The ROP Needle: Hiding Trigger-based Injection Vectors via Code Reuse

Pietro Borrello, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu
Sapienza University of Rome
borrello.1647357@studenti.uniroma1.it
{coppa,delia,demetres}@diag.uniroma1.it

ABSTRACT

In recent years, researchers have come up with proof of concepts of seemingly benign applications such as InstaStock and Jekyll that remain dormant until triggered by an attacker-crafted condition, which activates a malicious behavior, eluding code review and signing mechanisms. In this paper, we make a step forward by describing a stealthy injection vector design approach based on Return Oriented Programming (ROP) code reuse that provides two main novel features: 1) the ability to defer the specification of the malicious behavior until the attack is struck, allowing fine-grained targeting of the malware and reuse of the same infection vector for delivering multiple payloads over time; 2) the ability to conceal the ROP chain that specifies the malicious behavior to an analyst by using encryption. We argue that such an infection vector might be a dangerous weapon in the hands of advanced persistent threat actors. As an additional contribution, we report on a preliminary experimental investigation that seems to suggest that ROP-encoded malicious payloads are likely to pass unnoticed by current security solutions, making ROP an effective malware design ingredient.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; *Operating systems security*; Intrusion detection systems.

KEYWORDS

Malware, APT, code reuse, ROP, antivirus

ACM Reference Format:
Pietro Borrello, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu. 2019. The ROP Needle: Hiding Trigger-based Injection Vectors via Code Reuse. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3297280.3297472

1 INTRODUCTION

Advanced Persistent Threats (APTs) are continuous, stealthy cyber attacks perpetrated by financially or politically motivated coordinated groups against specific private organizations or nations. In order for an APT campaign to be successful, a high degree of covertness must be maintained over time: APT actors possess the technical sophistication to craft malware that can avoid detection by antivirus products (AVs), perform reconnaissance activities, and eventually exfiltrate sensitive data, trying to remain undetected for as long as possible. Similar stealthiness requirements are shared with Remote Access Trojans (RATs)—another kind of cyber menace where many machines are compromised to permit an intruder to take control of them, and eventually have them to work in a symphonic effort to carry out malicious actions on their behalf.

Both classes of attacks typically require an injection vector to compromise a machine and possibly spread to others. Depending on the scenario, an infection may be carried out with or without an unaware user taking an active role in the process: consider, for instance, spear phishing and social engineering attacks, infected physical drives for air gapped systems, or zero-day network protocol vulnerabilities for infection spreading.

After the initial intrusion, the malware may or may not establish a backdoor into the network, obtain user credentials, install apparently legit software components, and perform data exfiltration or lateral movement. For an effective APT campaign to work, attackers should be able to evade antivirus and intrusion detection systems, as well as other security best practices, and deceive incident responders. Attack vectors used in current APT approaches, however, may still be countered using thorough forensic analysis aimed at pinpointing the root cause of the infection, which may lie for instance in a document attached to a spear phishing email. AVs might also have caught up on the infection vector ever since, providing signatures or better detections that can promptly identify similar documents as malicious also outside the context of the attack.

Contributions. This paper aims to explore novel offensive technologies for APT scenarios by studying how code reuse techniques could be used to build injection vectors that are *by design* less prone to signature- or behavioral-based detections by antivirus products. We show how to design vectors that can stay undetected longer, and possibly be reused for different targets over time. As a key idea, we decouple the infection vector from the code that actually performs the injection, which is hidden by means of encryption and code reuse techniques such as Return Oriented Programming (ROP) [36]. We thus yield components that look clean to an analysis system or a human analyst when taken separately. The application is meant to act benignly the whole time, and strike only when the attacker is able to feed it with the right input.

Previous research in the context of iOS devices has explored the idea of building *Jekyll apps* [42] that yield malicious behaviors that are absent in the application that undergoes code reviewing and signing. This is achieved by embedding in a benign application backdoors that are triggered over an encrypted communication with

SAC '19, April 8–12, 2019, Limassol, Cyprus

ACM ISBN 978-1-4503-5933-7/19/04…$15.00
https://doi.org/10.1145/3297280.3297472

a remote server. The ideas presented in this paper improve over Jekyll's approach in a number of ways. In particular, our proposal:

- addresses the significantly more complex defenses adopted by modern mainstream environments such as Windows;
- does not rely on an active remote counterpart, hence avoiding suspect network traffic that could be detected;
- does not require relying on the IP addresses of the intended victims for targeting purposes.

We argue that code reuse techniques are better suited to APT scenarios than unpacking or shellcode injection. In fact, such techniques raise an alert for behavioral detection due to their calls APIs for memory allocation and protection; furthermore, AV vendors continuously generate signatures for them. Preliminary experimental results, discussed in Section 5, seem to suggest that current AVs are instead not quite watchful in the face of ROP sequences.

2 RETURN ORIENTED PROGRAMMING

This section briefly describes Return Oriented Programming, putting it into the technical perspective in which it was incubated.

Memory corruption bugs for memory unsafe languages are one of the oldest problems in software security [38]. Historically, attackers have exploited buffer overflows caused by coding errors to inject their own code into the application and have the instruction pointer jump to it. OS designers reacted by deploying system-level mechanisms such as Data Execution Prevention (DEP) that extend memory management facilities to prevent code execution from writable pages, unless the programmer decides differently for some pages (think, e.g., of just-in-time compilers).

Attackers thus turned their attention to reuse code that is already present elsewhere in the program. Such code can be seen as entire functions, as in return-into-libc attacks [28], or as short instruction sequences that can be chained together to carry out complex tasks. The most famous embodiment of the latter approach is Return Oriented Programming (ROP). In a seminal paper [36] Shacham shows how to use the stack pointer as an instruction pointer: by arranging the addresses of the code sequences to be executed (dubbed *gadgets*) on the stack along with their operands as part of a *ROP chain*, the `ret` instruction present at the end of each gadget instructs the CPU to follow the flow entailed by the chain itself. ROP chains can be used in buffer overflow attacks to perform remote code execution without violating a DEP policy in place.

