROP code, we propose initially to add one level of indirection around local variables: in other words, all stack-allocated variables should be transformed and packed as fields of a single global data structure. In this way, we remove interferences with gadgets in stack usage and we can transparently relocate the storage across the chain by patching the references to the beginning of the data structure.

Next, we compile the code in Visual Studio with optimizations and stack protections disabled, obtaining an output that closely resembles a shellcode (although it should be noted that this strategy is not optimal in terms of code size). We then partially automate its translation by looking up gadgets in the `DLL-TxF` for each of the involved instructions, adding missing gadgets to `.text` caves, and laying out a chain skeleton. Finally, the attacker completes it by solving labels for relative offsets for control transfer sequences (i.e., branches and calls to ROP subroutines) and adding chunk delimiters. As future work, we plan to adapt the binary rewriter component of the Raindrop ROP obfuscator [3] to fully automate these tasks.

4.3 API: Resolution, Mitigations for Calls, Hook Evasion

The bootstrap component and the distributed ROP payload inevitably need to carry out actions through the use of standard Windows APIs. As both components are operating in the context of victim processes, there are three problems to consider for an attacker:

❶ How to locate the addresses of the desired APIs within hardened processes;
❷ The ROP mitigations of Windows that shepherd calls to sensitive APIs;
❸ API hooking mechanisms that security solutions may have deployed.

Problem ❶

A naive solution to locate APIs would be to solve all the required ones from the loader and hard-code their addresses in the bootstrap component and in the ROP chunks for the payload. On the other hand, this mechanism would place constraints on the payload: all the required functionality must be known upfront, hindering complex malware designs where, e.g., a sample receives from the network a functional update that exercises new APIs. Also, the loader would have to import the APIs or solve them dynamically by loading the enclosing DLLs—which could be conspicuous: think of, e.g., network or crypto-related libraries—while victims may have instead already loaded such libraries for their own activities.

To solve APIs from the victim processes, we came up with two alternative solutions involving two new bypasses for as many Windows mitigations. We get around the EAF and IAF mitigations (Section 3) of WDEG, which prevent code not backed from an image on disk from scanning, respectively, the Export Address Table (EAT) of any loaded DLL and the Import Address Table (IAT) of the victim executable [9, 14]. With either bypass we are able to locate covertly any required APIs just like shellcodes would do in non-hardened processes.

Both EAF and IAF shepherd memory accesses to the sensitive regions hosting, respectively, export and import information by using guard pages. The key to getting around both mitigations is to trick the guard page handler into believing that the code attempting to access the EAT/IAT is legitimate code. In fact, code that belongs to the main executable or any DLL is allowed to scan the EAT/IAT for the current policy employed by WDEG.

What we leverage is an arbitrary memory read primitive borrowed from a legitimate module in the form of a ROP gadget. Note that our new injection primitive does not follow standard Windows loading, hence gadgets from the `DLL-TxF` are not suitable for EAF/IAF bypass. However, the kind of gadget that we need is really simple and we can find it in nearly any Windows library that the victims load. A suitable gadget, for instance, is:

```
// 8b 00           mov eax, dword ptr [eax]
// c3              ret
```

To locate a gadget we inspect the PEB of the process, which is not guarded by either mitigation, and look up the base address of `kernel32.dll`, which is loaded by almost any Windows process by default. Then we execute a scanning sequence to look up the bytes `8b 00 c3` for the gadget in the `.text` section of the DLL. In our implementation each victim performs its own search. However, for simplicity one may also have the loader look up the gadget and hard-code its address in the bootstrap component and in the `ROP-TxF`.

We use a standard EAT/IAT scanning procedure to look up function names and corresponding addresses, replacing read accesses to the EAT/IAT region with a wrapping sequence semantically equivalent to the subroutine reported in Listing 2.

```
1 DWORD readp(LPBYTE target, DWORD GADGET_read){
2     DWORD res = NULL;
3     __asm{
4         mov eax, target;
5         call GADGET_read; // address of kernel32.dll gadget
6         mov res, eax;
7     }
8     return res; // reads 4 bytes at a time
9 }
```

Listing 2 ROP-based primitive for EAF/IAF-compliant memory scanning

This subroutine can be used as-is in shellcode, and is even simpler to embed in ROP chains as we just need to use `GADGET_read` for accessing EAT/IAT locations.

