TECHNISCHE UNIVERSITÄT
BRAUNSCHWEIG

Monitoring and Securing Execution at the OS Level

Marinos Tsantekidis

Monitoring and Securing Execution at the OS Level

Marinos Tsantekidis

Monitoring and Securing Execution at the OS Level

Der Fakultät für Elektrotechnik, Informationstechnik, Physik

der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines Doktors

der Ingenieurwissenschaften (Dr.-Ing.)

eingereichte Dissertation

von Marinos Tsantekidis

aus Korinth, Griechenland

eingereicht am: 01.03.2022

mündliche Prüfung am: 23.05.2022

Referent: Prof. Dr. Vassilis Prevelakis

Acknowledgments

I was not alone in writing this thesis. Many people offered their support and love during all these years and this is my chance to show them my gratitude.

First of all, I am truly grateful to my supervisor, Professor Vassilis Prevelakis, for his guidance and encouragement. He has given me the freedom I needed to complete my work, while at the same time nudging me towards interesting directions. Without his precious help, the work presented in this thesis would not have been possible. I am very fortunate to have had such a good supervisor!

Besides my advisor, I would like to thank Prof. Sotiris Ioannidis (my external supervisor) for his time and effort to read my work and Prof. Admela Jukan for chairing my Ph.D. committee.

I take this opportunity to express gratitude to my friends at the Institute of Computer and Network Engineering at TU Braunschweig for their help and support, especially, Mohammad Hamad whose presence and friendship played an important role both in progressing my work and in having a good time in the office and in the every-day life.

I also thank my parents, my brother, and my friends in Greece for their encouragement, love, and support all these years. Words can not express my love and gratitude to all of them.

Finally, I want to thank all of you for reading this work.

Abstract

Throughout the history of computer systems, there has been a never-ending arms race developing, in an effort to gain the upper hand when trying to attack or defend a system. On one hand of the equation, we find adversaries trying to come up with new ways to attack and bypass any security measures in-place. The research community (industrial, academic or otherwise), on the other hand, tries to propose new and safe ways to develop software. The main aim of these attempts is to reduce bugs and vulnerabilities in code, as much as possible. Furthermore, new and innovative ideas for security mechanisms are constantly being developed, in order to defend against any attempt to jeopardize the underlying system. However, complete security of a program is not possible due to increased software complexity and market cost considerations. Consequently, the aforementioned vulnerabilities need to be detected before an attacker gets a chance to take advantage of them, or "catch in the act" the attacker, while trying to actively exploit them.

This dissertation focuses on providing a unified approach that, during execution, ensures the security of the applications and the underlying system by extension. This approach breaks up an application into different components and detects attacks against them. It achieves this, by monitoring all the execution paths among the components. Furthermore, it responds to these attacks properly, by enforcing a set of security policies. These policies ensure a high degree of security, when an application is being executed.

The thesis starts by listing a number of challenges that make its aim difficult and complicated, and its contributions towards delivering a secure operating environment. Continuing, it presents a comprehensive list of the state-of-the-art research and approaches that focus in the same field as what we are trying to achieve, i.e., defend against Code Injection/Reuse Attacks.

The thesis, then, presents the design considerations of the unified approach: (i) its architecture, (ii) the adversary model considered during the stage of development, (iii) the security assumptions for the underlying system, (iv) the testbed used during implementation and testing and (v) the use-cases used to evaluate its applicability and efficiency.

The thesis, also, shows how to enforce access control for library calls at the user-code level, in order to restrict access to specific functions. Furthermore, a library is divided in smaller segments, offering more precise control over any access attempt. The thesis, then, deals with an approach to generate the security policies that can be applied at run-time and dictate to the monitoring sub-system what transitions are permitted.

Following, the thesis presents a kernel-level mechanism that can cooperate with its user-level counterpart, in the attempt to strengthen the underlying system. By modifying the Memory Management Unit, this mechanism divides the memory space of a running application into separate code regions based on page permissions, and determines if a call should be made from one region to another. In this way, we provide a finer-grained level of compartmentalization of a process's memory area, than what is currently the norm. Additionally, the thesis presents a training environment, which has been incorporated in the platform of the THREAT-ARREST EU H2020 research project.

Concluding, the thesis sets a number of requirements and characteristics that any security mechanism should have, in order to be practical and have better chances of gaining wide acceptance and adoption. Moreover, the overall approach is compared to the state-of-the-art mechanisms, which shows that it can stand as a concrete solution among them.

Kurzfassung

Im Laufe der Geschichte der Computersysteme hat sich ein nicht enden wollendes Wettrüsten entwickelt, bei dem es darum geht, die Oberhand zu gewinnen, wenn es darum geht, ein System anzugreifen oder zu verteidigen. Auf der einen Seite der Gleichung stehen die Gegner, die versuchen, neue Wege zu finden, um Angriffe zu starten und die vorhandenen Sicherheitsmaßnahmen zu umgehen. Auf der anderen Seite versucht die (industrielle, akademische oder sonstige) Forschungsgemeinschaft, neue und sichere Wege zur Entwicklung von Software vorzuschlagen. Das Hauptziel dieser Versuche besteht darin, Fehler und Schwachstellen im Code so weit wie möglich zu reduzieren. Außerdem werden ständig neue und innovative Ideen für Sicherheitsmechanismen entwickelt, um jeden Versuch abzuwehren, das zugrunde liegende System zu gefährden. Eine vollständige Sicherheit eines Programms ist jedoch aufgrund der zunehmenden Softwarekomplexität und der Kostenerwägungen des Marktes nicht möglich. Folglich müssen die oben genannten Schwachstellen entdeckt werden, bevor ein Angreifer die Chance hat, sie auszunutzen, oder der Angreifer muss auf frischer Tat ertappt werden, während er versucht, sie aktiv auszunutzen.

Diese Dissertation konzentriert sich auf die Bereitstellung eines einheitlichen Ansatzes, der während der Ausführung die Sicherheit der Anwendungen und des zugrundeliegenden Systems durch Erweiterung gewährleistet. Dieser Ansatz zerlegt eine Anwendung in verschiedene Komponenten und erkennt Angriffe auf diese Komponenten. Dies wird erreicht, indem alle Ausführungspfade zwischen den Komponenten überwacht werden. Außerdem wird auf diese Angriffe angemessen reagiert, indem eine Reihe von Sicherheitsrichtlinien durchgesetzt wird. Diese Richtlinien gewährleisten ein hohes Maß an Sicherheit, wenn eine Anwendung ausgeführt wird.

Die Arbeit beginnt mit einer Auflistung einer Reihe von Herausforderungen, die das Ziel schwierig und kompliziert machen, und ihren Beiträgen zur Bereitstellung einer sicheren Betriebsumgebung. Anschließend wird eine umfassende Liste der neuesten Forschungsergebnisse und -ansätze vorgestellt, die sich auf das gleiche Gebiet konzentrieren wie das, was wir zu erreichen versuchen, nämlich die Verteidigung gegen Code Injection/Reuse Attacks.

Anschließend werden die Designüberlegungen des einheitlichen Ansatzes vorgestellt: (i) seine Architektur, (ii) das in der Entwicklungsphase berücksichtigte Gegner-Modell, (iii) die Sicherheitsannahmen für das zugrundeliegende System, (iv) die während der Implementierung und des Testens verwendete Testumgebung und (v) die zur Bewertung

seiner Anwendbarkeit und Effizienz verwendeten Anwendungsfälle.

In dieser Arbeit wird auch gezeigt, wie die Zugriffskontrolle für Bibliotheksaufrufe auf Benutzercode-Ebene durchgesetzt werden kann, um den Zugriff auf bestimmte Funktionen zu beschränken. Außerdem wird eine Bibliothek in kleinere Segmente unterteilt, was eine genauere Kontrolle über jeden Zugriffsversuch ermöglicht. Die Arbeit befasst sich also mit einem Ansatz zur Erstellung von Sicherheitsrichtlinien, die zur Laufzeit angewendet werden können und dem Überwachungs-Subsystem vorschreiben, welche Übergänge erlaubt sind.

Anschließend wird ein Mechanismus auf Kernel-Ebene vorgestellt, der mit seinem Gegenstück auf Benutzerebene zusammenarbeiten kann, um das zugrunde liegende System zu stärken. Durch eine Modifikation der Memory Management Unit teilt dieser Mechanismus den Speicherbereich einer laufenden Anwendung auf der Grundlage von Seitenberechtigungen in separate Coderegionen auf und bestimmt, ob ein Aufruf von einer Region in eine andere erfolgen soll. Auf diese Weise bieten wir eine feinere Unterteilung des Speicherbereichs eines Prozesses, als es derzeit die Norm ist. Darüber hinaus wird in dieser Arbeit eine Trainingsumgebung vorgestellt, des Forschungsprojekts THREAT-ARREST, welches von der EU im Rahmen des H2020-Programms gefördert wurde, integriert wurde.

Abschließend stellt die Arbeit eine Reihe von Anforderungen und Merkmalen auf, die jeder Sicherheitsmechanismus haben sollte, um praktikabel zu sein und bessere Chancen auf eine breite Akzeptanz und Annahme zu haben. Darüber hinaus wird der Gesamtansatz mit den modernsten Mechanismen verglichen, was zeigt, dass er als konkrete Lösung unter diesen bestehen kann.

Contents

Part I

THE LANDSCAPE

1 Introduction

Throughout the history of computer systems, we have witnessed an unending race between attacks and defenses. As the complexity of systems is ever-increasing, we have not seen a commensurate improvement in code design and development. This results in new challenges and vulnerabilities being discovered every day, while users increasingly require security considerations and provisions for their applications. At the same time, there is a parallel and oftentimes one-step-ahead increase in attackers' capabilities and effectiveness [Pog; Lip]. Hence, systems are becoming ever more vulnerable to attacks at multiple levels.

To make matters worse, unlike the early-day Morris worm, where the intent was just to show off, today there may be more serious incentives behind an attack: monetary gain, industrial espionage, disruption of public benefit services, etc. This means that the stakes, now, are higher and the problem has become more difficult to solve.

Complete security of a program is unfeasible and it is further impaired by two main factors: (a) the increasing complexity of computer systems and (b) market considerations involving cost and time-to-market. These factors generally work to the detriment of the code quality and often lie behind most security vulnerabilities.

Conceding that vulnerable code will be included in production systems, there is a need to either detect these vulnerabilities so that they may be fixed before an adversary can exploit them in a zero-day attack or determine if such a vulnerability is actively being exploited. Systems such as Control Flow Integrity (CFI) [Aba+05], `systrace(8)` [Pro03] or SecModule [KP06], examine each change in the execution of a program by intercepting all calls during run-time at various levels (machine-code instructions, system calls, library calls, etc.). This approach offers a variety of options when dealing with a security breach, without necessarily terminating the offending process (e.g., rewriting function arguments, opening virtual files, etc.). However, many

of these techniques prove to be labor intensive and error prone (due to program preprocessing, application patching, argument evaluation, etc.), as well as computationally expensive for generic use.

1.1 Challenges

Several issues can be identified that make securing a running application (and the underlying system by extension) a difficult task. Here, we shed some light to the most relevant of these challenges, which we later attempt to tackle in this thesis.

1.1.1 Code Diversity

The majority of programs nowadays are dynamically-linked, meaning that some portions of code that are required for the program to run are contained in external shared libraries and are only included in the program at run-time. This has two main advantages:

(a) Commonly-used functions (possibly developed by a third party) are included in a unique shared library that is maintained separately from the main application. This lightens the load of the application developer and can significantly reduce the size of the executable, thus saving disk space.

(b) Several applications running simultaneously can use the functionality provided by a specific shared library, thereby reducing run-time memory usage while enhancing performance.

However, this means that in order an application be fully functional (especially the more complex ones), it must use a diverse set of libraries that are written by different teams with varying degrees of experience in writing secure code. For example, in Table 1.1 we can see the set of libraries - tens of them - that comprise the AnyDesk remote desktop application which provides remote access to personal computers and other devices running the host application. On one hand, the AnyDesk team does not need to concern itself with "reinventing the wheel" with respect to the functionality provided by the other libraries, it just uses them out-of-the-box. On the other hand, however, the developers of each library are responsible for the proper functionality and security of their product, in order the main application work seamlessly and without being vulnerable to attacks.

The more varying code quality libraries are included in an application, the more chances there are of the underlying system ending up in an insecure state. This stems from the fact that if an application is compromised by a vulnerability found in one of the libraries, the entire system (computer, network, etc.) can be compromised as well, as a result.