3 OVERVIEW

In this section we describe a possible deployment scenario that motivates our approach, and the design goals behind it. We discuss the defensive capabilities that a victim might possess, and present the main conceptual steps behind our approach. Section 4 will address several trade-offs in the implementation of our approach.

3.1 Adversarial Model

Our infection vector might undergo a number of inspections once it reaches the target victim. In the adversarial model that we consider, the defenses may be deployed at different architecture levels: the *local* machine where our injection vector gets installed, the *organization* network where the victim machine is connected, and

Mitigation	Description	Discussion
Mandatory ASLR	Randomization of the image address base	Section 3: Detonator
StackPivot	Ensures that the stack has not been redirected for sensitive APIs	Section 3: Detonator
CallerCheck	Ensures that sensitive APIs are invoked by legitimate callers	Section 3: ROP Gadgets
SymExec	Ensures that calls to sensitive APIs return to legitimate callers	Section 3: ROP Gadgets

Table 1: Code reuse defenses in Microsoft Windows 10 (excerpted from [26]) that may hinder our approach.

the *cloud* service provided by a security vendor where an organization might send untrusted or suspicious executables for deeper inspection. We review the main mitigations at these three levels that could hinder the effectiveness of our approach.

Local. As a first level of protection, most modern operating systems implement a series of defenses against malicious behaviors. As discussed in Section 2, DEP is one of the first mechanism devised by OS designers to hinder code injection by denying code execution from writable regions. One approach often used by attackers to circumvent the constraints imposed by DEP is to rely on code reuse techniques. These make use of existing code within an application and thus do not have to inject additional code at run time. To limit such kinds of attacks, one widely adopted mitigation is Address Space Layout Randomization (ASLR) [29]: by randomly arranging the address space positions of key areas of a process, ASLR makes it hard for an attacker to identify gadget addresses. To make our adversarial model stronger (and unlike the default behavior of Microsoft Windows where image base address randomization is an opt-in feature), we assume that ASLR is enforced on every binary executed by the system[1]. Since ASLR may be bypassed by an attacker by leaking at runtime gadget addresses, Microsoft Windows integrates additional advanced mitigations that can make code reuse attacks ineffective. Table 1 provides a summary of those mitigations that may affect our approach, while we refer the reader to [26] for an exhaustive list of the defenses implemented in Microsoft Windows 10. Mitigations not considered in Table 1 are designed for protecting legit applications that are exploited through the use of heap vulnerabilities, exception chain hijacking, loading of untrusted code, as well as other techniques that our approach does not adopt. Additionally, we do not consider mitigations that require compiler assistance, such as *Control Flow Integrity*[2] (CFI), since an attacker may freely opt-out when building the application.

Another defense component that is often deployed on local machines is an antivirus solution. Although AVs are closed-source commercial solutions, we can often learn about their internals through patents and press releases [7]. AV vendors typically combine different detection techniques in their products, which we can be categorized as follows. *Signature-based detection* has historically been used to look for static code patterns that are known to be found in malware: a database of unique flags of known threats is updated over time, and if one of them is found in some portion of the file, the AV deems the file malicious. When an exact file

[1] In Windows terminology, this behavior is called *mandatory ASLR*. Applications not compiled with `/DYNAMICBASE` may crash when launched and thus look suspicious.
[2] *Control Flow Guard* is the CFI implementation supported by Microsoft. Recent works [6, 11, 17] show that it can be bypassed by attackers, especially when cooperation occurs from the application side.

signature match is not found, the AV typically resorts to *heuristic detection* by examining the file for suspicious characteristics. The inspection can take place statically, by having the AV predict the code sequences that the CPU may follow, or dynamically, by letting the initial portion of the execution take place inside a safely emulated environment, e.g., a CPU emulator with stubs for Windows APIs. Heuristics can expose new threats that are variants of known patterns; however, flagging an activity as suspicious is a matter of interpretation, thus threshold-based mechanisms are used. If a file is deemed clean by heuristic detection as well, the executable is eligible to start running. When a user actually launches it, the AV can immediately examine the execution looking for suspicious activities for some time, and repeat such task periodically as the program runs. This process is frequently referred to as *behavioral detection* and requires integration with the operating system, for instance by means of a Windows driver.

To complement and extend the task performed by an AV, a local agent of an endpoint protection system may be placed into the local machine. This component closely monitors the behavior of the system, blocking or sandboxing untrusted applications, and collecting interesting events or data that may be sent to the endpoint protection system server, allowing an organization to perform further analysis and cyber threat intelligence.

Organization. To protect an organization from internal and external attacks, it is common to analyze the traffic within the local network in order to detect and possibly react to ongoing threats. Besides checking the traffic against well-known malicious patterns, our adversarial model assumes that the analyzer is able to perform advanced inspections against shellcodes [31] and sequences that may resemble code reuse attacks [19]. Additionally, the organization may been able to intercept and analyze even encrypted network traffic: several commercial solutions [34] nowadays are able to analyze encrypted traffic, such as SSL connections, in real time by deploying interceptors within the local network. An additional level of protection can be provided by an endpoint protection system. Differently to other components that evaluate a threat based only on the behavior and actions of a single machine, an endpoint protection system is designed to reason also on the behavior of multiple machines within the same network, aiming at detecting even attacks with complex dynamics. This is achieved by deploying agents on the local machines that gather any suspicious data and send them to a centralized server, where sophisticated analyses, often based on machine-learning techniques [9], detect ongoing threats. Network traffic and data collected on local machines may be stored safely by an organization, possibly allowing subsequent forensic investigation in case of cyber incidents.

Cloud. When an application is not deemed malicious by one of the security components within an organization but still raises a suspicion[3], the corresponding executable may be sent by an AV or an endpoint protection system to the security vendor for further inspection. Vendors have a larger amount of resources and tools than a client machine. For instance, the executable might be run in a sandbox equipped with monitoring tools (e.g., based on virtual machine introspection) for some amount of time. Additionally, it might undergo closer inspection by a security analyst, which can resort to state-of-the-art techniques against anti-analysis features of malware: for instance, we assume they can use symbolic execution, which is perceived as the most effective analysis over obfuscated code [5] and has recently been used for RAT dissection in [3].