Problem 2

The second problem concerns the ROP mitigations of WDEG that shepherd calls to **sensitive APIs** (e.g., `VirtualAlloc`, `LoadLibraryA`, `CreateFileMappingA`). These mitigations, namely StackPivot, CallerCheck, and SimExec (Section 3), turn out effective in exploitation contexts as the attacker typically hijacks the stack pointer to a location different than the native stack and may face a limited availability of gadgets to deceive mitigations.

In our scenario, however, we can craft chains in a way such that the stack pointer falls into legitimate ranges upon API calls, and operates instead from a heap region or a

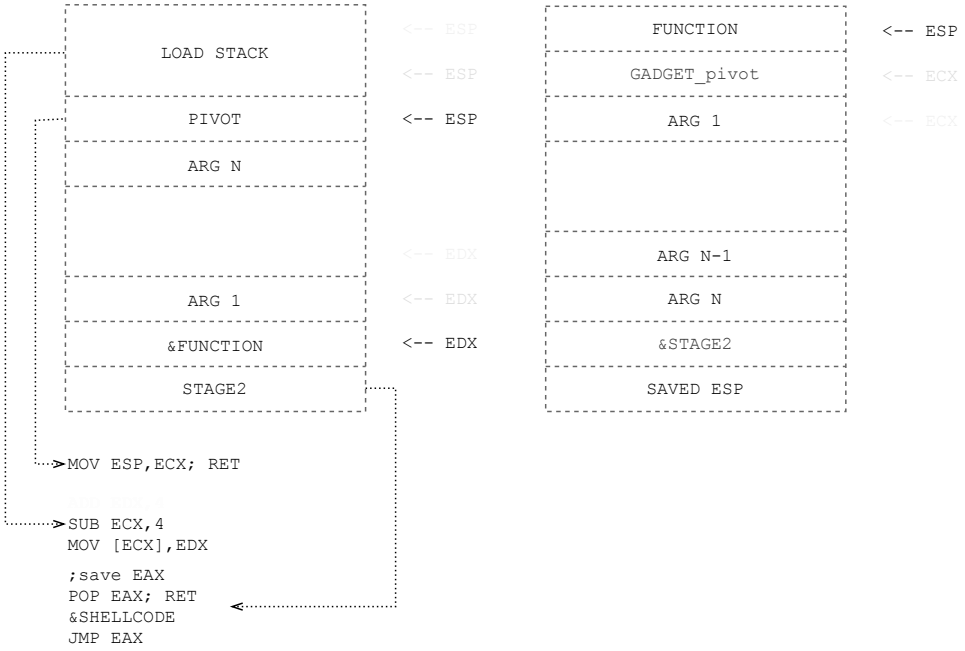

Figure 2 Calling non-sensitive Windows APIs. On the left, the fragment of ROP chain for setting up a call. On the right, the native stack during the call. By `&SHELLCODE` we indicate the return address in the boostrap component when implemented as shellcode. For the ROP-only variant of the bootstrap component we can follow the approach of [3] with custom stack-switching gadgets.

memory-mapped `ROP-TxF` when carrying internal computations. Similar considerations apply to CallerCheck and SimExec as we will detail shortly.

Switching stacks is helpful also with non-sensitive APIs. The careful reader would observe that the normal execution of EIP-driven code can pollute the chain with stack-allocated variables and parameter passing for function calls. Before we discuss chain crafting for calls to sensitive APIs, we first detail how we handle non-sensitive API calls with stack switching.

Figure 2 shows a chain fragment that executes an API call and, for the sake of presentation, returns to the bootstrap component as if the current chunk ended. We use a set of gadgets to fill up the stack for the function call and a gadget to pivot the stack pointer `ESP`, followed by each of the parameters to be loaded. When the function returns, a second fragment of the chain deals with saving the function's return value (i.e., the contents of register `EAX`) and returning into the bootstrap component. In more detail, the chain uses register `ECX` to point to the real program stack, while register `EDX` walks the chain to select each parameter. Concisely, to place a parameter on the stack we increment `EDX` to point to the next slot, decrement `ECX`, and push the argument with a "`MOV [EDX],EDX; MOV [ECX],EDX`"-like sequence. Once everything is arranged, we perform a stack pivoting for `ESP` from the chain to the real stack through a `MOV ESP,ECX; RET` sequence.