Several recent studies [BJ21; MSW17; Ven+19; AC19] indicate that, despite the fact that software development teams tend to increasingly incorporate

AnyDesk Libraries			
anydesk (main)	libnss_dns	libnss_mdns4_minimal	libnss_files
libnss_nis	libdatrie	libxcb-dri2	libxcb-present
libnss_compat	libgpg-error	libgcrypt	liblzma
libgraphite2	libuuid	libdrm	libXxf86vm
libxcb-glx	libglapi	libxshmfence	libxcb-sync
libxcb-dri3	libXdmcp	libXau	libexpat
libpcre	libffi	libthai	libharfbuzz
libresolv	libselinux	libz	libxcb-render
libpng12	libpixman	libXcomposite	libXcursor
libXinerama	libc	libgcc_s	libm
libstdc++	libdl	libgmodule	libpangox
libICE	libSM	libXt	libXmu
libGL	libGLU	libgdkglext	libgtkglext
libxkbfile	libXdamage	libXfixes	libXext
libXtst	libXrandr	libXi	libpthread
libxcb-shm	librt	libXrender	libpolkit-gobject
libxcb	libX11	libX11-xcb	libfreetype
libfontconfig	libglib	libgobject	libpango
libpangoft2	libgio	libgdk_pixbuf	libcairo
libatk	libpangocairo	libgdk-x11	libgtk-x11
ld	libsystemd		

TABLE 1.1: Libraries comprising the AnyDesk main application

security practices into their software development processes, organizational challenges - such as lack of funding, limited resources and process support - are frequently responsible for security vulnerabilities still finding their way into production-level software. Moreover, security patches are not being applied fast enough, leaving the underlying system susceptible to attacks for a period of time. Even decades-old vulnerabilities (e.g., SQL injections, buffer overflows, etc.) are still prominent among the top critical security risks today, in relevant lists such as [OWA] and continue to affect popular and widely-used products even after many years of use, as for example the *zlib* compression library which was discovered to be vulnerable to a buffer overflow after a long time, resulting in a denial-of-service or arbitrary code execution condition [Inc05].

1.1.2 Security Policy Development and Enforcement

In order to intercept an attack before it can cause any harm, it is imperative to strictly monitor and control the interactions between the libraries of an application. Security policies are a set of rules that define (a) who should talk to whom, (b) what conditions constitute unexpected program/library behavior which may designate a possible attack, (c) under which circumstances a suspicious event may be considered benign or malicious and finally (d) the

course of action taken when such abnormal behavior is detected. The possibly large number of libraries that a program uses, makes creating such comprehensive security policies - that include the complete list of communication paths among the different libraries, as well as the exact points in execution time - an important task. This task needs to be carefully undertaken as it depends on the complexity of the library/application (i.e. its development as well as its functionality), the way that the paths among the several communicating parts are provided (automatically produced, provided by the developing team, requiring sophisticated analysis tools, etc.) and the effort/time required to track these paths. Another differentiating factor is the evolving computer ecosystem, in which applications/libraries are updated constantly either to provide new and improved functionality or to close security gaps previously discovered. Any mismatches in the configuration of the security policy may lead to mistakes that prevent the application from functioning as intended, or worse, create new security vulnerabilities that potential attackers can exploit.

The enforcement of security policies is another important aspect. Using a centralized evaluation and enforcement engine to authorize the various interactions between libraries is not optimal, as the centralized engine becomes a single point of failure. An attack vector could possibly circumvent the engine and have access to the system as if no access control protection was in place, or at the very least target the engine in order to drain the system's resources leaving it in an unresponsive state.

1.1.3 Behavioral-based Monitoring

Software systems are constantly evolving, as they are regularly being updated with new functionality and security patches throughout their lifecycle. With each new addition/update there is a high chance that a new vulnerability will be introduced and this cycle goes on and on.

> *"Why aren't defects fixed more cleanly? First, even a subtle defect shows itself as a local failure of some kind. In fact, it often has system-wide ramifications, usually non-obvious. Any attempt to fix it with minimum effort will repair the local and obvious, but unless the structure is pure or the documentation very fine, the far-reaching effects of the repair will be overlooked. Second, the repairer is usually not the man who wrote the code, and often he is a junior programmer or trainee."*
>
> ---
>
> The Mythical Man-Month
> Frederick P. Brooks, Jr.
> Addison Wesley Pub. Co., 1975
> 25th Anniversary edition, 2000.

Static and dynamic code analysis are used to detect these vulnerabilities. During static analysis an application's source code is examined before it is executed, taking into consideration a given set of rules or coding standards.

However, not all conditions are met where a vulnerability can be discovered, making dynamic analysis an imperative complimentary technique that examines the program-under-test during or after execution. For example, the fuzzing method provides invalid, unexpected, or random data as inputs to the application aiming to identify implementation faults or bugs that may lead to vulnerabilities. However, these approaches cannot be used alone, as experience shows that applications are still troubled by vulnerabilities.

Signature-based monitoring mechanisms, which are employed by the majority of anti-virus solutions, update a set of static security rules whenever new threats appear. This may not always be adequate, especially when dealing with the exploitation of zero-day vulnerabilities which have never been discovered before or self-mutating viruses, meaning there is no signature matching them yet. This shows the need for behavioral-based monitoring mechanisms that predict and detect at run-time the off-nominal behavior of (the libraries of) an application. However, the definition of the nominal behavior of the different libraries that comprise an application is a challenge, due to the fact that identifying inaccurate or wrong behavior may lead to false positives (the monitoring system considers a library that is behaving nominally as malicious) or false negatives (identify a malicious library as benign)

1.1.4 Security Trade-offs

As already established, complete security of a program is unfeasible. Even if all available mechanisms are adopted and work well together, there are still chances that an attack will happen. So, when deploying a security mechanism, there is an important decision to be made about how it will respond to an incident. If the mechanism is very strict, it will lead to a high rate of false positives and may disrupt the operation of the whole system, affecting this way its availability, usability and overall performance. On the other hand, if the mechanism is overly permissive in order to have less impact on these traits, this will result in a high number of false negatives, where, malicious attempts are considered benign and are left to continue executing. The golden rule needs to be achieved with regards to the response of the security mechanism to an incident and the level of security provided, compared to the availability, usability and performance penalties incurred by employing the mechanism.

Furthermore, the security mechanism itself needs to be secure and correctly configured, in order to not impede the proper functionality of the underlying system, which may otherwise have unforeseen ramifications, e.g. as in the case of the Ariane V booster crash [Lan96], where a simple counter overflow caused an exception that was left uncaught and ultimately resulted in the launch vehicle crashing. In other words, it was the enforcement mechanism that caused the crash.

1.2 Contribution

All the challenges mentioned above show that the need for a unified approach that will ensure the security of the running applications and the underlying system by extension, continuing to deliver a secure operating environment and reacting properly even when an attack is successful, is still prevalent. Although this has been an important research topic for several years, in this thesis we aim to advance the efforts to defend against control flow-hijacking attacks, taking into consideration related work in the area, while improving many of its drawbacks. By observing the control flow transfers between a program's memory segments during run-time, we can deduce either that the program behaves nominally or that it misbehaves because the execution flow is not as expected. If there is a deviation from the expected execution path, this will raise suspicion. However, we do not know if this differentiation in execution is caused by a deliberate attack or by a coding bug. In short, we aim to prove that

> *"By monitoring all transfers of control and by enforcing the orderly inter-segment transfers, we can detect and ensure the proper operation of programs. Hence we can defend against various attacks that attempt to hijack a program's execution."*

Specifically, we develop a security framework that monitors the execution of the software, controls its every communication with any and all ubiquitous external shared libraries and steps in when there is suspicion of foul play. Furthermore, with this approach users can be trained to determine characteristics of such actions in order to be aware and take matters into their own hands if everything else fails.

The contributions of the thesis can be categorized in two levels: a *strategic* one and a *tactical* one. The former category represents our high-level approach on *how* to deal with the challenges identified in the previous Section. The latter category deals with *what* we do in order to materialize our approach.

The strategic level comprises of these contributions:

- Limit the surface available to an attacker when trying to compromise a piece of software, forcing them to exploit vulnerabilities and bugs only found in a small portion of the application. Execution can continue only under specific circumstances (stated in a security policy) [TP19; TP21a; Tsa21; TP21b].
- Establish a Trusted Execution Environment-like region at the memory space of an instrumented application [TP21b].
- Produce security policies, upon which execution depends in order to continue [Tsa18].
- Implement and evaluate the prototype and demonstrate its effectiveness and efficiency by analyzing known, real-world applications and exploits [TP19; TP21a; TP21b; Tsa18].

- Present a training environment where users can look for potential issues or vulnerabilities in code that is distributed in binary form [TP17; TP20].
- Release the prototypes as an open source project to allow kernel and application developers to validate their code with it.

At the same time, the tactical level - i.e., the actions taken to tackle what is described in the list above - includes:

- Presenting a modified Linux kernel that separates the memory of a running application into regions at the granularity of shared libraries/executables [TP19; TP21a; TP21b].
- Developing a novel technique to perform the separation, leveraging the Memory Management Unit (MMU) [TP19; TP21a; TP21b].
- Presenting a novel technique to intercept any attempt to invoke a separate region, based on page faults [TP19; TP21a; TP21b].
- Installing a policy enforcement engine - *gate* (specially crafted library) - "before" each region. This *gate* is a mechanism that allows the controlled entry into a library. The kernel intercepts all calls to a destination library and redirects them through that *gate*. This is similar to the system call entry point provided by most operating systems that distinguish between user and supervisor domains [TP19; TP21a; TP21b].
- Dynamically (un)mapping secure private pages in each separate region. Information stored in these pages is accessible only by a *gate* and only when the CPU executes code within the associated region [TP21b].
- Developing a technique to further break up a library into smaller sections, based on their functionality or even distinct functions. In this way, only the absolutely necessary functions/portions are loaded during execution and the attack surface related to a library is minimized. Additionally, less memory is used by the running program [Tsa21].
- Instrumenting and analyzing the execution of an application (Sophos AntiVirus solution), in order to present our approach for producing a security policy. Based on information uncovered during this procedure, we subsequently compromise the application [Tsa18].

This work has been a part of several EU H2020 research projects: SHARCS under Grant Agreement No. 644571, THREAT-ARREST under Grant Agreement No. 786890, I-BiDaaS under Grant Agreement No. 780787, CONCORDIA under Grant Agreement No. 830927, SmartShip under Grant Agreement No. 823916, SENTINEL under Grant Agreement No. 101021659 and ROXANNE under Grant Agreement No. 833635. Additionally, it was supported by the German Research Foundation (DFG), Controlling Concurrent Change (CCC) project, funding number FOR 1800.

1.3 Layout

This thesis is divided into three parts. The first part provides a landscape for the thesis via three chapters; Chapter 2 gives background information about Code Injection/Reuse Attacks (C[IR]As) and the different vulnerabilities that accompany them, as well as the defense mechanisms which have been proposed to mitigate these attacks. Chapter 3 presents the design of our framework, the software and hardware setup and the use-cases which are employed throughout the thesis. In addition, it introduces several assumptions about the underlying system, as well as the adversaries that it may face.

This framework is detailed in the second part of the thesis, which includes two chapters. Chapter 4 details the user-level part of the framework: how it is implemented, a use-case where it is applied, how it can strengthen its collaboration with the kernel counterpart via finer-grained segmentation and how it can create a policy to be enforced. In Chapter 5, we explain the kernel-level part of the framework, its design and implementation specifics and two situations where it is applied (a use-case analysis and a training environment).

In the last part of the thesis, in Chapter 6, we set a number of requirements that we believe a mechanism needs to have in order to be practical and evaluate the state of the art and compare it to our framework, based on them. Furthermore, we propose our envisaged directions for future research. Finally, we include a list of the research published during the writing of this thesis, of the figures, of the tables and of the bibliography contained in the thesis.

2 Background and Related Work

The use of languages that are not type-safe and the widespread use of pointers that do not support bounds checking, created a seemingly infinite series of buffer overflow attacks and resulted in an arms race to eliminate them. Nevertheless, we still see instances of classical buffer overflow attacks against fairly modern systems (e.g the attack against the time measurement ECU [Ham+18; LSL15]). By mounting such an attack, an adversary can eventually mislead the CPU to jump to an address that was not intended by the running program. This may cause the execution of (i) foreign code injected by the attacker into the address space of the program, (ii) code that already exists within the address space of the victim, or (iii) arbitrary code (junk data, middle of instructions, etc.) that is also located within the address space of the process. This is a direct result of the ability of code to jump anywhere within a process's memory area, as well as the absence of policy checks when a transfer is performed.

2.1 Timeline

Compiler and architectural modifications (e.g., the No-eXecute (NX) bit [WX03] that prevents the execution of code from the heap or stack) have made Code Injection Attacks (CIA) all but impossible.