One crucial consideration about cloud-based security solutions is related to privacy concerns. Organizations, as the ones often targeted by APTs, cannot allow security components to leak sensitive data outside the organization boundaries. For this reason, our adversarial model assumes that only executables may be sent to a security vendor for further inspection, while sensitive data, such as documents, are only analyzed locally. This assumption reflects the policy adopted by several endpoint protection systems (e.g., [9]).

3.2 Goals

Scenario. Our approach aims at providing an injection vector to attackers willing to take control of a system gradually over time, possibly covering their trail with respect to the initial infection mechanism. The attacker has the skills to deliver, by means of our vector, a payload that is not conspicuous against the defenses installed on the target, which may be represented by AVs, firewalls, and endpoint protection as well. The infection vector can reach also other users that are not meant as a target: for such users the vector will do no harm, as a special triggering input must be supplied.

Unlike classic crimeware where an infection takes place and runs out not long after the vector is spread, the attacker here is unwilling to take the risk of having its infection vector detected and eventually fingerprinted by AVs, as this would prevent further (re)use of the weapon. We believe this scenario might be of interest for APT actors, who possess the technical skills and the time span needed to carry out such an attack, as well as for writers of complex RATs that have a strong financial motivation.

Design Goals. We pursue the following main objectives in the design of our approach:

- *Behave like a benign application.* The infection vector could ideally hide in plain sight as part of a software component that an unaware user or an organization would download and use for legitimate purposes. This application should continue to behave as benign even in the crucial moment when the infection is carried out.
- *Look like a benign application.* The infection vector should not only behave as a benign application, but also *appear* as such when analyzed even by a human analyst or any advanced automatic analysis technique. Additionally, data processed by the application which will possibly activate the infection should not raise any suspicion when inspected using automatic tools. For these reasons, the infection vector should be made of two components that look clean to an analysis system or human analyst when taken separately, raising the bar for forensic analysis, which has to reproduce the interaction between the two components in order to figure out how the attack took place.
- *Avoid detection and fingerprinting.* Even when activated, the infection vector should be transparent to mainstream OS defenses for code injection and code reuse attacks, antivirus products, and network monitoring systems. To this aim, we

[3]Organizations may adopt a *default-deny* policy, where any application that is not whitelisted is deemed as suspicious.

take into account a number of defenses that might be in place when the infection vector becomes active. We also want to hinder fingerprinting techniques that could prevent later activations (also on a different target) as a consequence of the defenses being updated: consider, for instance, a network protocol vulnerability that is eventually patched by a vendor.
- *Allow targeted attacks.* Attacks carried out during APT campaigns target specific organizations. As the infection vector may reach in principle also a large audience, it should allow attackers to restrict the infection to intended victims only. Additionally, the vector should not constrain the kind of payload that our weapon can carry: we wish to support the encoding of arbitrary code sequences.
- *Reusability.* The same vector can be reused to attack different organizations over time, provided it gained sufficient popularity to be used in their systems.
- *Channel-agnostic.* The targeted victim that runs the infection vector may be allowed to communicate only with the internal network, or be completely air-gapped. This may happen when the useful application containing the injection vector is installed on an isolated machine. Our design should not depend on the availability of an Internet connection but instead allow an attacker to strike targets whenever any kind of communication channel, even an unconventional one, may be established with them. For instance, an attacker may reach a victim through the help of unaware users that unadvertently propagate the activation input to the isolated machine via physical access[4].

3.3 Approach

We now provide the reader with a high-level view of our approach. We aim at disguising an injection vector as a benign application, which originates from an existing software and is extended with useful features. We rely on ROP to encode a malicious payload that, when the right input is received, is eventually triggered by a detonator component hidden in the normal application logic.

Workflow. Conceptually, we identify five steps in the process:

[S1] *Pick an existing benign application and extend it with extra features.* We start from an existing program that might be of interest for the victim once we augment it with useful features that make it appealing to them, e.g., by meeting their needs. Such features rely to some extent on an open source component that we add to the program and that we later use as a source of ROP gadgets. We argue that if the application is benign on its own, it is likely to stay undetected longer. Ideally, the application may even be trusted enough by the organization to be whitelisted. Otherwise, we expect that the application may be initially sandboxed or temporarily blocked, but then, after a thorough remote inspection from the security vendor, it may be allowed to execute within an organization [9].

[S2] *Encode the malicious behavior as ROP chain.* We rely on code reuse, namely ROP, to encode malicious flows. Compared to unpackers and shellcode injectors, the actions required to trigger a ROP chain are more stealthy in the face of AVs: we do not need to make conspicuous API calls that disable Data Execution Prevention (DEP) for a page, nor to deploy unpacking sequences that might be amenable to fingerprinting or heuristic-based detection.

[S3] *Hide the detonator in plain sight.* In the process of extending the program, we intersperse a detonator component in the application's normal processing logic. When a document is opened in the program, the detonator performs an input-dependent ROP chain decoding, and looks for an activation key in the input to determine whether a valid chain has been extracted. The extraction is done also for benign inputs: the decoded sequence is simply meaningless in such cases.

[S4] *Spread the extended application.* The attacker can resort to a number of strategies (e.g., social engineering, spear phishing) to inform the victim of the availability of a new application. Depending on the scenario, an attacker might want also other people to download it: this might be necessary to build a reputation for the program, and might also turn out convenient (especially for RATs) if the attacker decides to perpetrate the same attack against other victims in the future.

[S5] *Strike by spreading the triggering input.* A pivotal element of our approach is the decoupling between the infection vector and the triggering sequence. The attacker might repeat the strategy from S4 or resort to other channels to prompt the victim to open a document that triggers the infection. This step may possibly take place much later in time than S4. Also, if an open document format is used it is crucial that other viewers or analyzers for that format are not tipped off by it.

Implementation Strategies. To deploy such an approach, a key factor is the placement of the ROP chain. We identify two possible strategies, which we depict in Figure 1:

(1) *The ROP chain is embedded in the application itself:* the detonator extracts a key from the input and uses it for the chain decoding process. Eventually, the decoded chain is checked against a validation key used to determine if the extracted sequence is the intended ROP chain (i.e., the input document was crafted by the attacker) or it should be ignored.
(2) *The ROP chain is shipped as part of the document:* the detonator extracts the chain from the document using a steganography decoding algorithm. Note that if the document format is not an open one, but has rather been created by the attacker as part of the augmented application functionality, an arbitrary extraction sequence can be used. Eventually, the decoded chain is checked against a validation key as in (1).