At this time, `ESP` points to the entry point of the desired API, whose code takes over once the `RET` instruction is executed. As Windows APIs follow the *stdcall* calling convention [9], the callee is appointed for the cleanup and this operation will modify the stack pointer accordingly upon return. Hence, to the location corresponding to the return address we directly encode a new pivot that alters `ESP` so as to point to the exiting sequence (`&STAGE2` in the figure). Here the return value of the API, available in `EAX`, is saved and the control returns to the bootstrap component, waiting for a new block to execute.

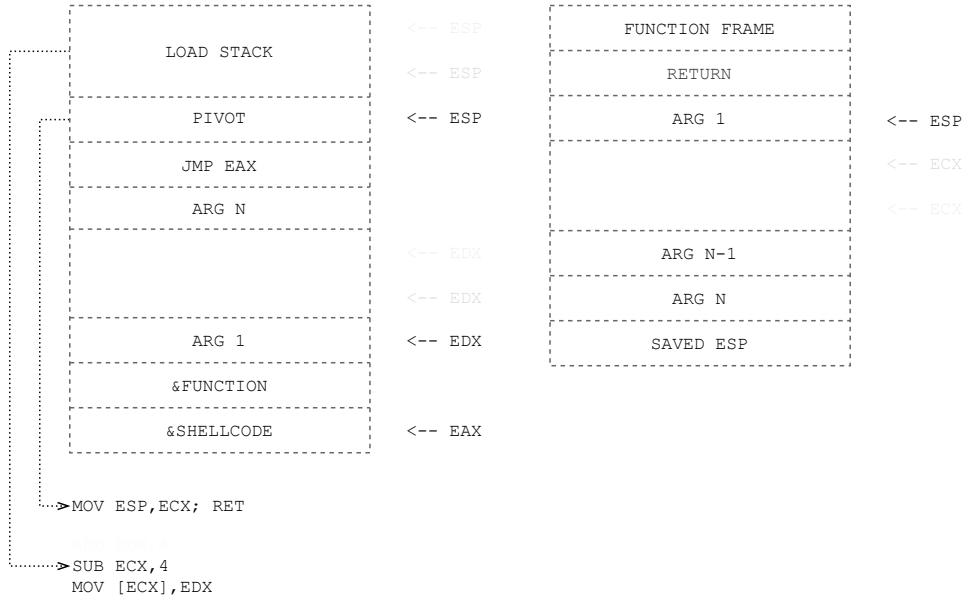

Figure 3 Revised stack switching mechanism for the invocation of sensitive APIs guarded by Windows ROP mitigations.

For sensitive API calls, the requirements to meet are the following:

- To comply with the *StackPivot* mitigation, the function call must occur with a stack pointer value being between the upper and lower stack limits defined in the Thread Environment Block of the caller thread;
- For *CallerCheck*, the sensitive function should return to an address preceded by a `CALL` instruction. Further checks may be performed on the targets of surrounding indirect control transfer instructions [17];
- Lastly, for *SimExec*, the execution must return to a legitimate caller also for subsequent return addresses, since the mitigation walks from the return address and simulates a bunch of instructions in order to deny any return transfers to non-call-preceded targets [18].

We satisfy the first mitigation by design as all calls are executed on the real stack of the process. For the second mitigation, we replace the pivoting gadget of Figure 2 by making the bootstrap component issue the call. We place on the stack a long gadget to stop any backward search, and load the address of the sensitive function into `EDX`. Since the control flow returns back into the bootstrap component, no attack is detected by *SimExec*. Figure 3 depicts the overall flow. For ROP-only versions of the bootstrap component we can use techniques discussed in previous works [17, 4] to get around the ROP mitigations of WDEG.

Problem 3

The problem of evading hooking mechanisms from AV/EDR solutions is a rather well-known hurdle in nearly any malware design, as the use of certain “conspicuous” APIs turns out to be clearly suspicious in the eyes of a behavioral analysis. Reducing the footprint of specific unavoidable actions can be valuable also in the context of distributed malware design.

In Rope this step is mainly relevant for the loader component, which will operate in plain sight as a single process. For the ROP chunks, instead, the distributed nature of the computation seems already well-suited to deceive behavioral detections as studied in prior

literature. However, for the sake of discussion, we note that avoiding API hooks from chunks may still be useful under further specific conditions, e.g., when using very large ROP chunks or for atomic actions that are excessively conspicuous on their own.