Instead of trying to inject custom code, the attackers responded by using code already present in a program's memory space, resulting in a new type of attacks, i.e. CRAs. One of the most common forms of CRA is return-to-libc. It first appeared in 1997 [Des97], redirecting the flow of execution in the `libc` library. However, in this attack the adversary could only execute straight-line code, chaining together one function after another, resulting in attacks that are not Turing-complete. Stack smashing protection [Cow+98; Hir03] and randomization techniques such as Address Space Layout Randomization (ASLR) [PaX01], were then introduced to defeat return-to-libc attacks.

In response, researchers proposed more sophisticated approaches - namely Return Oriented Programming (ROP) [Sha07; Che+10; Roe+12] and Jump Oriented Programming (JOP) [Ble+11]. These attack vectors form snippets of code located at predetermined memory addresses, "gadgets", chaining together legitimate commands already in memory. In ROP, each gadget ends with a "return" instruction. When it is reached, it diverts control to the next gadget, eventually forming a sequence of gadgets that perform the unauthorized action desired by the attacker.

The chained execution of gadgets ending in `RET`, has enabled researchers to develop several anti-ROP defenses (eg. [PPK13; Che+14; Fra12]) to detect or prevent it. In order to circumvent these methods, JOP was presented. This attack also uses chained gadgets to execute arbitrary code, but it does not need `RET` instructions to alter the flow of execution. It is based on indirect branches and a dispatcher gadget to steer and chain them together.

A different attack vector targets the data plane, which consists of memory variables not directly used in control-flow transfer instructions. These types of attacks, referred to as non-control data attacks [Che+05], do not require diverting the application's control flow and can be crafted using a systematic construction technique known as Data Oriented Programming (DOP) [Hu+16; Hu+15]. Similar to ROP/JOP, DOP identifies (a) data-oriented gadgets and (b) gadget dispatchers that chain together previously identified gadgets in an arbitrary sequence.

Initially, CRAs were based on the principle that gadgets are located at known addresses in memory. ASLR, however, randomizes the location of data and code regions every time a process is executed. By randomizing code, ASLR makes CRAs more difficult to succeed as they cannot locate already present code, while randomizing data disrupts the redirection of execution flow as CIAs have far less chances to locate potentially injected code. However, Shacham et al [Sha+04] proved that due to low entropy, caused by a small number of bits available for randomization in the 32-bit architecture, a brute-force attack can lead to a memory leak and eventually reveal the location of the randomized segments.

Furthermore, Snow et al. introduced "Just-In-Time" ROP (JIT-ROP) [Sno+13], a technique that exploits the ability to repeatedly abuse a memory disclosure vulnerability to map an application's memory layout on-the-fly, thus bypassing ASLR. Next, it identifies and collects gadgets, and then constructs and delivers a ROP payload based on those gadgets.

Later, Bittau et al. presented a new attack, Blind Return Oriented Programming (BROP) [Bit+14]. BROP works against modern 64-bit Linux with ASLR, NX memory and stack canaries enabled [WC03]. It exploits a single stack vulnerability and uses two techniques to succeed: (a) generalized stack reading, which generalizes a known technique used to leak canaries, to also leak saved return addresses in order to defeat ASLR on x64 even when Position Independent Executables (PIE) are used, and (b) remotely finds enough gadgets to perform the write system call, after which the application's binary can be transferred from memory to the attacker's socket.

2.2 State of the Art

The idea of a *gate* has been applied before. Multics [Mula] operating system uses multiple rings of protection [Mulb; Inc83; SS72] that isolate the most-privileged code from other processes, forming a hierarchical layering. Each process is associated with multiple rings – domains – so it is necessary to change the domain of execution of a process. This way the process can access specific domains only when particular programs are executed. To prevent arbitrary usage, specific "gates" between rings are provided to allow passing from an outer (less-privileged) to an inner (more-privileged) ring, restricting access to resources of one layer from programs of another layer. The change of domain occurs only after the control is transferred to a gate of another domain. Switching to a lower ring requires more access rights as opposed to a higher ring where reduced rights suffice. Downward switching requires a control transfer to a gate of an inner ring, if the transfer is to be allowed, whereas an upward domain switch is an unrestricted transfer that can be performed by any process. Nevertheless, the need-to-know principle cannot be enforced, because if a resource needs to be accessible by a ring `a` but not from another `b`, then `a` needs to be lower than `b`. But, in this case every resource in `b` is accessible in `a`.

Over the years, additional important work has been carried out with respect to defenses against CIAs/CRAs, which we classify in three major categories: (i) randomization, (ii) control and (iii) monitoring. However, randomization mechanisms have two inherent shortcomings: lack of entropy and information leakage. If the entropy is not high enough, they are vulnerable to spray-based attacks [DHM08] or brute-force based attacks [Bit+14]. Furthermore, if a memory corruption vulnerability causes information leakage, randomization protection can be bypassed by JIT-ROP attacks. Attempts to counter JIT-ROP attacks have also been bypassed [Dav+15; Con+15]. CFI-based access control mechanisms depend on two factors to defend against CRAs: (a) how precise the computed Control Flow Graph (CFG) is and (b) how precise the checks of the execution flow are. If the CFI implementation is too coarse (i.e., more permissive CFG) in order to incur lower performance overhead [Dav+14], it may be circumvented by illegal control transfers. At the same time, this leads to a decrease in the precision of the checks of the execution flow, leaving the implementation vulnerable to CRAs. Finer-grained solutions (i.e., more strict CFG) suffer from non-negligible performance overhead and have been proven to be ineffective against CRAs [Eva+15].

2.2.1 Randomization

Under this category fall the mechanisms that propose some kind of randomization in their implementation. They can defend against CIAs/CRAs by rendering the exact location of data/gadgets unpredictable [Lar+14]. These techniques operate at different levels, depending on their design and intended functionality, namely at the function, basic block, or instruction level.

Instruction Set Randomization (ISR) [KKP03; Boy+10] is a generic technique against Code Injection Attacks that is based on diversifying the language the execution environment understands to prevent "foreign" code, which is not expressed in the correct language, from executing. Normally, a CPU architecture provides a runtime that can execute binaries following the CPU's instruction set (x86, ARM, etc.). ISR proposes a runtime that can understand different randomized languages that an attacker cannot possibly know, so they are unable to execute their own code. ISR usually relies on simple cryptographic functions to generate random languages with low performance overhead. On binaries, for instance, a random key (*rnd_key*) can be used to randomize every instruction (*instr*), by employing a simple transformation like *XOR* ($rnd\_key \oplus instr$), while the reverse process is applied by the runtime after fetching bytes to execute and before decoding them to instructions. The runtime can be implemented directly in hardware [Pap+13], or in software [PK10] using virtualization. Other implementations use stronger cryptographic algorithms, like AES [Bar+05], but incur higher performance overhead and operate on larger code blocks.

Backes and Nürnberger developed Oxymoron [BN14] to counter JIT-ROP attacks. Oxymoron combines two methods: fine-grained memory randomization with the ability to share the entire code among other processes. It uses the x86 processor's segmentation feature to disable access to unique indices, which are organized in a translation table. This level of indirection hides the target address of a direct branch from a JIT-ROP attacker, making it infeasible for them to identify the gadgets necessary for the attack.

Venkat, et al. [Ven+16] propose a security defense called HIPStR to thwart ROP attacks. HIPStR performs dynamic randomization of run-time program state, both within and across Instruction Set Architectures (ISA). For any program in execution, HIPStR dynamically randomizes the location of its program state (registers and stack objects) in order to render brute-force attacks infeasible. Furthermore, HIPStR detects a potential break-in attempt via JIT-ROP, and when detected, it migrates execution to a different ISA, in order to limit JIT-ROP attacks.

Chen, et al. [Che+16] developed Remix, a live randomization system for user-space applications and kernel modules. Remix randomly reorders basic blocks within their respective functions at undetermined time intervals, to change the run-time code layout. This way, functions remain at their original, expected locations, while basic blocks are moved around but never cross the function boundaries.

Hiser et al. in [His+12] introduce a novel technique called Instruction Location Randomization (ILR), which randomizes the location of every instruction within an binary, with high entropy, thwarting an attacker's ability to re-use program functionality. ILR adopts an execution model where each instruction has an explicitly specified successor. Thus, each instruction's successor is independent of its location. This model of execution allows instructions to be randomly scattered throughout the memory space.

Hiding the explicit successor information prevents an attacker from predicting the location of an instruction based on the location of another instruction. This model is provided through the use of a process-level virtual machine (PVM) that handles executing the non-sequential, randomized code on the host machine. This approach, however, depends on disassemblers and position-independent code for randomization, which makes the amount of position-dependent code crucial for achieving complete coverage. Also, the external libraries included in the program are not randomized. Another limitation of ILR is that incorrect branch target analysis during the offline stage, may result in false positives. Moreover, it requires a database to store the identified instructions and rewrite rules. Additionally, it suffers from high performance overhead incurred by the PVM, which determines the next instruction to be executed at run-time [YSX16].

Kim, et al. [Kim+15] present a micro-architecture design that can support native execution of control flow randomized software binary, while at the same time preserve the performance of instruction fetch and use of on-chip caches. They use ILR to enhance dependability and security of software against code reuse attacks. For that purpose, they propose an approach named Virtual Control Flow Randomization (VCFR), which introduces an address space randomization/de-randomization interface before the instruction fetch requests are handled by the on-chip L1 instruction cache. The control flow of a binary executable is randomized similar to ILR and presented to the processor execution pipeline. However, the binary instructions are still stored in the memory hierarchy (both on-chip caches and off-chip memory) in the original layout.

[LRL15] presents system call diversification as a method for protecting operating systems and rendering malicious programs ineffective. The idea is to change all the system call numbers in the kernel and in the applications that invoke these system calls. As a result, it becomes much more difficult for a harmful program to access resources of a computer since the new system call interface is not known by the malware. However, system calls are rarely issued directly; applications use libraries to request the system's resources making this approach insufficient to provide security.

Habibi et al. in [Hab+15a] introduce a new form of ROP attack applicable on UAVs, called "stealthy ROP attack", which first executes the attack payload completely and then reconstructs the 'smashed' stack frame before the final return. This way, the victim application continues executing, giving the attacker the upper hand when trying to avoid detection. They also introduce a trampoline technique in this stealth attack that allows the attacker to inject arbitrarily large payload into the application's stack. In addition, they propose a defensive technique – MAVR – to mitigate code-reuse attacks on UAV systems that combines software and hardware techniques. At software level, they propose a fine-grained randomization-based approach that modifies the layout of the executable code and hinders code-reuse attacks. Moreover, the mitigation technique aims at breaking critical factors that are required for this type of attacks, leveraging a specialized hardware design.

Kumar and Kisore [KK14] propose a technique called Function Frame

Runtime Randomization (FFRR). FFRR offers memory layout randomization at runtime and performs randomization at the granularity of individual variables on the stack. Their approach makes the memory location of the program objects on the stack unpredictable, by randomizing the relative distance between any two objects on the stack at runtime. The basic idea is that random variables are introduced, as many as the number of local variables declared in a function. These random values are used to add a random number of words before the local variables are pushed on to the stack function frame at runtime. The random numbers can be generated using a computationally inexpensive technique like Linear Feedback Shift Register (LFSR) during the function frame setup phase.

Kanter and Taylor in [KT13] explore compiler and linker based approaches to increase attacker workload by generating diversity in the binary code associated with a single source, measured by entropy. This is achieved by injecting randomness into the binary image. By inserting code at the start of every logical block and randomizing function layout, it is possible to inject a quantifiable level of entropy based on the source parameters of number of functions, function size, and blocks per function.

Stanley et al. [SXS13] built a technique based on reordering memory layouts. They describe two different ways to mutate an operating system kernel using memory layout randomization to resist kernel-based attacks. They introduce method for randomizing the stack layout of function arguments. Additionally, they refine a previous technique for record layout randomization by introducing a static analysis technique for determining the randomizability of a record. Their design has three distinct but related parts: Record Field Order Randomization (RFOR), RFOR suitability analysis, and Subroutine Argument Order Randomization (SAOR). RFOR occurs at compile-time, during which the field order of the record definition is randomized. SAOR also occurs at compile-time, during which the argument order for each definition of, type of, and call to a given subroutine are randomized.

In [Hom+13] Homescu et al. use profile-guided optimization to reduce the performance overhead of software diversity against code-reuse attacks. Their primary insight is to diversify cold code, but restrict diversification efforts in hot code (code where a program spends most of its execution time). More specifically, they focus on NOP insertion, to randomize the code layout of a program. At each program instruction, they randomly decide to prepend a NOP or not. In case a NOP is inserted, it is chosen at random from a list of NOP candidates. Furthermore, they implement a form of automated software diversification [Hom+15] as a means to thwart code reuse attacks, namely ROP and JOP. They implement compiler-based diversity by extending the LLVM compiler infrastructure, thus enabling automated diversification for all languages supported by the LLVM front-end. They use two main transformations in their design. First, they insert a series of NOP instructions, carefully selected to preserve the processor state at all times and to minimize the likelihood of creating new gadgets. Second, they randomly rearrange the order of instructions (instruction scheduling).