In Section 4 we will discuss the rationale behind the proposed strategies and address other design choices underlying the five main steps of our injection vector approach.

4 DESIGN CHOICES

Target Application Definition. As mentioned in Section 3.2, an injection vector that behaves and looks like as a benign application is a key element in our approach. A first question to be asked is whether extending a benign application, rather than simply poisoning it, yields any benefits. In the late summer of 2017 the supply chain of a popular utility, CCleaner, was compromised by attackers

[4] The Stuxnet worm—used to sabotage Iranian nuclear facilities—reached its victims via infected USB flash drives, thereby crossing any air gap.

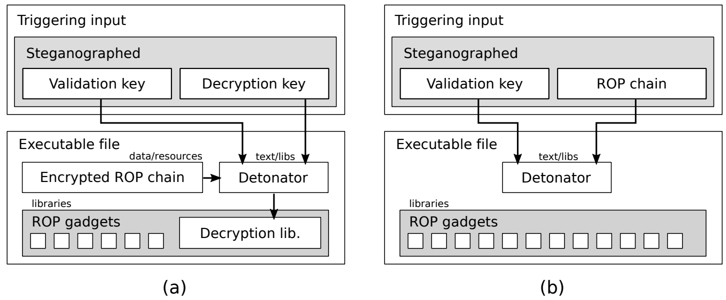

Figure 1: Implementation strategies: ROP chain embedded in (a) the executable file or (b) the input.

that were able to spread poisoned versions of the tool with a valid digital signature. The executables contained a malicious payload featuring a domain generation algorithm and a command and control functionality, which were detected shortly after by security firms. Although the attack could potentially reach a very large user base, the way the payload was appended to the application looked conspicuous in the face of advanced exploit detection technologies[5].

We argue that extending an existing application with additional features, albeit a more laborious task in principle, may give a lot more leeway in concealing the detonation sequence, since code modifications and additions are needed to support them. Also, it may provide more incentive to the victim for its download, especially when the extra features aim at meeting their needs.

A second question concerns instead the nature of the application to modify. An ideal candidate is an application that performs encoding/decoding operations (even better when of cryptographic nature) or heavy-duty input transformations as part of the natural behavior, which may encompass, for instance, compression, format conversion, media reproduction or visualization, secure communications, and so on. The enhancements may or may not extend the range of input file formats supported by the application: if so, an additional format may offer further opportunities in the design of an attack-triggering input, as well as in the choice of the concealing strategy for the ROP chain.

ROP Gadgets. In our approach, the implementation of extra functionalities in the application relies—at least to some extent—on an open source library that is added to the binary. Although the application might in principle already contain sufficient gadgets to encode arbitrary ROP programs, a library offers more possibilities in terms of gadget variety and quality. For instance, an application might legitimately invoke only a subset of the functions of a library, and rely on otherwise unused functions as additional sources of gadgets: adding junk code to the application itself to provide extra gadgets would look indeed more suspicious.

The choice of using an *open source* library is motivated by a number of practical factors. First of all, an official signed version is not necessary: the same library may be compiled in different projects with different compilers, so having multiple versions in principle should not arouse any suspicion. From the point of view of an adversarial system, an AV signature for a popular library used in a possibly malicious context should be written very carefully, otherwise the AV could potentially disrupt the normal functioning of many legit applications.

The availability of its source code offers indeed a simpler way for adding gadgets on demand when compared to closed-source libraries, especially when they belong to Windows itself. More importantly, *static library linking* is a common practice in the Windows realm (e.g., for the sake of compatibility, or to build a portable version of a program), and helps us dodge ASLR using a simple yet effective mechanism within the detonator that we will discuss later on in this section.

The ability to add gadgets on demand is crucial also to bypass two other code reuse mitigations that are shipped with the latest releases of Microsoft Windows. Namely, *CallerCheck* and *SymExec* aim at detecting whether a sensible API has been invoked through a ROP chain. The former checks how a sensible API has been called inside the application binary, while the latter checks what happens when the sensible API returns the control to the application code. In more detail, CallerCheck inspects the return address present on the stack when a sensible API is executed. If such address does not point immediately after a code location containing a `call` or `jump` instruction, then the mitigation raises an alert. Additionally, if the preceding instruction is an indirect control transfer, CallerCheck verifies whether the involved register is consistent with the API address, as the chain could have faked the return address on the stack. To bypass CallerCheck, a ROP chain generated by the attacker has to use gadgets that naturally present a call to a sensitive API or resort to *call-preceded gadgets* [27], i.e., gadgets that contain indirect `call` instructions where the involved register can be easily controlled by the attacker.

SymExec verifies instead that calls to sensitive APIs return to legitimate callers. This is achieved when returning from the API by simulating up to a fixed number of instructions (15 by default) to determine whether the execution is returning control to a ROP chain. To bypass this defense, there are different effective strategies. One natural approach is to craft the chain such that when a sensible API is invoked, the execution is returned to apparently legitimate code sequences of the application that are long enough to mislead the simulation [36]. Another strategy could be to exploit the inaccuracy in the simulation engine. In particular, it has been shown that SymExec stops its simulation whenever a conditional jump or indirect call is met [27]. This happens because SymExec does not track register values precisely.

Albeit CallerCheck and SymExec pose severe constraints to exploit writers[6], they may easily be bypassed when the attacker is free to craft ROP gadgets within an application. Since our approach

[5]https://blog.talosintelligence.com/2017/09/avast-distributes-malware.html.

[6]CallerCheck and SymExec are currently configurable only for 32-bit applications.

```
void patch(LONG_PTR *chain,
           unsigned len) {
 for (unsigned i = 0;
      i < len / sizeof(LONG_PTR);
      i++)
   if (chain[i] < 0)
     chain[i] = chain[i] + &MARK;
}
```

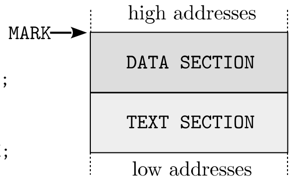

Figure 2: A possible implementation of the online patching mechanism performed by the detonator to make the ROP chain work in the presence of ASLR. `LONG_PTR` is a Windows signed data type used to represent native pointers.

builds on top of the idea of a benign application that integrates a large and complex library, the attacker should been able to disseminate the required gadgets without raising any suspicion.