For 64-bit code, bypassing user-mode hooks typically involves the use of direct syscalls [9]: this process involves retrieving the syscall ordinals for the Windows version in use, and possibly the removal of any registered post-syscall callbacks[5]. The case of 32-bit code poses a more difficult challenge as the Windows kernel is 64-bit: to bypass 32-bit user-mode hooks, in the loader component we experimented with Heaven's Gate to enter 64-bit mode and invoke the WOW64 versions of the conspicuous APIs we needed, using the WOW64Ext library written by Rolf Rolles[6]. We left to future work attempting direct syscalls after Heaven's Gate to cope with products that may place instrumentation in the WOW64 subsystem.

An alternative to getting around user-mode hooks could be attempting stolen code [12] and similar attacks that bypass instrumentation in the prologue of API functions by emulating their initial instructions, e.g., with own code. However, this avenue may raise compatibility issues for different Windows versions, and also increases the complexity in the implementation especially of low-level APIs (unless the stolen sequences are really short, which would make them amenable to detection and/or mitigation).

4.4 Practical Considerations for Distributed Execution

There are a few unique challenges in orchestrating a distributed malware execution, which we attribute to two main categories. One is *synchronization*, that is, how the different victims can coordinate in transferring both control and program state when processing the chunks. The other is *resource sharing*, that is, how multiple processes can access and share resources (such as file and socket descriptors) for implementing functionalities that in standalone designs are meant to run as a whole; this problem is well studied in malWASH [11]. This section covers implementation solutions that we used in our Rope proofs-of-concept (PoCs).

Synchronization

As we discussed in Section 3, the design of Rope supports two execution modes: one where the Rope bootstrap component aids the chunks by coordinating their execution, and one where those can execute simultaneously and synchronize their actions on their own if needed.

As the first mode requires explicit assistance, in the `ROP-TxF` file hosting the chunks we use a portion of its header contents for bookkeeping relevant aspects of the execution. Such header maintains the PID of the last-executer victim, the index of the current chunk, and a per-victim structure that contains the index of the last chunk to execute when scheduled and the TxF handles that the loader component duplicated for it. The `ROP-TxF` header also encloses additional useful fields such as pointers to solved APIs and scratch locations.

Each victim carries out the execution of the ROP chunks within a loop, starting from the current chunk until the one marked as last for it has been executed. Synchronization and mutual exclusion can be ensured using standard means explored in previous literature [11, 5], e.g., a mutex, a semaphore, or a covert side channel. In the PoC implementations we used a named mutex. When a victim is about to complete its turn and release the lock, the index of the current chunk in the header field is updated accordingly in the `ROP-TxF`.

[5] For the interested reader an excellent reference is `https://www.mdsec.co.uk/2020/12/bypassing-user-mode-hooks-and-direct-invocation-of-system-calls-for-red-teams/`.
[6] Available at `https://github.com/rwfpl/rewolf-wow64ext`.

Resource sharing

As highlighted in [11], a major challenge for multi-process distribution of chunks is the use of handles and other descriptors in the code. Indeed, it should be noted that classic Windows `HANDLE` objects as well as registry and socket ones are unique per process. In Section 4.2 we mentioned that stack variables (but this can apply to global storage as well) should be enclosed in a data structure to ease the ROP encoding, so that every variable dereference will take place through indirection. This means that we can process handle fields in the structure and duplicate them for other processes when resource sharing is required.

Regions

We conclude by discussing further implications of using ROP for encoding the chunks compared to prior solutions based on plain machine instructions scheduled by an emulator. Rope brings the advantage of not having any executable memory mapped for the payload, as we borrow gadgets from legitimate sources. We also use a single memory region for the payload and its data—that is, the memory-mapped or heap-mirrored `ROP-TxF`—instead of having as many extra regions in the victim as the sections of the "distributed" PE file.

The payload chopping design of malWASH and D-TIME requires special handling for block relocations and for the multiple sections of the standalone PE file. Our approach is conceptually simpler as in the ROP encoding that we use all the references to the chain for data accesses and control transfers are relative offsets, similarly to position-independent code also on 32-bit instances. We leave to future work the implementation of ROP rewriting techniques [3] that can fully automate the translation of payloads to Rope, and the addition of heap allocation primitives to dynamically map extra space in the `ROP-TxF` file.