Pomonis et al., in [Pom+17], present a kernel hardening solution that diversifies the kernel's code and prevents any memory read accesses to it. They achieve this by applying code instrumentation to prevent memory reads from code sections, as well as by coupling execute-only memory with extensive code diversification (function and basic block reordering) and return address protection (XOR-based encryption or decoy return addresses). Extending the isolation mechanism of kRX on x86-64 systems, they also propose kSplitStack [Pom20] which leverages a multi-stack scheme where functions use an unprotected stack for their local variables, but switch to a protected one when pushing or popping return addresses, protecting this way the secrecy and integrity of control data.

In [Jel+21], the authors present Mardu, an on-demand system-wide runtime re-randomization technique capable of scalable protection of application as well as shared library code. They also achieve code sharing with diversification by implementing reactive and scalable, rather than continuous or one-time diversification. Mardu relies on an event trigger design and acts on permissions violations of Intel's Memory Protection Keys (MPK). It leverages immutable trampolines which, while not re-randomized, they are protected from read access and decouple function entry points from function bodies, obstructing attackers from inferring and obtaining ROP gadgets.

2.2.2 Control

Mechanisms that control the flow of execution of a program compose this category, which is further divided into two subcategories, namely (i) access and (ii) behavior, based on their implementation.

Access

In [Pro03], Provos developed `systrace(8)`, a system that supports fine-grained policy generation. It guards the calls to the operating system at the lowest level, enforcing policies that restrict the actions an attacker can take to compromise a system. However, a high-level intent would be lost in a multitude of low-level calls. For example, when the `mktemp(3)` function generates a unique temporary file, it first checks if a file name already exists. If not, then it creates and opens the file with the specific name. During this sequence of events, several system calls (`stat(2)`, `open(2)`, `mkdir(2)`, etc) are being made. To prevent arbitrary usage of these functions (e.g., during a race between testing whether the name exists and opening the file, when an attacker can create a symbolic link to an inaccessible file and have access to it from a privileged program), the system calls would need to be examined and policies enforced under a finer-grained framework, like `systrace(8)`). However, this kind of framework would need to check these calls that result in a number of lower-level calls to the operating system. Since only the high-level calls are of importance in this case, the examination of underlying calls would be not only unnecessary, but undesired too. The fine-grain control offered by the framework, while checking calls required by system or user

level libraries when implementing complex operations, is overly verbose. Additionally, it may leave a library in an inconsistent state if the sequence of these calls is interrupted in the middle of execution by a misconfiguration [KP06]. Furthermore, for applications that use high-level abstractions away from low-level system calls, there may be difficulties generating precise policies. Later research [Wat07] showed that concurrency vulnerabilities were discovered that gave an attacker the ability to construct a race between the engine and a malicious processes to bypass protections. More specifically, in a multiprocessor environment, the arguments of a system call were stored by a process in shared memory. After `systrace(8)` performed the check and permitted the call, another malicious process had a time window to replace the cleared arguments in shared memory, effectively negating the presence of `systrace(8)` and evading its restrictions. In a uniprocessor environment, this could be achieved by forcing a page fault or in-kernel blocking, so the kernel would yield to the attacking user process.

Kim et al. in Access Controls For Libraries and Modules (SecModule) [KP06], force user-level code to perform library calls only via a library policy enforcement engine providing mandatory policy checks over not just system calls, as in the case of `systrace(8)`, but calls to user-level libraries as well. The access rights in question would be whether a process (which may be malicious) is allowed to execute some function held securely in a library module. This framework retrofits functions in order to be included in a secure "enclosure" (SecModule). The kernel has a list of all the SecModules and when a process asks for access to a secured function, the kernel verifies that the requested SecModule is registered and that the process is valid with respect to its policy. Then, it allows that and only that process to use only the specific function. This means that access to a specific function or procedure is controlled by the kernel. While this is particularly suited to SecModule-enabled applications, the overhead of two context switches per function invocation (once to transfer control to the kernel and – when it reaches a decision - once more to transfer control back to the caller), makes the technique quite expensive for more general use. One of the issues identified by the authors of the SecModule paper is the difficulty in encapsulating library modules. This manual process is error prone and extremely labor intensive, since most of the applications compiled within the framework required patching. Another issue is the inability to evaluate call arguments. Although they are contained in a known structure pointed to by a stack pointer, their examination requires lots of casting in the C++ functions, which in turn needs additional information for these functions held in the module.

Abadi et al. propose CFI [Aba+05] which enforces the execution of a program to adhere to a CFG, which is statically computed at compile time. If the flow of execution does not follow the predetermined CFG, an attack is detected. This approach, however, suffers from two main disadvantages. First, the implementation is coarse-grained. Computing a complete and accurate CFG is difficult since there are many indirect control flow transfers (jumps, returns, etc.) or libraries dynamically linked at run-time. Furthermore, the

interception and checking of all the control transfers incur substantial performance overhead.

In [Kay+12] Kayaalp et al. propose Branch Regulation (BR), a hardware-supported protection mechanism against CRAs that addresses all limitations of software CFI, including potential vulnerabilities due to the presence of unintended branch instructions in certain architectures. BR enforces simple rules in hardware to limit control flow transfers to an address within the same function, or to an entry point to a new function, or to a return address generated by a legitimate prior call, disallowing arbitrary transfers from one function into the middle of another. However, it annotates the binary, resulting in increased code size. In addition, the implementation of CFI is coarse-grained allowing the program flow to be transferred to any function entry point or any point within the current function. Although the mechanism makes use of a shadow stack to keep track of the return addresses, the shadow stack itself is not secure since it resides in mapped memory. Moreover, when an indirect jump is performed (e.g `longjmp()`) it may transfer the control flow in the middle of a function, which is disallowed from this scheme, thus throwing a false-positive exception.

Das et al. [DZL16] in their work present an approach to enforce fine-grained CFI at a basic block level, named Basic Block CFI (BB-CFI), which aims to defend against stack smashing and code reuse attacks. The key idea is to verify the target address (TA) of control flow instructions (CFINs), which may be modified by the adversary. BB-CFI contains two stages: 1) offline profiling of the program – to extract the control flow information and 2) runtime control flow checking – to verify the TA of CFINs using the extracted information. Additionally, they propose an architectural design of control flow checker (CFC), which monitors the program execution during runtime to enforce BB-CFI, offering an implementation of it on an FPGA.

In [KRK16], Kanuparthi et al. propose Dynamic Sequence Checker (DSC), a framework to verify the validity of control flow between basic blocks in the program, which works in tandem with a dynamic integrity checker to provide control flow integrity. Unique codes are assigned to every basic block in the program at compile time in such a way that the Hamming distance between two legally connected basic blocks is a known constant. At runtime, the Hamming distance between the codes assigned to the source and destination basic blocks is calculated and compared against the known constant, to verify the control flow. Execution is aborted if the Hamming distance comparison does not match.

Niu and Tan [NT14a] present Modular Control-Flow Integrity (MCFI), a CFI technique that supports separate compilation. In MCFI, an application is divided into multiple modules. Each module contains code, data and other information that help its linking with other modules and the generation of the module's CFG. Code of a module is instrumented separately for CFI. When modules are linked either statically or dynamically, their auxiliary information is combined and used to generate a new CFG, which is the new control-flow policy for the combined module after linking. The new policy may allow an indirect branch to target more destinations. The CFG is

represented in a run-time data structure and the reads and updates of it are wrapped in transactions to ensure thread safety. The authors leverage MCFI in order to build RockJIT [NT14b]. RockJIT secures Just-In-Time (JIT) compilers through CFI by enforcing fine-grained CFI on the JIT compiler, while dynamically updating the control-flow policy when new code is generated on-the-fly and coarse-grained CFI on the JITed code, resulting in improved security. Based on MCFI and RockJIT, they also present Per-Input Control-Flow Integrity (PICFI or πCFI) [NT15] that can enforce a CFG computed for each concrete input (which may include unnecessary edges in the CFG). πCFI starts executing a program with an empty CFG and lets the program itself add edges to the enforced CFG, if such edges are required for the concrete input. To prevent attackers from arbitrarily adding edges, πCFI uses a statically computed all-input CFG to constrain what edges can be added at runtime. During execution, πCFI dynamically activates target addresses lazily before the addresses are needed by later execution.

In [GEN15], Gionta et al. present a system - HideM - for protecting against memory disclosures in modern commodity systems. It uses the split-TLB architecture, to enable fine-grained execute-and-read permissions on memory. HideM uses code reading policies to divide read data from executable data (e.g., machine code) on executable pages. Shadow memory pages are created containing only the required readable or executable data. The OS kernel configures the hardware split-TLB to hide executable data from userspace read access. As a result, HideM can apply code reading policies and enforce fine-grained permissions to commercial off-the-shelf (COTS) binaries.

Evans et al. in [Eva+15], present an attack to show that, for architectures that do not support segmentation in which Code Pointer Integrity (CPI) relies on information hiding, CPI's safe region can be leaked and then maliciously modified by using data pointer overwrites. CPI protects access to code pointers by storing them in a safe region that is protected by instruction level isolation. On x86-32, this isolation is enforced by hardware; on x86-64 and ARM, isolation is enforced by information hiding. They focus on the latter architectures and show that the use of information hiding to protect the safe region is problematic and can be used to violate the security of CPI. Specifically, they show how a data pointer overwrite attack can be used to launch a timing side-channel attack that discloses the location of the safe region on x86-64.

Backes et al. [Bac+14] propose an approach to thwart the root cause of memory disclosure exploits, by preventing the inadvertent reading of code while the code itself can still be executed. They introduce a new primitive called Execute-no-Read (XnR) which ensures that code can still be executed by the processor, but at the same time it cannot be read as data. In this way they prevent JIT-ROP attacks, since they require a disclosure vulnerability that enables an adversary to read arbitrary memory locations, which allows searching for gadgets. Forbidding code from being read and hence disassembled, prohibits an attacker from constructing a gadget chain on-the-fly. Consequently, XnR requires OS/MMU modifications, because of the way modern architectures (x86, ARM) are designed.

In [Zha+13] Zhang et al. discuss several representative defense mechanisms against function pointer exploitation. Based on them, they propose Function Pointer Gate (FPGate), a method that rewrites x86 binary executables and implements a method to overcome compatibility issues with the existing development process. FPGate utilizes relocation tables already required by ASLR in modern x86 binary executables, to disassemble binaries and identify all indirect control transfer instructions and all their valid targets. Furthermore, it encodes each valid function pointer into a pointer to a new trampoline memory section, thus enabling modules hardened by FPGate to inter-operate seamlessly with unhardened ones.

In [Lin+21], Lin et al. propose an address-based CRA mitigation technique for shared objects at the binary-level. They set several principles that must be followed by the execution of indirect branch instructions. More specifically, they reconstruct function boundaries at the program's dynamic-linking stage by combining shared object's dynamic symbols with binary-level instruction analysis. They also leverage static instrumentation to hook vulnerable indirect branch instructions to their proposed target address computation and validation routine. This results, at runtime, in protection against CRAs based on the computed target address.

In [BR21], Bauer and Rossow present a compiler-assisted library isolation system (CALI) that shields a program from a given library, using shared memory to allow secure interactions between a program and its libraries. They compartmentalize libraries into their own process and use a Program Dependence Graph (PDG) to observe and propagate data flows crossing the security contexts, in order to preserve the functionality of the interactions between program and library. They then place the according memory regions in shared memory, and isolate the remaining memory in the application and library processes, respectively.

Behavior

The DisARM defense technique [Hab+15b] protects against both code-injection and code-reuse based buffer overflow attacks by breaking the ability of attackers to manipulate the return address of a function. DisARM uses a fine-grained analysis of the binary to find all critical interactions that manipulate the hardware Program Counter (PC) and verifies any change to the PC before it is applied. For each such critical instruction, a verification block is inserted immediately before the instruction in order to evaluate whether the target address is valid with respect to the current instruction the program is executing.

Kanuparthi et al. [Kan+12] propose a hardware-based dynamic integrity checking approach that does not stall the processor pipeline. It permits the instructions to commit before the integrity check is complete, and allows them to make changes to the register file, but not the data cache. The changes made by the instructions are held in the store buffer or in a shadow register file until the check is complete. Then, the values are accordingly written to

the L1 data cache or the original register file. The system is rolled back to a known state, if the checker deems the instructions as modified.