Detonator. The detonator component is pivotal to our strategy: its activities include retrieving the validation key and—depending on the implementation strategy—the decryption key or the ROP chain from the input document, decoding the payload, checking whether it yields a valid ROP chain using the activation key, and if so transferring at some point the control to the extracted chain.

For the control transfer to take place, the detonator must first patch the chain to make it actually executable. Indeed, since ASLR could be enforced by the system (Section 3.1), the attacker cannot determine beforehand the gadget addresses. However, the ASLR implementation available in Microsoft Windows only randomizes the base address where the image containing the text and data sections of an application is loaded. Hence, the relative distance between the base address and any code gadget, function or data symbol is not affected by the randomization performed by the OS. The attacker can thus exploit this invariant and locally generate a ROP chain where gadget addresses are computed as offsets with respect to one specific function or data symbol. The address of this symbol will be determined by Windows when loading the binary, allowing the detonator to exploit it for computing the correct gadget addresses. Figure 2 shows one possible implementation of the online patching mechanism that could be integrated into the detonator. Albeit more sophisticated implementations could be devised, we remark that this mechanism should resemble common operations performed by benign applications in order to avoid to raise any suspicious and making hard for an AV vendor to define an ad-hoc signature. In our proposal, the `patch` function takes as input the runtime location of the ROP chain and the chain length[7]. Since a chain is composed of data operands and gadget addresses, the detonator must distinguish between these two types and patch only the latter. To this end, the attacker builds the chain by using only non-negative data operands and by computing the gadget offsets with respect to a symbol `MARK` that is placed for instance at the end of the data section. Since the data section is loaded by the Windows loader immediately after the text section[8], the chain contains gadget offsets which are always negative numbers. Function `patch` can thus exploit this property and conditionally patch elements of the chain that contain negative numbers, summing to them the address of the `MARK` symbol to recover the run-time addresses of the gadgets. To overcome the limitation on using only non-negative data operands, the attacker can resort to code gadgets that dynamically build negative constants by arbitrarily composing positive numbers (e.g., using a gadget implementing a subtract operation). Although this may seem a severe disadvantage, the ROP practice faces similar restrictions (e.g., only bytes representing printable characters may be used in the chain).

Besides ASLR, another mitigation that may affect the execution of the ROP chain is *StackPivot*. The main idea behind this defense is to detect whether the stack pointer has been diverted to a memory region different from the expected one whenever a sensible API is executed by the application. To avoid this kind of alert, the detonator should place the ROP chain in the stack before transferring the control to it. If the ROP chain exceeds the size of the stack, then the attacker should implement a multi-stage mechanism using ROP to gradually copy slices of the chain into the stack [41].

In the implementation design space, one may choose to place the detonator in the main application or in the library. We believe the first scenario allows in general for a better blending of the detonator logic within the application logic. On the other hand, a detonator is likely to be overlooked during a preliminary inspection when placed inside a library that has many exports.

The control flow hijacking step in the activation sequence might in principle be concealed also as a vulnerability that the attacker plants in the code. This idea has been explored in previous research in tandem with another vulnerability that leaks memory layout information to a remote active counterpart [42].

Finally, the detonator should provide means for the ROP chain to terminate gracefully, thus resuming the normal benign behavior intended for the application.

ROP Chain Placement. The two scenarios of Figure 1 allow for a variety of trade-offs in the design space.

The main advantage of embedding the ROP chain in the executable file is that the size of the triggering input is kept small. A downside, however, is that the malicious code is chosen beforehand by the attacker. Furthermore, the decryption algorithm embedded in the target application could look suspicious to an analyst, unless it is part of the application's normal stream of activities.

Conversely, embedding the ROP chain in the triggering input allows an attacker to defer the specification of the malicious behavior until the attack is struck, possibly customizing it based on the victim's peculiarities. Also, the same infection vector may also be reused for delivering multiple payloads over time. The main disadvantage is that the size of the input may grow larger, especially if the ROP chain is concealed with steganographic techniques, which need statistical properties that ought to be preserved.

ROP Chain Concealing. If the ROP chain is embedded in the executable file, it has to be kept hidden to an analyst, countering any reconnaissance techniques they may adopt to discover it as it is loaded in memory [37]. A natural approach is to store an encrypted version of the ROP chain in some data or resource section, keeping the decryption key separated from the executable file by embedding it in the input. Since triggering inputs are not released until the attack strikes, an analyst can only explore the application with legit inputs to determine its potential dangerousness. Hence, decryption is most likely performed with the wrong key, yielding arbitrary

[7] When the chain length is not known a priori, this information must be encoded within the triggering input.

[8] Unless position-independent code is used, relative position and distance between text and data sections are chosen by the compiler and cannot be altered by the loader.

bytes that do not resemble a valid ROP chain while the intended malicious payload is not given away.

If the ROP chain is stored in the triggering input, different scenarios may take place: the simplest setting arises when the input has a custom format designed by the attacker. As previously pointed out, this yields plenty of opportunities to conceal the ROP chain along with the input data. Conversely, embedding a chain within a standard format may require steganography techniques, as we elaborate in Section 6. A downside is that the presence of steganographic code in the application may tip off an analyst, unless it is properly blended along with the application's logic.

Key Placement. As we have observed before, if the (encrypted) ROP chain is embedded in the executable file, it is natural to keep the *decryption key* in the input.

Regardless of where the ROP chain is stored, the *validation key* may be kept either in the executable file or in the triggering input. We advocate the latter choice as the lack of information on the validation key, released only when the attack strikes, makes it more difficult for an analyst to reverse-engineer the malicious code, even using automated techniques such as symbolic execution [4]. Furthermore, the effects in terms of file size from embedding a key in a triggering input are likely to be negligible.