5 Validation and Final Remarks

Methodology

To evaluate the proposed design we have implemented several proof-of-concept (PoC) malware samples and executed them on the latest Windows 10 releases available right before the responsible disclosure process started: 2004, 20H2, and an Insider build from January 2021.

We considered 10 state-of-the-art AV/EDR products (reported in Table 1) and a selection of common applications running with medium Mandatory Integrity Control[7]: Chrome v86.0.4240.198, Skype v8.66.0.77, Telegram v2.4.7, Discord v0.0.309, Steam v2.10.91.91, Firefox v83.0, Dropbox v112.4.321, Adobe Reader DC v19.010.20098, and Opera v73.0.3856.257. We used their 32-bit versions so we could test also prior research concepts on the payloads meant for distribution, as well as better evaluate the API hook evasion techniques and SimExec. However, we remark that Rope can work on 64-bit victims as well.

The Windows releases ran with the following system mitigations enabled: DEP, Mandatory ASLR, Bottom-up ASLR, High-entropy ASLR, SEHOP, and Validate Heap Integrity, as well as Control Flow Guard for compliant applications. For each of the victim processes, we enabled the following opt-in mitigations: CIG, EAF, IAF, and alternatively ACG or the ROP mitigations (StackPivot, CallerCheck, and SimExec) as at the moment they cannot both be enabled at the same time according to the documentation of WDEG.

Notice that ACG and CIG face compatibility issues with non-Microsoft applications whenever they need to load unsigned DLLs or modify code (think of, e.g., browsers). For

[7] `https://docs.microsoft.com/en-us/windows/win32/secauthz/mandatory-integrity-control`.

AV/EDR Product	Version
Avast	2.1.27.0
Bitdefender Total Security	25.0.10.52
Comodo Client Security	12.5.0.8351
Defender	January 2021
Defender ATP	January 2021
Kaspersky Total Security	21.2.16.590
Intezer Endpoint Scanner	1.0.1.8
Malwarebytes	4.1.1.167
Sophos Intercept X	2.0.18
Webroot SecureAnywhere	9.0.29.62

Table 1 List of AV/EDR products used for validating Rope.

this reason, we enabled both mitigations in audit mode so as not to prematurely kill any hardened process, and manually investigated in the Windows Event Logs if the events they reported originated from Rope or from the own activities of the application.

Validation

We now report on our experimental findings, presenting two of the Rope PoCs that we wrote. Each PoC alerts the behavioral detection of the tested security products when executed as a standalone sample. Our expectation is that once encoded and executed in Rope the PoC will no longer raise an alert. For simplicity, both PoCs were crafted to have one chunk for each straight-line C fragment hosting one API call, but the design is open to any payload partitioning strategy. Each PoC is representative of one of the two execution modes envisioned in Section 3. We use the smallest possible number of victim processes (i.e., two) to distribute the payload, choosing from different combinations of the listed applications.

Our first PoC adds a key entry to the Windows registry under the path `HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run` to achieve persistence or alternatively to run `bcdedit` to tamper with the Windows boot process. The two injected victims race to run one chunk at a time, synchronizing through a named mutex for acquiring the execution lock.

The second PoC is a file dropper mimicking a download-and-execute scenario. The first victim process requests a file containing a PowerShell script from a C&C server and downloads it into the `%TEMP%` folder. Later on, the second process reads the file path just written in the `ROP-TxF` by the first process and in turn spawns a new process that executes the script from the command line. The first victim commits its state after the execution of the first 6 chunks and releases the lock, so that the second process can map the ROP chunks in its address space and resume from where the other left off. To some extent, this scenario is also probative of tasks that cannot be broken down into too short execution units.

During the injection stage, we could not complete our tests with two products since their "sandboxes"—in which the Rope loader initially runs—restrained us from interacting with other processes. The issue arises when the loader tries to get a handle on each victim with `OpenProcess`, and the API call fails with error code 5 (i.e., `Access denied`).

We believe the reason is that the loader executable is not signed with a certificate, hence it likely runs with low integrity within the user-space sandbox. We note that this approach can pose compatibility issues with benign applications, and can likely be circumvented using stalling tactics (as the sandboxed execution is slow and the process eventually gets released from it), a different injection primitive, or other evasion techniques. By further investigation,

we observed that with `OpenProcess` or `DuplicateHandle` our samples see flawed outputs or denied accesses, despite having the necessary privileges to obtain or duplicate a handle. With tests of this kind [2], one can deduce when their sample is running within an emulated AV/EDR environment and decide not to expose the malicious behavior, invalidating the detection and a potential cloud submission of data for detailed analysis.