Kayaalp et al. [Kay+15] examine a signature-based detection of code CRAs, where the attack is detected by observing the behavior of programs and detecting the gadget execution patterns. They demonstrate a new attack that renders previously proposed signature-based approaches ineffective by introducing delay gadgets. Delay gadgets have a single purpose of obfuscating the execution patterns of the attack without performing any useful computation. They develop a complete working JOP attack that incorporates delay gadgets. Then, they propose and develop the Signature-based CRA Protection (SCRAP) hardware-based architecture for detecting such stealth JOP attacks. SCRAP recognizes the formal grammar that expresses the attack signatures or the patterns of executed instructions that are indicative of a JOP attack, which are significantly different from those of the regular programs as they execute frequent indirect jump (or call) instructions to jump from gadget to gadget.

2.2.3 Monitoring

This category consists of techniques that observe the flow of execution of a program, stepping in to take action whenever it is deemed necessary. The monitoring can be inline or parallel to the execution.

Inline

Tian et al. [Tia+14] propose PHUKO, an on-the-fly buffer overflow prevention system which leverages virtualization technology. This system offers the monitored program a fully transparent environment and easy deployment without restarting the program. PHUKO combines static analysis and online patching provided by the hypervisor to instrument buffer accesses in the running program. Specifically, it first uses static binary analysis to identify the interesting instructions that are related to buffer overflows and then it replaces them with trap instructions by which the execution of a program will be trapped to the hypervisor. Then, when the monitored program executes these replaced instructions, the built-in bounds checking mechanism will dynamically take effect to ensure that the buffer access is limited within the scope of the allowed memory area.

Crane et al. [Cra+13] identify the technique of booby trapping software. They define booby traps as code providing active defense that is only triggered by an attack. These booby traps do not implement program functionality and do not influence its operation - in fact, the program does not know about its own booby traps and under normal operation cannot trigger them. They propose to automatically insert booby traps into the original program code during compilation or program loading. Whenever an attack triggers one of the booby traps within the program, the trap instantly knows that an attack is underway and is in a position to react to the threat.

Chen et al. [Che+13] propose a kernel-based security testing tool, named ARMORY, for software engineers to detect Program Buffer Overflow Defects (PBODs) automatically when applying testing, without increasing the testing workload. Whenever a developer provides an input string to a program to test its functionality, ARMORY automatically forks a child process, called PBOD test process, and utilizes it to test whether the code used to handle the input string has any PBOD. The parent and the child processes work independently; thus, they do not influence each other. The original process only handles the original input string to test the functionality.

Zhang et al. [Zha+21] present Punchcard, a system that isolates memory objects by placing red-zones between them to ensure spatial memory safety, and to prevent the reuse of deallocated objects by replacing them with red-zones to ensure temporal memory safety. They implement it by extending the page permission checker inside the MMU, in order to store the red-zone metadata within the physical frame number of an address. Additionally, they perform memory access validation and upon an access violation, the Punchcard checker raises a privilege exception which is handled by a custom OS handler.

Parallel

Liu et al. [Liu+14] present a hardware framework for providing detection and prevention of code injection attacks using a lockstep diversified shadow execution. Their goal is to enrich the diversity of software execution with the facilitation from programmable hardware decoder and novel CPU support of a tightly coupled shadow thread technique. Specifically, given a piece of (legacy) binary code, firstly diversified binary versions are generated using an offline binary rewriter and programmable hardware binary translator at runtime. Two diversified binary code images are launched as dual simultaneous threads in the hardware layer with one as the primary thread and the other one as shadow thread. Instructions from the shadow thread are not executed but just compared. The extended CPU is able to decode instructions from both threads, and dispatch them to the next stage pipeline for a lockstep comparison. Any mismatch of the decoded instructions from the two threads caused by remotely injected binary code, will be detected and flagged by the hardware as an intrusion.

Volckaert et al. [VCS16] present Disjoint Code Layouts (DCL), a technique that complements multi-variant execution and DEP protection to immunize programs against control flow hijacking exploits such as ROP and return-to-libc attacks. This technique relies on the execution and replication of multiple runtime variants of the same application under the control of a monitor, with the guarantee that no code segments in the variants' address spaces overlap. Lacking overlapping code segments, no code gadgets co-exist in the different variants to be executed during ROP attacks. Hence no ROP attack can alter the behavior of all variants in the same way. By monitoring the I/O of the variants and halting their execution when any divergent I/O operation is requested, the monitor blocks any ROP attack before it can cause harm.

Zeng et al. [ZZL15] propose HeapTherapy, a solution against heap buffer overflows that integrates exploit detection, defense generation and overflow prevention in a single system. During program execution it conducts on-the-fly trace collection and exploit detection and initiates automated diagnosis upon detection to generate defenses in real-time. It handles both over-write and over-read attacks. It employs techniques to identify vulnerable heap buffers based on the intrinsic characteristics of an exploit, as opposed to filtering out malicious inputs based on signatures.

2.2.4 Hardware approaches

SMEP/SMAP

Recent CPU architectures introduce hardware features that help prevent arbitrary code execution from kernel space, when the kernel tries to access unintended user-space memory. Supervisor Memory Execute Protection (SMEP) [Fis11] and Supervisor Memory Access Protection (SMAP) [Mul15] leverage the User/Supervisor (U/S) bit in page table entries that designates where a page belongs to (user-space application or OS kernel). SMEP prevents supervisor mode from unintentionally executing user-space code. SMAP is designed to complement SMEP and extends the "execute" protection to "read" and "write" attempts. However, Gruss et al. [Gru+16] introduced Prefetch Side-Channel Attacks that allow unprivileged attackers to obtain address information and thus compromise the entire system by defeating SMEP and SMAP and proposed a strong kernel isolation to protect commodity systems. Additionally, Google's Project Zero released an exploit [Kon17] using a vulnerable kernel function to disable SMEP/SMAP protection, which was later rectified [Lar19].

Trusted Execution Environment

A Trusted Execution Environment (TEE) [All18] is another hardware feature that offers a secure, integrity-protected processing environment, consisting of memory and storage capabilities [Aso+14]. It establishes an isolated execution environment that runs parallel to a standard OS and it protects sensitive code and data from privileged attacks without compromising the native OS. It prevents unauthorized access or modification of executing code and data while they are in use, so that the applications running the code can have high levels of trust in the TEE, because they can ignore threats from the rest of the system.

Major hardware vendors have already included this feature into their products. Intel's Software Guard Extensions (SGX) [Ana+13; McK+13; Hoe15; McK+16] helps encrypt a portion of memory. This portion - *enclave* - is used by the OS/applications to define private regions of code and data that cannot be accessed by any (potentially running at a higher privilege level) process outside the enclave, thus preserving the confidentiality and integrity of sensitive code and data. However, several attacks have been developed

that brake the security of SGX. In [Sch+17], Schwarz et al. were able to extract a full RSA private key by performing a cache side-channel attack on a co-located SGX enclave. Later on, countermeasures were released against this attack [Gru+17; Bra+17]. Additionally, the Spectre attack [Koc+19] was adapted to target SGX enclaves [O'K+15]. Similarly, the Foreshadow attack exploits speculative execution (e.g., Spectre) in order to read the contents of SGX-protected memory [VB+18]. Moreover, it has been proven that a ROP attack can be constructed and launched all from within an enclave [Lee+17; SWG19]. However, a defense against this attack vector was later presented in [Wei+19]. More recently, two more attacks have been introduced: (a) Plundervolt [Mur+20] that can break cryptography (RSA and AES) and inject controlled memory-safety bugs into secure code, so as to redirect enclave secrets to be written to untrusted memory outside the enclave and (b) SmashEx [Cui+21] which demonstrates that, in the absence of safe atomic execution, asynchronous exception handling in SGX enclaves is prone to reentrancy vulnerabilities that can be exploited to cause arbitrary disclosure of enclave private memory and ROP attacks in the enclave. In light of these several security vulnerabilities, SGX has been shown to be insecure, leading Intel to discontinue the feature in its most recent processor families [Int21; Int22].

Another major hardware vendor - ARM - specializing in mobile environments and embedded devices has developed its own TEE. ARM's TrustZone [ARM09] has been employed in industrial control systems [Fit+15], servers [Hua+17], and low-end devices [Aso+18]. However, similarly to SGX, over the years there have been several security vulnerabilities uncovered and disclosed [Cer+20], in this case as well. Recently, the CLKSCREW fault attack [TSS17] was developed, which overclocks the CPU to generate hardware faults in order a malicious kernel driver extract secret cryptographic keys from TrustZone, and escalate its privileges by loading self-signed code into TrustZone. Even more recently, VoltJockey [Qiu+19] was released, a software-controlled hardware fault-based attack that manipulates the voltages of the CPU in order to reveal an AES encryption key from TrustZone and breach the RSA-based TrustZone authentication. Lately, in [SBYZ21], the authors present a Direct Memory Access (DMA) attack that allows an attacker to execute arbitrary code in the secure world or read arbitrary data from the secure world, and exploit a hardware vulnerability that compromises TrustZone, in order to replace trusted applications with malicious ones.

2.2.5 Shadow stacks

An extra layer that is employed in conjunction with some security mechanism, in order to further strengthen execution, is a shadow stack [CH01; Ven00]. The original idea behind shadow stacks is that when a function is called, the return address of that call is stored in a separate protected memory region - the shadow stack - where an attacker cannot have access. When the function returns, the saved return address is either compared against the

program's return address on the main stack, or it is placed directly on the main stack overwriting the return address.

There are two categories of shadow stacks, based on their design: (a) parallel [DMW15] and (b) compact [BZP18]. A parallel shadow stack is a copy of the main stack and its position is determined based on the position of the main stack. This design allows for quick mapping of return addresses, however it requires twice the memory size. Compact shadow stacks utilize a separate pointer to hold the position of the return address in the shadow stack (e.g., in a register), while their position is irrelevant to the main stack's. In this case the overhead is increased due to the extra pointer, however memory usage is not as high as in the case of a parallel stack because the return addresses are stored one after the other.

Furthermore, shadow stacks can be (a) hardware-assisted [Fra+18; MRD18] which leverage hardware features (e.g., Intel Control Enforcement Technology (CET) [Cor19]) to provide shadow stack support, or (b) software-only implementations that ensure the shadow stack's integrity [Kuz+14; Lu+15; Wah+93b].

2.3 Summary

Over the past several years, significant developments have been made in research concerning attacks and defenses related to software systems. In Section 2.1, we present the timeline that led to the introduction of CRAs. Additionally, in Section 2.2, we present an overview of the latest efforts to develop the equivalent techniques (that mainly relate to the work of the thesis) to counter CRAs. We classify them in three major categories: randomization, execution control and execution monitoring. Within each group, we try to cover the most relevant efforts to our proposed solutions. Lastly, in Sections 2.2.4 and 2.2.5, we list two complimentary categories that are used to secure a software system - hardware approaches and shadow stacks.

3 Design Considerations

This Chapter deals with the design of our framework. In Section 3.1, we present the basic idea of how we conceive memory segmentation and execution flow redirection. Sections 3.2 and 3.3 detail the assumptions under which we operate with regards to the threats the underlying system may face and the security measures already in place to thwart them, respectively. In addition, in Section 3.4 we present the hardware and software setup which is used to support the development and testing of the security layer. Finally, in Section 3.5 we offer some insight on the use-cases we employ throughout the development of the framework, in order prove its applicability and efficiency.

3.1 Architectural Approach

As it stands, when a program is running, the execution flow can move anywhere within the process's memory area (e.g., libraries, main executable, etc.). Originally, the application and library code share their stack and heap spaces (Figure 3.1), which provides a breeding ground for interfering with the execution of library code. Under specific circumstances and using specially-crafted attack vectors (e.g., ROP, JOP, etc.), an adversary can mislead the CPU to execute code that is already present in the process's memory area, although its execution is not intended by the application designer. For example, when executing a ROP attack, the adversary chains together different sets of instruction sequences ending in *ret* (Figure 3.2). These code snippets - *gadgets* - have been proven to be frequent in libraries and Turing-complete [Sha07].