We note that, depending on the input format, both the validation key and the decryption key may be embedded in the input without needing steganography, e.g., by exploiting unused padding bytes.

5 PRELIMINARY ASSESSMENT

To assess the feasibility of our architecture, we conducted a preliminary investigation aiming at exploring whether mainstream OS protection mechanisms, in combination with current AV defenses, are equipped to deal with payloads encoded as ROP sequences.

Test Suite. We manually encoded a number of 32-bit ROP programs[9] that implement the online patching mechanism presented in Section 4 and carry out benign actions, either depending on an external input or in any possible execution. Such programs are made of mathematical operations, conditional branches, and calls to Windows API functions; we used the `zlib` compression library as source of gadgets. Our primary interest was to determine whether the means by which execution is carried out (i.e., ROP) is suspicious for the defenses in place, rather than in the nature of such actions: indeed, our goal is to provide an infection vector, leaving to the ability of the attacker the task of encoding actions in the malicious code that do not prompt the AV to intervene.

ROP vs. OS Defenses. We tested modern OS defenses against code reuse using the March 2018 cumulative update of Windows 10. Such defenses may apply at system and app-level [26]. System-level defenses do not affect the execution of our ROP programs: this was expected, as we do not interfere with aspects such as DEP or heap corruption. On the other hand, app-level defenses should be selectively enabled for each installed application: some can be configured for 32-bit applications only, while others may disable functionalities such as child process creation or DLL injection. As discussed throughout the paper, three app-level defenses are particularly relevant to our approach: *CallerCheck* and *SymExec*, which validate API calls, and *StackPivot*, which detects stack redirections at API-call time. Our programs were able to bypass these mitigations without raising any alerts. Compared to exploitation attacks that leverage memory vulnerabilities, our scenario is indeed quite different: the application cooperates in the deployment of the chain, and we have more freedom in the choice of gadgets.

ROP vs. AV Defenses. We then submitted our programs to VirusTotal and they were marked as clean by all the 67 engines available to date. Since VirusTotal may not run all the heavyweight inspection features offered by AV engines, we decided to perform an additional experiment by creating three Windows 10 environments equipped with AVG Antivirus, Avira Antivirus, and KasperskyLab EndPoint Security, respectively. Running our programs did not raise any alerts from these solutions. Although the details of each product are undisclosed, our experiments suggest that—unlike unpacking and shellcode injection—execution patterns typical of ROP did not trigger heuristic detections, which may resort to control flow prediction, code emulation, or a combination of both (Section 3.1). As the payloads from our programs do not perform malicious tasks once activated, the absence of alerts seems to suggest that the ROP execution patterns went unnoticed by behavioral analysis as well.

6 RELATED WORK

Code Reuse Defenses. In the arms race between OS designers and exploit writers, a number of defenses have been proposed by security researchers during the last decade to counter code reuse attacks. [40] divides these solutions into four main categories: *control-flow integrity [14, 39]*, *information hiding* [16, 43], *pointer integrity* [23, 25], and *re-randomization* [22, 44].

A crucial aspect that should be taken into account when evaluating our technique against these countermeasures is the freedom for the attacker to craft the application in a way that can make these defenses ineffective. Indeed, most of the techniques either require that the application be built using a specific compilation toolchain, or implicitly assume that the application will not deliberately cooperate at run time with the attacker. In other words, similarly to previous works [42], our use of ROP is substantially different from what most defensive techniques would expect, effectively undermining their assumptions on the adversarial model. For instance, when the attacker can gain control of the stack several CFI implementations cannot protect a victim [10]. Even when considering mainstream ROP mitigations shipped with the latest releases of Microsoft Windows [26], recent works (e.g., [6, 11, 17]) show that these countermeasures can be bypassed by an attacker, especially when cooperation occurs on the application side.

A different kind of protection scheme is explored in recent research that builds statistical models of the behavior of a program, in order to detect deviations from usual patterns when a ROP attack occurs. Such works often leverage hardware performance counters to detect microarchitectural effects, and employ statical methods to build behavioral profiles [12, 30]. We believe these techniques could raise the bar for our approach; however, they have not been integrated yet in mainstream systems.

Poisoned Applications. A fundamental concept behind this paper is that an attacker is able to spread a malicious code on a victim's

[9] Some of the ROP mitigations shipped with Windows are configurable only for 32-bit binaries. We performed the same tests also on 64-bit binaries.

machine thanks to an apparently innocent application. This strategy has been already explored in the past. One anecdotal example is related to InstaStock [20], an iOS application developed and released on the Apple Store by Charlie Miller in 2011. Although this application appeared as an innocent stock ticker, it had functionalities for interacting with a remote server. During the mandatory app review phase, these interactions were designed by Miller to be innocent. In a later moment the remote server behavior was altered, sending malicious code that was executed in the iOS device exploiting a bug in the Javascript sandboxing component.

Two years later, Jekyll [42] made a step further by demonstrating how to design iOS applications that could be remotely exploitable, allowing an attacker to execute malicious behaviors by carefully rearranging fragments of signed benign code. To bypass the ASLR protection integrated in iOS, Jekill was designed to leak address information to the remote attacker through a network communication. Using this information, the attacker could easily send back a working code reuse attack and perform malicious activities. When considering the design goals discussed in Section 3.2, Jekyll fails at fulfilling several of them. First, the design behind Jekyll is strictly based on a client-server paradigm, which is widely popular in applications targeting mobile devices, but this might not be the case for targets of APT attacks. Victims may be connected to isolated networks, impeding any incoming traffic from a remote server. Even when the victims may be connected to the Internet, organizations may still deploy restrictions on the connections, allowing communications only with trusted IP addresses and thus cutting out the remote server used by Jekyll. Second, even assuming that the target victim can be reached by the attacker from the Internet, the communication can raise suspicion and alert the organization on the ongoing threat. Although Jekyll sends the code reuse attack within an encrypted connection, modern security solutions may be able to decrypt the traffic [34] and detect the code reuse sequence [19]. Third, Jekyll can be configured to deliver exploits only to the intended victims by reasoning on the IP addresses of the incoming connections to the remote server. However, it is common for organizations to place machines behind a NAT gateway that hides the internal network address of the machines. In this scenario, Jekyll is not able to target specific victims within an organization. Although our approach does not provide immediate support for targeting, its design makes it suitable for implementing different targeting strategies (e.g., phishing emails). APT campaigns often use targeting strategies that are tailored to their victims; providing a general strategy to this end seems indeed difficult. Finally, Jekyll has been designed for the iOS environment, where online defenses for code reuse attacks were not deployed. This paper, in contrast, discusses how to take into account the constraints imposed by the most advanced mitigations for code reuse attacks currently shipped in the latest Microsoft Windows releases, discussing how to circumvent them when needed.