For the other 8 products, 7 did not detect our loader (nor the subsequent distributed actions) already in the version that does not attempt to bypass user-mode hooks. The remaining product would classify our initial actions as suspicious, but resorting to the WOW64 API counterparts via Heaven's Gate in the loader resulted in no detection. As for the Windows mitigations, Rope caused no violations in any of the victim processes.

When we tested D-TIME (Section 2) on the standalone PoCs that we encoded and evaluated with Rope, 7 of the 10 products detected a threat and halted the execution.

Final Remarks

Return-oriented programming represents a valuable ally to hide executable code from the prying eyes of AV/EDR solutions, removing the need for suspicious allocations or modifications of executable memory and hindering also code fingerprinting attempts. The lack of immediate operand values in plain sight for instructions, the intrinsic polymorphism [20] of gadget ordering and contents, and other obfuscation-related properties of ROP [3] pose new challenges for AV/EDRs when applied to the context of distributed malware execution.

Our design is modular and allows for individual replacement of components: for instance, the `DLL-TxF` can be removed if a different stealth injection primitive is available, or we may interleave ROP with other code reuse flavors in the presence of fine-grained ROP mitigations or fingerprinting techniques. The design itself is amenable to further extensions and improvements, for instance by exploring other means for hijacking victims and for spawning the bootstrap component (e.g., we explored IAT hijacking after bypassing IAF).

We anticipate different ways in which defenders could react to Rope and enhance their products. Monitoring handle duplication would provide an initial alert for possibly ongoing malicious activity, especially when involving TxF handles. While extending behavioral detections to account for fine-grained chunks executed alongside normal activities of multiple victims may be challenging for scalability, an alternative point to look at for detecting distributed malware may be treating the activities involving duplicated handles as if they were coming from a single execution unit. Also, the in-memory code scanning solutions of the most sophisticated AV/EDR products could be enhanced with ROP-aware analyses like ROPDissector [7] and ROPMEMU [10], at least in order to be able to spot basic elements of ROP chains such as data operands (e.g., typical constants used in common API calls from malware) or recurrent patterns of gadgets (e.g., those that we use to set up API calls).

We refer our interested readers to the Black Hat USA 2021 Briefings presentation and to the upcoming companion academic paper [8] for the methodological aspects behind Rope and for other details that are not covered in this White Paper.

References

1 Marco Angelini, Graziano Blasilli, Pietro Borrello, Emilio Coppa, Daniele Cono D'Elia, Serena Ferracci, Simone Lenti, and Giuseppe Santucci. ROPMate: Visually Assisting the Creation of ROP-based Exploits. In *Proceedings of the 15th IEEE Symposium on Visualization for Cyber Security*, VizSec '18, 2018. URL: `https://doi.org/10.1109/VIZSEC.2018.8709204`, `doi:10.1109/VIZSEC.2018.8709204`.

2 Jeremy Blackthorne, Alexei Bulazel, Andrew Fasano, Patrick Biernat, and Bülent Yener. AVLeak: Fingerprinting antivirus emulators through black-box testing. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, August 2016. USENIX Association. URL: https://www.usenix.org/conference/woot16/workshop-program/presentation/blackthorne.

3 Pietro Borrello, Emilio Coppa, and Daniele Cono D'Elia. Hiding in the particles: When return-oriented programming meets program obfuscation. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '21. IEEE, 2021. https://arxiv.org/abs/2012.06658.

4 Pietro Borrello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. The ROP needle: Hiding trigger-based injection vectors via code reuse. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 1962–1970, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3297280.3297472.

5 Marcus Botacin, Paulo De Geus, and Andre Gregio. Vanilla malware: vanishing antiviruses by interleaving layers and layers of attacks. *Journal of Computer Virology and Hacking Techniques*, 15, 12 2019. doi:10.1007/s11416-019-00333-y.