Under our approach, all calls are intercepted at a double level. On one hand, at the user level, a custom library wrapper intervenes before a library call reaches its destination (Figure 3.3) (see Section 4). There, we have a

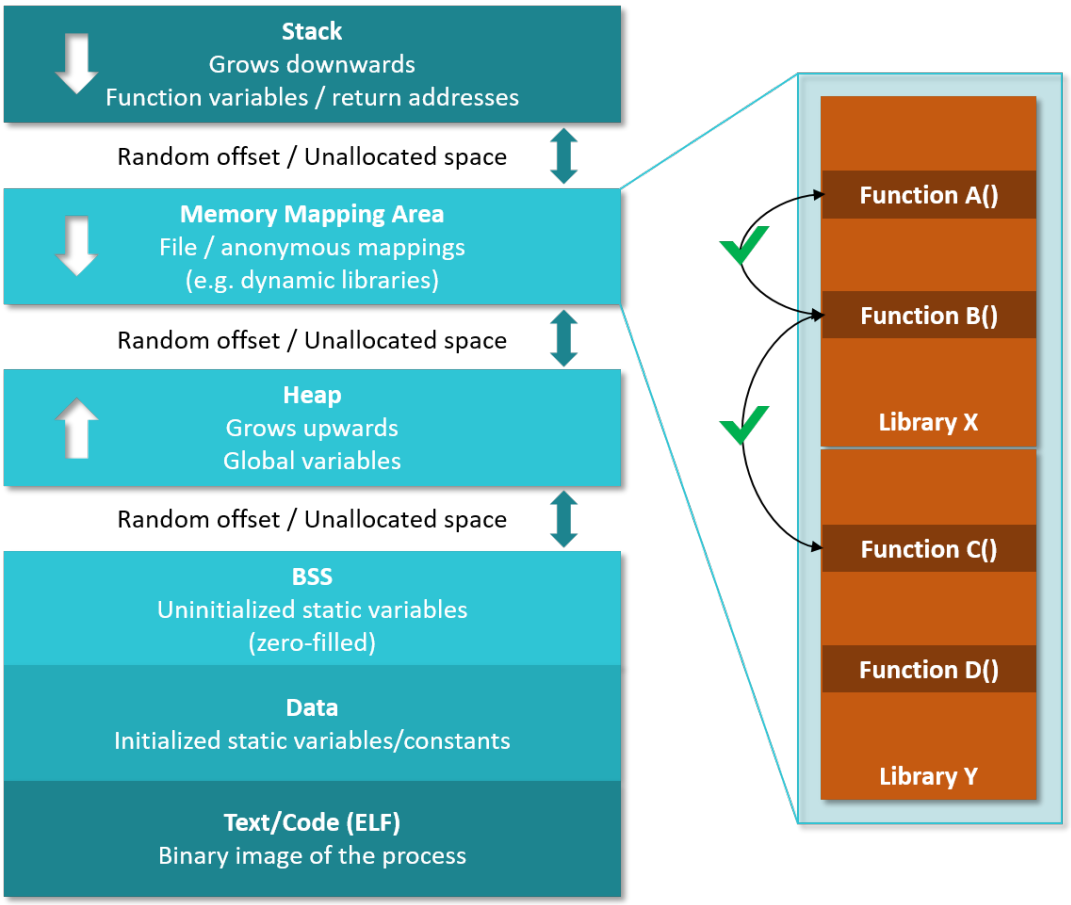


FIGURE 3.1: Current memory layout of a process

chance to analyze the arguments of the call. Moreover, we can apply security policies before allowing the call to move ahead. By doing this, we can detect any deviations from a predefined behavioral profile of the application under test. At this level, however, the wrapper can be bypassed since it resides in user land. Both variants of the default randomization technique (32/64-bit ASLR) were bypassed several years ago [Sha+04; MGR14; EPAG16], although stronger protections have been introduced [MGRR19]. Consequently, a tech-savvy attacker can find out the address of the original library and call the destination function directly, without going through the wrapper. Nevertheless, we point out that this level of interception is primarily used for educational/training purposes. A user (system administrator, security analyst/engineer, student, etc.) can leverage this mechanism in order to gain insight on how the inner workings of a library are carried out, in both cases (i) under normal execution and (ii) when under attack. In a suspicious event, they can then use this knowledge to infer if it is malicious or not (e.g., when performing forensic analysis). Moreover, by redirecting the execution flow through our wrapper in an isolated environment (in order to ensure nominal conditions), our mechanism can operate in *learning mode*. This way, crucial information about how the library should behave in normal circumstances can be extracted. This information can, then, help create more

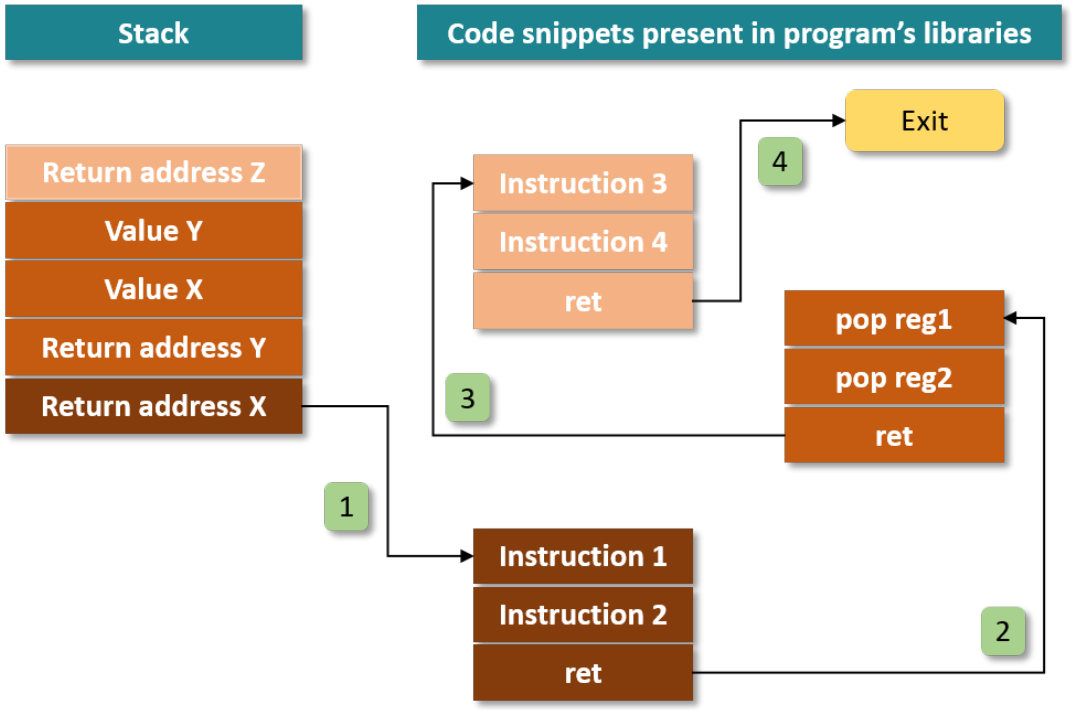


FIGURE 3.2: ROP sequence

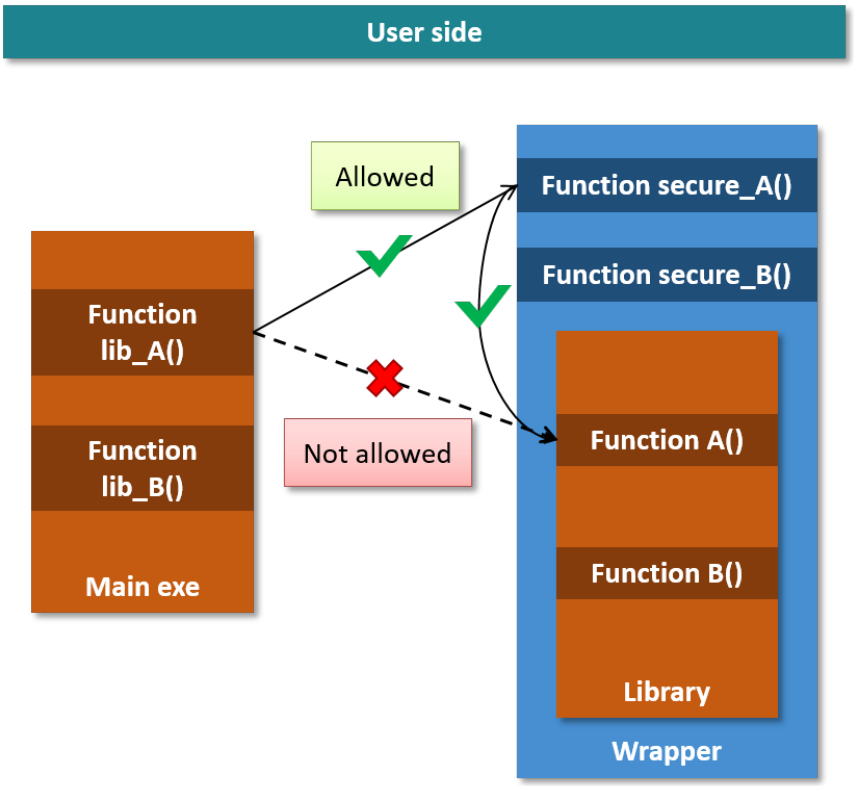


FIGURE 3.3: Library wrappers

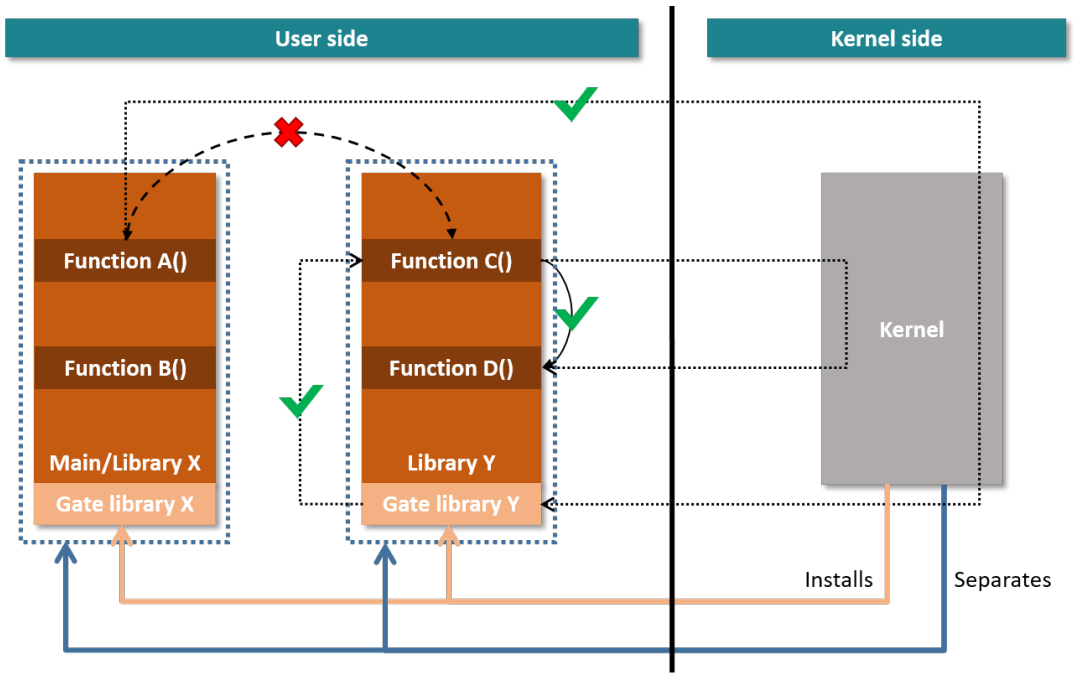


FIGURE 3.4: Memory segmentation and access control

comprehensive and complete security policies.

On the other hand, the kernel intercepts and redirects all calls through a custom library - a *gate*. Currently, inter-library function calls are performed directly from shared library to shared library (Figure 3.1). In our approach, we map the aforementioned *gate* in the process's memory area, but before doing so, first we separate all libraries as well as the main executable in order to form standalone partitions that can communicate with each other (Figure 3.4). Each partition is equipped with its own *gate*. Communication within a specific partition is performed normally, without any restrictions ($C() \rightarrow D()$). However, calls from one partition to another are not permitted directly, but only after being passed through the *gate* by the kernel ($A() \rightarrow C()$). At this level, in contrast to the user-side interception, the kernel takes over the execution and redirects all calls through the respective policy enforcement engine (essentially the associated *gate*), leaving no way for an attacker to bypass it. A more detailed and technical view for this follows, in Section 5.

3.2 Adversary Model

Throughout the thesis, we assume a powerful yet reasonable threat model that is consistent with the equivalent models used in the state-of-the-art. The work described here is concerned with control-flow hijacking attacks, the purpose of which is to divert the control flow to a point that would not otherwise be reachable in that same context, had the execution not been maliciously altered. The attacker has full control over data and heap/stack

memory regions, which is in-line with published CRAs that exploit memory bugs in order to mislead the CPU to unintentionally execute arbitrary code. Additionally, the attacker cannot modify the code segment, because the corresponding pages are marked executable and not writable. More specifically, we consider an adversary model which is composed of three classes of threats, against which our approach can defend.