More recently, ROPInjector [32] proposed a technique for concealing shellcodes inside a benign application. The key idea is to translate the malicious code into a ROP chain and then append it at the end of the code section. To prevent a monitoring system such as an AV from easily detecting the chain's execution, ROPInjector executes it at the program's exit. Three notable aspects differentiate ROPInjector from our approach: (i) the malicious code is fixed and cannot be remotely customized by an attacker, (ii) the ROP chain is not protected and could be detected by AV heuristics, and (iii) the ROP chain is always executed on the victim's machine, i.e., there is no triggering condition. Nonetheless, the experimental evaluation in [32] highlights two interesting results that are relevant for this paper: (i) AVs emulate only a limited portion of an application code (likely due to time constraints), and (ii) a few elementary mutations on the malicious code can elude AV analysis (at least temporarily) even when considering well-known shellcodes.

Steganography. Steganography [33] is the practice of concealing a piece of data within another piece of data. During the last few decades, the interest for steganography has increased significantly. At the same time, several practical countermeasures have been proposed to detect the use of steganography and also to reconstruct the hidden data from the stego medium. These techniques are commonly referred as *steganalyses*. For an in-depth discussion of modern steganalysis and steganography techniques we refer the reader to [8, 15, 21], where several approaches and trade-offs are discussed. In this paper, we do not impose any constraint on the steganography technique or on the stego medium, leaving wide room for an attacker to choose among the latest techniques that do not raise alerts in the state-of-the-art steganalysis systems.

In our architecture, steganography is a means for hiding a ROP chain within an application input. The idea of concealing malicious code in the context of an application has already been explored in previous research. [1] proposes a trigger-based malware whose malicious code is split into fragments disseminated into the application code via unaligned instructions. Compared to this approach, the use of ROP makes our solution less conspicuous, reducing the probability of detection. Moreover, our triggering condition is given by a special input while [1] manually inserts triggering bugs that transfer the control to the malicious code. Another interesting technique, proposed in the different context of code obfuscation, is RopSteg [24], which translates code into a ROP chain that makes use of *unintended* ROP gadgets present in the application code for its execution. Following the approach behind [1] or [24] in our setting would require choosing the malicious code embedded in the attack vector beforehand, while our approach allows deferring the specification of the malicious behavior until the attack is struck.

7 CONCLUSIONS

In this paper we have discussed a novel malware design approach based on ROP code reuse techniques for turning a benign application into an infection vector that remains dormant until triggered by an attacker-crafted input. The key insight of using a ROP-based concealing strategy over a conventional unpacking or shellcode-based approach is that it is less prone to signature and emulation-based detection from antivirus products, more stealthy against behavioral detections, and harder to detect and inspect for malware analysts.

A main design goal of our architecture is the ability to support the encoding of arbitrary code sequences. While previous research has shown that even in the presence of advanced ROP defenses it is possible to build realistic and Turing-complete gadget sets [6], one missing element in the research landscape is a publicly available, working ROP compiler that meets the needs of real-world attackers, for which building ROP chains remains predominantly a manual task [2, 13]. Conversely, in recent years we have witnessed an increase in the complexity of ROP chains, which moved from being short simple sequences that bypass DEP to inject some shellcode, to very complex behaviors encoded entirely as ROP code [18]. While this trend backs up the assumption that in our scenario a skilled attacker may encode arbitrary behavior even manually, we believe that the development of a full-fledged ROP compiler [35] would be beneficial to the research community, shedding light on aspects of the ROP writing practice that are sometimes overlooked.

Acknowledgements. We thank the anonymous WOOT and SAC referees for their precious suggestions. This work is supported in part by a grant of the Italian Presidency of the Council of Ministers.

REFERENCES

[1] Dennis Andriesse and Herbert Bos. 2014. Instruction-Level Steganography for Covert Trigger-Based Malware. In *Proc. of the 11th Conf. on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'14)*. Springer-Verlag.

[2] Marco Angelini, Graziano Blasilli, Pietro Borrello, Emilio Coppa, Daniele Cono D'Elia, Serena Ferracci, Simone Lenti, and Giuseppe Santucci. 2018. ROPMate: Visually Assisting the Creation of ROP-based Exploits. In *2018 IEEE Symposium on Visualization for Cyber Security (VizSec '18)*.

[3] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Assisting Malware Analysis with Symbolic Execution: A Case Study. In *Proc. of the First Int. Conf. on Cyber Security Cryptography and Machine Learning (CSCML '17)*. Springer International Publishing.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (2018).

[5] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code Obfuscation Against Symbolic Execution Attacks. In *Proc. of 32nd Conf. on Computer Security Applications (ACSAC '16)*. 189–200.

[6] Andrea Biondo, Mauro Conti, and Daniele Lain. 2018. Back To The Epilogue: Evading Control Flow Guard via Unaligned Targets. In *25th Annual Network and Distributed System Security Symposium (NDSS '18)*.

[7] Jeremy Blackthorne, Alexei Bulazel, Andrew Fasano, Patrick Biernat, and Bülent Yener. 2016. AVLeak: Fingerprinting Antivirus Emulators through Black-Box Testing. In *10th USENIX Workshop on Offensive Technologies (WOOT '16)*.

[8] J. Anita Christaline, R. Ramesh, and D. Vaishali. 2015. Critical Review of Image Steganalysis Techniques. *Int. J. Adv. Intell. Paradigms* 7, 3/4 (Dec. 2015), 368–381.

[9] Comodo. 2018. Advanced Endpoint Protection. https://enterprise.comodo.com/whitepaper/AEPWhitePaper09292016.pdf?track=8996&af=7639.