6 Fabio De Gaspari, Dorjan Hitaj, Giulio Pagnotta, Lorenzo De Carli, and Luigi V. Mancini. The Naked Sun: Malicious cooperation between benign-looking processes. In Mauro Conti, Jianying Zhou, Emiliano Casalicchio, and Angelo Spognardi, editors, *Applied Cryptography and Network Security*, pages 254–274, Cham, 2020. Springer International Publishing.

7 Daniele Cono D'Elia, Emilio Coppa, Andrea Salvati, and Camil Demetrescu. Static analysis of ROP code. In *Proceedings of the 12th European Workshop on Systems Security*, EuroSec '19, pages 2:1–2:6. ACM, 2019. doi:10.1145/3301417.3312494.

8 Daniele Cono D'Elia, Lorenzo Invidia, and Leonardo Querzoni. Rope: Covert multi-process malware execution with return-oriented programming. In *Computer Security – ESORICS 2021*, Cham, October 2021. Springer International Publishing.

9 Daniele Cono D'Elia, Simone Nicchi, Matteo Mariani, Matteo Marini, and Federico Palmaro. Designing robust API monitoring solutions. *arXiv*, abs/2005.00323, 2020. URL: https://arxiv.org/abs/2005.00323.

10 Mariano Graziano, Davide Balzarotti, and Alain Zidouemba. ROPMEMU: A framework for the analysis of complex code-reuse attacks. In *Proceedings of 11th Asia Conference on Computer and Communications Security*, ASIACCS '16, pages 47–58. ACM, 2016. doi:10.1145/2897845.2897894.

11 Kyriakos K. Ispoglou and Mathias Payer. malWASH: Washing malware to evade dynamic analysis. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, August 2016. USENIX Association. URL: https://www.usenix.org/conference/woot16/workshop-program/presentation/ispoglou.

12 Yuhei Kawakoya, Makoto Iwamura, Eitaro Shioji, and Takeo Hariu. API Chaser: Anti-analysis resistant malware analyzer. In Salvatore J. Stolfo, Angelos Stavrou, and Charles V. Wright, editors, *Research in Attacks, Intrusions, and Defenses*, RAID '13, pages 123–143. Springer Heidelberg, 2013.

13 Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh-Charn Liu. Shadow attacks: automatically evading system-call-behavior based malware detection. *Journal in Computer Virology*, 8(1):1–13, 2012. doi:10.1007/s11416-011-0157-5.

14 Microsoft. Exploit protection reference. https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/exploit-protection-reference?view=o365-worldwide (Accessed: July 19, 2021).

15 Microsoft. Windows Defender Exploit Guard: Reduce the attack surface against next-generation malware, 2017. https://www.microsoft.com/security/blog/2017/10/23/windows-defender-exploit-guard-reduce-the-attack-surface-against-next-generation-malware/.

16 B. Min and V. Varadharajan. Design and analysis of a new feature-distributed malware. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 457–464, 2014. doi:10.1109/TrustCom.2014.58.

17 Z. L. Nemeth. Modern binary attacks and defences in the windows environment – fighting against microsoft emet in seven rounds. In *2015 IEEE 13th International Symposium on Intelligent Systems and Informatics (SISY)*, SYSY '15, pages 275–280, 2015. doi:10.1109/SISY.2015.7325394.

18 Christoforos Ntantogian, Georgios Poulios, Georgios Karopoulos, and Christos Xenakis. Transforming malicious code to ROP gadgets for antivirus evasion. *IET Information Security*, 13(6):570–578, 2019. doi:10.1049/iet-ifs.2018.5386.

19 Jithin Pavithran, Milan Patnaik, and Chester Rebeiro. D-TIME: Distributed threadless independent malware execution for runtime obfuscation. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, August 2019. USENIX Association. URL: https://www.usenix.org/conference/woot19/presentation/pavithran.

20 Giorgos Poulios, Christoforos Ntantogian, and Christos Xenakis. ROPInjector: Using return oriented programming for polymorphism and antivirus evasion. *Black Hat USA*, 2015. https://www.blackhat.com/docs/us-15/materials/us-15-Xenakis-ROPInjector-Using-Return-Oriented-Programming-For-Polymorphism-And-Antivirus-Evasion-wp.pdf.

21 M. Ramilli, M. Bishop, and S. Sun. Multiprocess malware. In *2011 6th International Conference on Malicious and Unwanted Software*, pages 8–13, 2011. doi:10.1109/MALWARE.2011.6112320.

22 Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM. doi:10.1145/1315245.1315313.