The bugs/vulnerabilities that can be found in an application and exploited comprise the first class. These can be either (a) already discovered and known without, however, a security patch being available or having being applied by the system administrator or (b) zero-day/undiscovered, without anyone knowing about their presence. Coding errors that lead to memory corruption vulnerabilities remain one of the most common and dangerous weaknesses in modern software [Tea22; Vij21]. Very common among such vulnerabilities are: (i) buffer overflow and (ii) dangling pointer (use-after-free).

Buffer overflows are one of the most widespread memory corruption classes and are usually caused by a coding mistake/oversight. They happen when the length of user-supplied data is not properly checked by a program. When the program attempts to store more data in a buffer than it can hold, this can result into writing past (overflowing) the buffer. In this case, the buffer is a sequential section of allocated memory space. Writing outside the bounds of a block of allocated memory can corrupt neighboring data, crash the program, or cause the execution of malicious code. These bugs can be exploited by powerful methods that take advantage of the way data is laid out on the heap or stack (heap/stack smashing) or of the unexpected results of arithmetic operations (e.g. integer overflow) among other approaches.

Dangling pointers point to a memory region that has been freed and is no longer valid. They are the cause of the use-after-free exploit, where an object or structure in memory is deallocated (freed) but still used. An object is created and the attacker triggers a "free" operation on it. Then, they create an arbitrary, fake object as closely similar as possible, to the first. Later, when the program tries to use the original object, the fake one is used instead leading to arbitrary code execution.

These vulnerabilities do not lead directly to privilege escalation. In that case, a CRA would not be necessary, as the adversary - having sufficient privileges - could circumvent any defense in place. The attacker, however, may seek to achieve privilege escalation indirectly, by mounting a CRA that exploits the bugs in question.

Secondly, the attacker knows the binary executable of the process and the OS version of the system under attack. Hence, they can precompute potential gadget chains in advance. Based on the attacks and approaches described in Section 2, the adversary is in possession of a technique (e.g. BROP, JIT-ROP, etc.) that can exploit the aforementioned vulnerabilities and can mount an attack that eventually compromises an application.

Lastly, the third threat class is the behavior of a program. At run-time, an application can present abnormal behavior that opposes what is nominally expected. If we see - for example - that a program is trying to make multiple socket connections to a remote host (despite the fact that during its normal execution it only makes one such connection e.g., to upload/exchange some data), this change in behavior will be considered suspicious and flagged as such.

3.3 Security Assumptions

Modern Linux environments have incorporated several defenses over the years to counter attacks against the system. Here, we make the following assumptions about the underlying system, in accordance with these by-default enabled/widely used approaches:

ASLR is enabled, which randomizes the location of data and code regions every time a process is executed, in order to defend against code reuse exploits.

CFI or a similar variant can be used against code reuse attacks as well, which enforces the execution of a program to adhere to a pre-calculated control flow graph.

W⊕X / NX-bit is enabled, which prevents memory pages from being writable and executable at the same time, defending against code injection attacks.

Stack canaries / `-fstack-protector` **/** `-fsanitize` **flags** are used to protect from buffer overflows.

The hardware is considered not flawed and the OS kernel trusted and secure.

The security policies to be enforced by our mechanism are ideally provided by the application/library designers and are considered to be complete and correct. As mentioned in Section 1.1.2, the policy generation needs to be done in a careful way. The designer/programmer needs to ensure that all code is tested, so it is important to provide a sufficient testing methodology that can cover the software in its entirety. All code regions must be comprehensively tested, in order to create a complete profile of the application behavior. However, this is out of the scope of this thesis, since we assume that we receive such a profile ready to be enforced. That being said, in Section 4.3 we present our approach in producing the application behavior profile, which was used for testing purposes throughout development.

A system administrator can follow this approach, which however will impact the completeness and correctness of the respective policy. Furthermore, the policies cannot be tampered with by an adversary (e.g., they are digitally signed).

A user-space application (APP) has been tested, but not guaranteed to be free from programming errors that might reveal memory corruption / control-flow hijacking vulnerabilities. These can be repeatedly exploited by an attacker, giving them the ability to bypass the default security measures in place. However, the attacker cannot obtain root privileges (i.e., cannot access the kernel or disable / circumvent any defense mechanisms in this way).

Other attack vectors such as side-channel attacks / speculative execution / microarchitectural leaks, although important, are considered out of scope.

3.4 Testbed

Throughout the duration of the design and implementation of the prototype, the following hardware and software setup was used, as can be seen in Figure 3.5. This is reported from the Phoronix Test Suite (PTS) benchmark application [Pts] and is based on Linux kernel version 4.16.7, the latest one when the implementation phase started.

PHORONIX-TEST-SUITE.COM	Phoronix Test Suite 9.0.1
AMD FX-8370 Eight-Core @ 4.00GHz (4 Cores / 8 Threads)	Processor
ASRock 970M Pro3 (P1.60 BIOS)	Motherboard
AMD RD9x0/RX980	Chipset
16384MB	Memory
256GB SAMSUNG MZ7TD256	Disk
Sapphire AMD Radeon HD 6450/7450/8450 / R5 230 OEM 1GB	Graphics
Realtek ALC892	Audio
DELL U2412M	Monitor
Realtek RTL8111/8168/8411	Network
Ubuntu 16.04	OS
4.16.7.CUSTOM (x86_64)	Kernel
Unity 7.4.5	Desktop
X Server 1.19.6	Display Server
modesetting 1.19.6	Display Driver
3.3 Mesa 18.0.5 (LLVM 6.0.0)	OpenGL
GCC 5.4.0 20160609	Compiler
ext4	File-System
1920x1200	Screen Resolution

FIGURE 3.5: System configuration

3.5 Use-Cases

In order to evaluate the applicability and efficiency of our prototype, we used two main test cases during the course of development: (a) a ChaCha20-Poly1305 cipher vulnerability and (b) an NGINX HTTP Server vulnerability. Although these vulnerabilities are several years old and were subsequently addressed after discovery, we use them as prime examples of what our framework can achieve in real-life situations that affect the availability and security of a production system. These two use-cases are in-line with the adversary model presented in Section 3.2. More specifically, they are representative examples of buffer overflow vulnerabilities that lead to interference with the heap or stack. They are adequately generic in order to cover a wide range of similar use-cases. Additionally, they are comprehensive enough to show the efficiency and complete functionality of our approach. We argue that, for these reasons, the two scenarios that we detail next are sufficient and there would be no reason to include more, as the results from our mechanism would be equivalent.

3.5.1 ChaCha20-Poly1305 heap-based buffer overflow

CVE-2016-7054 [CVE16b; CVE16a] is a heap-based buffer overflow vulnerability related to TLS connections using *-CHACHA20-POLY1305 cipher suites. It was discovered on September 2016 and characterized as highly severe. Servers implementing versions 1.1.0a or 1.1.0b of OpenSSL, can crash when using the ChaCha20-Poly1305 cipher suite to decrypt large payloads of application data, making them vulnerable to DoS attacks. It is triggered by an error during the verification of the MAC. If it fails, the buffer on which the decrypted ciphertext is stored, is cleared by zeroing out its content via the *memset* function. However, the pointer to the buffer that is passed to the function points to the end of the buffer instead of the beginning. If the payload to be cleared is large enough, the contents of the heap will be erased, resulting in a crash when OpenSSL frees the buffer.

3.5.2 NGINX stack-based buffer overflow

CVE-2013-2028 [Com13] is a stack-based buffer overflow vulnerability in the NGINX Server application, related to the chunk size of an HTTP request with the header `Transfer-Encoding:chunked`. It was discovered in 2013 and received a high severity score. Specifically, the *ngx_http_parse_chunked* function in *http/ngx_http_parse.c* in NGINX server versions 1.3.9 through 1.4.0 with the default setup, allows remote attackers to cause a crash via a DoS attack or execute arbitrary code when a request with a large chunk size is received, which triggers an integer signedness error and a buffer overflow.

Part II

SOLUTIONS

4

User-Level Execution Monitoring

Access control, in a narrow sense, is the ability of a system to grant or reject access to a protected resource. This way, in the context of software security, the system can keep track of who has access to what code, who can call what function in a library and under which conditions this is possible. These restrictions are imposed by a set of mandatory controls that are enforced by the system in the form of policies. Policies may represent the structure of an organization or the sensitivity of a resource and the clearance of a user trying to access it. A mechanism maps a user's access request to a collection of rules that need to be implemented in order for the system to function in a secure manner.

An access control system can be implemented in many places and at different levels in an infrastructure (e.g., operating system, database management system, etc.) and must be configured in a way that provides the assurance that no permissions will be leaked to an unintended actor, which may give them the ability to circumvent any defenses in place.

In this Chapter, we present a novel mechanism that allows access control policies for library calls to be enforced at the user-code level in order to restrict access to functions held in a protected library, in addition to identifying the complete execution path regarding the functions in question. Furthermore, we offer a mechanism that further divides a library into smaller segments (e.g. based on their functionality), which cooperates seamlessly with our kernel-based approach in Chapter 5. At run-time, the policy system may be used to enforce either an overall policy for the whole library, or more refined policies for each segment. It can coexist with existing defense techniques, boosting the security of the protected system.

The rest of the Chapter is organized as follows: Section 4.1 deals with the access control mechanism. In Section 4.1.1, we provide an overview of our mechanism, its strong points as well as its shortcomings. Then, we explain how we implement it, in Section 4.1.2. In Section 4.1.3, we show that it can

be applied in real-life scenarios and we present a case where we pinpoint the exploitation of a vulnerability. Section 4.2 refers to our efforts to break up a library, libc specifically, into smaller regions. In Section 4.3, we propose an approach to create the security policies that the mechanism can enforce. Additionally, we leverage and extend this approach to compromise an antivirus solution. Finally, we summarize the Chapter in Section 4.4.

4.1 Library-Level Access Control

This approach corresponds to Figure 3.3. It bears much resemblance to `ltrace` [Ces; Ltr], a utility that runs a specified command until it exits. It intercepts the calls made to shared libraries by an application and displays the parameters used and the values returned by the calls. Moreover, it can trace system calls executed by the application. However, because it uses the dynamic library hooking mechanism, it cannot trace statically linked executables/libraries, as well as libraries that are loaded automatically using `dlopen(3)`. `dlopen(3)` gives the programmer the ability to inject symbols in the dynamic library, but these symbols need to be unresolved in the main executable or be exported in its dynamic symbol table. When the linker tries to resolve them, it will find the injected symbols and not the original ones. A statically linked application has neither unresolved symbols nor a dynamic symbol table. Additionally, `ltrace` can only display the parameters used and values returned by the calls. It offers no ability to manipulate them. Parts of this work have been published in [TP17].

4.1.1 Overview

This work revisits earlier work on Access Controls For Libraries and Modules (SecModule) [KP06] that forces user-level code to perform library calls only via a library policy enforcement engine. Our approach automates the process of encapsulating library modules and allows entire libraries to be instrumented, checking the arguments of the calls to functions within a library along the way, before reaching a policy decision. The flow of execution inside the protected library can also be detailed, revealing the sequence of calls to its functions.

Figure 4.1 depicts a high level overview of the steps taken when an untrusted app calls a protected function. In step (1), the application calls a function secured in our custom library (in this case SHA1). In step (2), instead of the intended function, the secure wrapper version of it is executed. Instrumented inside the wrapper, there can be argument and policy evaluation code, which is first run before any other steps are taken (step 3). If the evaluation is successful, the originally intended function is called (step 4) and the execution continues normally.