[10] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks. In *Proc. 22nd ACM Conf. on Computer and Comm. Security (CCS '15)*. 952–963.

[11] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium*. 401–416.

[12] Mohamed Elsabagh, Daniel Barbara, Dan Fleck, and Angelos Stavrou. 2017. Detecting ROP with Statistical Learning of Program Characteristics. In *Proc. of 7th ACM Conf. on Data and Appl. Security and Privacy (CODASPY'17)*. ACM, 219–226.

[13] Andreas Follner, Alexandre Bartel, Hui Peng, Yu-Chen Chang, Kyriakos Ispoglou, Mathias Payer, and Eric Bodden. 2016. PSHAPE: Automatically Combining Gadgets for Arbitrary Method Execution. In *Security and Trust Management (STM '16)*. Springer International Publishing, 212–228.

[14] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proc. of the 22nd Int. Conf. on Architectural Support for Progr. Languages and Operating Systems (ASPLOS '17)*. 585–598.

[15] A. Girdhar and V. Kumar. 2018. Comprehensive survey of 3D image steganography techniques. *IET Image Processing* 12, 1 (2018), 1–10.

[16] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proc. of the 21st USENIX Security Symposium*. 475–490.

[17] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *Proc. 2014 IEEE Symp. on Security and Privacy (SP '14)*. 575–589.

[18] Mariano Graziano, Davide Balzarotti, and Alain Zidouemba. 2016. ROPMEMU: A Framework for the Analysis of Complex Code-Reuse Attacks. In *Proc. of 11th Asia Conf. on Computer and Communications Security (ASIA CCS '16)*. 47–58.

[19] Christopher Jämthagen, Linus Karlsson, Paul Stankovski, and Martin Hell. 2014. eavesROP: Listening for ROP Payloads in Data Streams. In *Information Security (ISC '04)*. Springer International Publishing, 413–424.

[20] John Brownlee. 2011. Apple's iOS Javascript Browser Tweak Hacked To Allow Any App To Run Malicious Code. https://www.cultofmac.com/128552/apples-ios-javascript-browser-tweak-hacked-to-allow-any-app-to-run-malicious-code/.

[21] Debina Laishram and Themrichon Tuithung. 2015. A Survey on Digital Image Steganography: Current Trends and Challenges. In *Proc. of 3rd Int. Conf. on Internet of Things and Connected Technologies (ICIoTCT '15)*. SSRN.

[22] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. 2016. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *23rd Annual Network and Distributed System Security Symposium (NDSS '16)*.

[23] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. 2015. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proc. of the 22Nd ACM SIGSAC Conf. on Computer and Communications Security (CCS '15)*. 280–291.

[24] Kangjie Lu, Siyang Xiong, and Debin Gao. 2014. RopSteg: Program Steganography with Return Oriented Programming. In *Proc. of the 4th ACM Conf. on Data and Application Security and Privacy (CODASPY '14)*. 265–272.

[25] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proc. of the 22Nd ACM SIGSAC Conf. on Computer and Communications Security (CCS '15)*. 941–951.

[26] Microsoft. 2017. Customize exploit protection. https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-exploit-guard/customize-exploit-protection.

[27] Z. L. Nemeth. 2015. Modern binary attacks and defences in the Windows environment – Fighting against Microsoft EMET in seven rounds. In *2015 IEEE 13th Int. Symp. on Intelligent Systems and Informatics (SYSY '15)*. 275–280.

[28] Nergal. 2001. The advanced return-into-lib(c) exploits: PaX case study. http://hamsa.cs.northwestern.edu/media/readings/advanced_libc.pdf. Phrack Magazine 58.

[29] PaX Team. 2016. Address Space Layout Randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt.

[30] David Pfaff, Sebastian Hack, and Christian Hammer. 2015. Learning How to Prevent Return-Oriented Programming Efficiently. In *Engineering Secure Software and Systems (ESSoS '15)*. Springer International Publishing, 68–85.

[31] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. 2006. Network–Level Polymorphic Shellcode Detection Using Emulation. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA '06)*.

[32] Giorgos Poulios, Christoforos Ntantogian, and Christos Xenakis. 2015. ROPInjector: Using Return Oriented Programming for Polymorphism and Antivirus Evasion. *Black Hat USA* (2015).

[33] N. Provos and P. Honeyman. 2003. Hide and seek: an introduction to steganography. *IEEE Security Privacy* 1, 3 (May 2003), 32–44.

[34] RSA. 2014. SSL Insight for RSA Security Analytics. https://community.rsa.com/api/core/v3/contents/114532/data?v=1.

[35] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy. In *Proc. 20th USENIX Conf. on Security (SEC'11)*.

[36] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proc. of the 14th ACM Conf. on Computer and Communications Security (CCS '07)*. ACM, 552–561.

[37] Blaine Stancill, Kevin Z. Snow, Nathan Otterness, Fabian Monrose, Lucas Davi, and Ahmad-Reza Sadeghi. 2013. Check My Profile: Leveraging Static Analysis for Fast and Accurate Detection of ROP Gadgets. In *Proc. of 16th Int. Symp. on Research in Attacks, Intrusions, and Defenses (RAID 2013)*. 62–81.

[38] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proc. 2013 IEEE Symp. on Security and Privacy (SP '13)*. 48–62.

[39] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium*. 941–955.

[40] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security (CCS '17)*. 1675–1689.

[41] Sebastian Vogl, Jonas Pfoh, Thomas Kittel, and Claudia Eckert. 2014. Persistent Data-only Malware: Function Hooks without Code. In *21st Annual Network and Distributed System Security Symposium (NDSS '14)*.

[42] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. 2013. Jekyll on iOS: When Benign Apps Become Evil. In *Proc. of the 22Nd USENIX Conf. on Security (SEC'13)*. 559–572.

[43] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z. Snow, Fabian Monrose, and Michalis Polychronakis. 2016. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In *Proc. of the 11th ACM on Asia Conf. on Computer and Communications Security (ASIA CCS '16)*. 35–46.

[44] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-randomization. In *Proc. 12th USENIX Conf. on Operating Systems Design and Implementation (OSDI'16)*. 367–382.