Due to the fact that we interject our evaluation code between the original call and the intended function, our approach is transparent. It requires no code modifications on the library's code, which makes it suitable for legacy

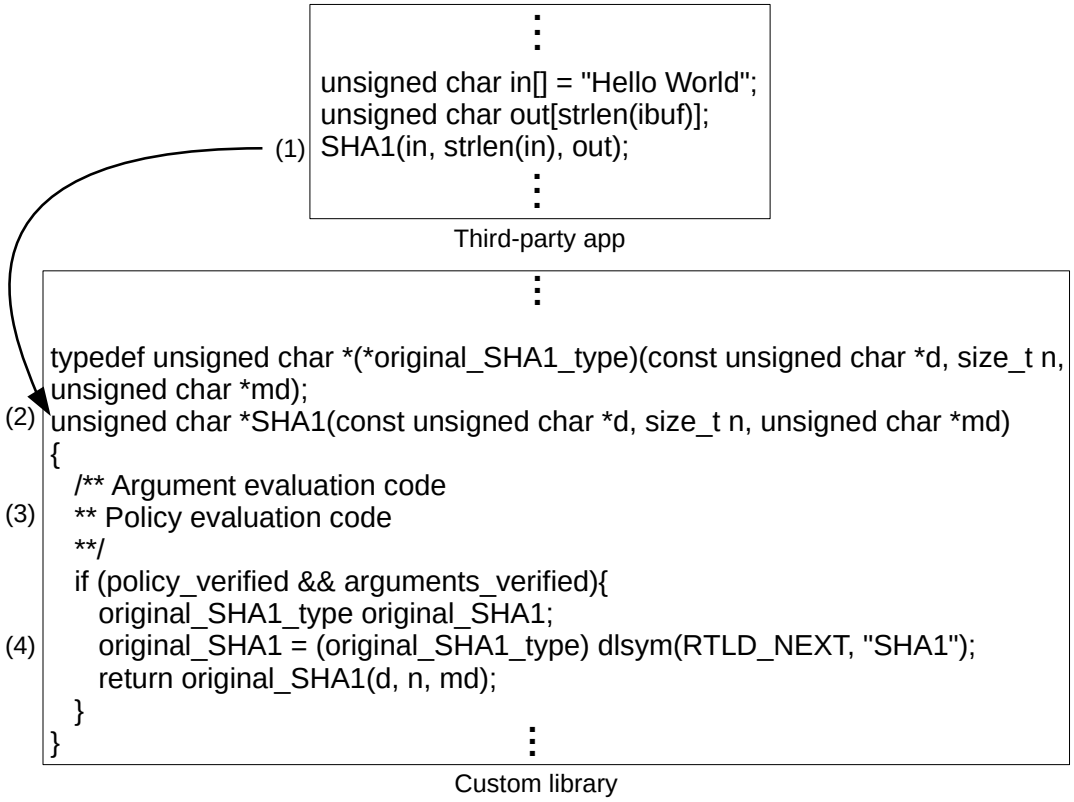


FIGURE 4.1: Overview of the call sequence

applications. Also, it can be used on binary programs, since there is no need to have access to or recompile the source code of the application.

The product of the customization of a library – which is a shared custom library – can be easily adopted by security experts and used in real-life environments, since it only needs to be preloaded before running an application.

No context switch is necessary, using our custom library, since the kernel is not invoked in anyway whatsoever. This makes our technique very efficient. Furthermore, the encapsulation of the library functions is straightforward using just a python script to automate the procedure, requiring only minimal manual intervention. Our past experience and simplicity in producing the code, as well as major support from the community, lead to the decision of using Python as the means to create the shared library.

Under our scheme, the parameters of the intercepted calls can not only be observed, but also manipulated in order to be sanitized if necessary. Unlike `ltrace`, our mechanism relies on `dlsym(3)` and `dlopen(3)` to find the address of a symbol in memory, but because it also relies on dynamic library hooking, it is unsuitable for tracing statically linked applications.

Based on our current approach, the size of the code is increased because extra code needs to be added for every function. Before making the intended call, an extra wrapper is executed in order to decide whether to redirect the flow to the initial call or not.

Additionally, our framework depends on the programming language used to develop the protecting application, since – currently – it can only

protect applications written in C/C++.

Furthermore, if an attacker knows of the presence of the protection mechanism, they might be able to bypass the policy evaluation step and call the intended function directly. Nevertheless, randomization techniques, such as ASLR [PaX01], make direct calls to libraries untenable.

4.1.2 Implementation

Our technique monitors calls to external functions inside a protected library. We investigated two ways of doing this: (a) individual wrappers or (b) one overall wrapper:

- In the first approach, we install separate wrapper functions. Each function in the library that has an interface to the outside world is enclosed in a wrapper. When the wrapper is called, first it executes policy evaluation code to determine if the caller is permitted to call the function and then redirects the flow to the originally intended function or not.

- In the second case, the wrapper stands at the entry point of the library. A policy enforcement engine inside the wrapper monitors the incoming requests and when a call is made to a function, it determines whether that call is warranted (i.e., in accordance to the system policies). It then diverts the flow of execution to the called function.

In both approaches, the policy evaluation code can examine the arguments of the call to ensure that they comply with the security policy associated with the running program.

For our prototype, we decided to follow the first path, due to the simplicity of the implementation. As an example, we create a wrapper for the OpenSSL library. The header files of the library can be included in any C/C++ program by the developers and contain all the functions that they can call. First, we extract from the header files all the relative functions and their signatures. The extraction is done using a custom Python script that identifies each function that is within the scope of our work and analyzes its arguments. This way we are able to manipulate each of the arguments in any way necessary. Before calling the originally intended function we add code that verifies that the module, indeed, captured the call and that we operate from within the custom library. After implementing the security features (e.g., argument examination, policy enforcement, etc.) and if the continuation of the execution is permitted, the flow progresses to the original path. The result is a C file that is compiled into a shared library which is preloaded when running a program.

Automatic generation of policy (learning phase) can also be supported, while at run-time the policy system can be used for policy enforcement and/or for ensuring that the program behaves in the same manner as in the learning phase. During this phase, as many as possible execution paths need to be discovered that correspond to actions taken from a benign application,

aiming to implement a CFI-like [Aba+05] scheme that uses library calls to extract execution paths, instead of intercepting or instrumenting or emulating the lower-level control flow instructions.

However, since this approach is essentially another shared library loaded into the program at run-time at the user space, an advanced attacker could be able to bypass it and circumvent the policy enforcement step. By-default enabled randomization techniques, as well as hardening approaches at the kernel side (detailed in Chapter 5) work in parallel to this user-level approach, making it much more difficult for an adversary to bypass the security policy evaluation.

4.1.3 Use-Case Study

There are several real-world scenarios that this mechanism can find application to. In the context of Digital Rights Management (DRM), it can provide access control in order to restrict usage of a piece of software. The owner of the software retains the right to distribute it on their own conditions (e.g., after getting some form of payment or even just recognition for their efforts) or prevent the theft of it.

In the case of a library that requires heavy resources from the host system, the administrator may wish to control access to the rights to invoke the library, in such a way that the system does not hang by over-use or is not affected by a DDoS attack. Access restrictions can be imposed according to certain criteria or security policies enforced by an organization.

The misuse of a critical component in a secure infrastructure can result in unforeseen consequences for the system. Our approach can make sure that only authorized personnel can have access to the secure part. Even in the case of deliberate actions that lead to an attack that jeopardizes the system, our framework can be used as a logging and/or training mechanism. The inner workings of a protected library will be traced, which will follow the flow of execution of functions held within the library. Forensic actions (after the fact) can, then, be taken to analyze in a more detailed view the events that led to the compromise and identify the culprits responsible. Training exercises are also possible, so that the personnel will be able to recognize the events of the attack.

To showcase the actual applicability of our mechanism, in this section, we present a scenario where a vulnerability of an application is exploited to affect the availability of the system. In our use-case, we use a vulnerable version of OpenSSL library, where a buffer overflow is triggered under specific circumstances to launch a DoS attack, in order to crash the application. By using our instrumented library to observe calls to the OpenSSL functions, we can better understand the behavior of the attack and characterize the vulnerability.

As mentioned in Section 3.5, ChaCha20-Poly1305 heap buffer overflow is a vulnerability related to TLS connections using *-CHACHA20-POLY1305 cipher suites. Although it was addressed in versions later than 1.1.0b, we can

use our prototype to examine the chain of events inside the OpenSSL library that result in a crash when the vulnerability is exploited.

When we first start an OpenSSL server instrumented with our shared custom library (e.g., `LD_PRELOAD=/home/user/Desktop/custom_lib.so ./bin/openssl s_server -cipher 'DHE-RSA-CHACHA20-POLY1305' -key cert.key -cert cert.crt -accept 4433 -www -tls1_2 -msg`), an initialization phase takes place, where we can see that memory is allocated for the `s_server` app. Excerpt from our mechanism:

.....
Intercepted call to function CRYPTO_strdup
String parameter: apps/s_server.c
.....

Then, the private key and certificate files are read. Excerpt:

.....
Intercepted call to function BIO_new_file
String parameter 1: cert.key
String parameter 2: r
.....
Intercepted call to function BIO_new_file
String parameter 1: cert.crt
String parameter 2: r
.....

After that, a pointer to every cipher supported by TLS v1.2 is pushed on the cipher stack, if it is not already there. Excerpt:

.....
Intercepted call to function EVP_add_cipher
Intercepted call to function EVP_aes_256_ccm
Intercepted call to function EVP_add_cipher
Intercepted call to function EVP_aes_128_cbc_hmac_sha1
.....

Continuing in a similar manner, a pointer to every message digest supported by TLS v1.2 is pushed on the digest stack, if it is not already there. In addition, aliases are mapped to ciphers/digests. Excerpt:

.....
Intercepted call to function EVP_md5
Intercepted call to function EVP_add_digest
Intercepted call to function OBJ_NAME_add
String parameter 1: ssl3-md5
String parameter 2: MD5
Intercepted call to function EVP_add_digest
Intercepted call to function EVP_sha1
.....
Intercepted call to function OBJ_nid2sn
Intercepted call to function EVP_get_cipherbyname
String parameter: DES-EDE3-CBC
.....

Then, memory is allocated based on the compiled-in ciphers and aliases. Excerpt:

.....
Intercepted call to function CRYPTO_malloc
String parameter: ssl/ssl_ciph.c
Intercepted call to function FIPS_mode
.....

At the end of this initialization process, an "ACCEPT" message is displayed, notifying the user that the server is up and running and awaits incoming connections. Excerpt:

.....
Intercepted call to function BIO_printf
String parameter: ACCEPT
.....

To automate our efforts we used an open-source TLS test suite and fuzzer named tlsfuzzer [Kar15], written in Python, which includes a script to exploit CVE-2016-7054.

When the script is executed, we see a number of calls to `BIO_printf` function which display the messages exchanged between client and server (ClientHello, ServerHello, ServerKeyExchange, etc.). Then, at some point during execution, we see a call to `ERR_put_error` which signals that an error occurred and adds the error code to the thread's error queue. Excerpt:

.....
Intercepted call to function ERR_put_error
String parameter 1: ssl/record/ssl3_record.c
.....

Continuing, the program gets the error's code from the queue via `ERR_peek_error`. Then `ERR_print_errors` is called to print the error string. At this point, memory is freed via calls to functions like `CRYPTO_free`, `BIO_free_all`, `CRYPTO_free_ex_data`, `OPENSSL_cleanse`, `EVP_CIPHER_CTX_free` etc. Under normal circumstances, the server would reset the connection awaiting new incoming messages, but due to the CVE-2016-7054 bug the heap is nullified and the sever crashes, potentially indicating a DoS attack.

During the exploitation of this vulnerability, our library shows all the system calls made from the phase of the initialization of the server, to the handshake between it and the client, to the crash after the attack. This provides a forensic trail to identify the functions executed in the OpenSSL session, in order to pinpoint where the vulnerability is triggered – in this case, when the memory is freed.

4.2 Finer-Grained Segmentation

A basic goal behind the secure framework is to significantly confine an attacker's code base that is available at any given point in time, which results in them having much lower chances of mounting an attack. Nevertheless, there

may be instances where the memory segmentation is insufficient to stop an attack this way (e.g. as in [Goo18; ZB18]). The same goal can be applied to each of the separate memory areas. Thus, the (sub)sections are only able to interact with other (sub)sections, which are specified by the security policy. The main aim of this confinement is to limit the propagation of the attack, in the sense that even if an attacker manages to gain a foothold inside a separate region, they will not be able to execute code not only from other regions indiscriminately, but from within the same region as well.

In a given program, there is a number of libraries loaded at run-time that are essential for the program to run. However, many of the functions included in such a library may not be necessary, depending on what the program does. This results in increased memory consumption, when there may not be any need for it. For example, libc is the standard library for the C programming language which is used extensively for development in Linux environments. Every program written in C uses some version of libc by default. However, libc and more specifically the version that we are dealing with – GNU libc, glibc – is big and contains an array of functions to perform many kinds of procedures. There are instances of programs that take up 1KB of memory, when libc that needs to be loaded takes up 17 MB of memory.

Up to a point, libc is already compartmentalized with respect to its portions that need to be specifically loaded when a program requires their use (e.g., libm, libcrypt, etc.). Nonetheless, based on the source code of glibc [Glic], it contains more than 700 directories, 17.500 files and 4.000.000 Lines of Code (LoC). Consequently, it is apparent that it is a big library that offers ample attacking ground for a determined adversary. The more code resides into the loaded libc, the more code space it takes up during execution and the more possibilities an attacker has to exploit a bug and mount an offensive against the underlying system. With our approach that requires only minimal changes and additions to glibc source, we minimize the code loaded at run-time only to what is absolutely necessary, by loading only specific portions of libc that are mandatory for the program to run correctly and nothing else.

In this section, we present our efforts to compartmentalize libc at a greater granularity, so that only the absolutely necessary functions/portions can be loaded during execution and less memory can be used by the running program. Figure 4.2 shows the idea behind our approach. The complete libc (to which one *gate* corresponds) is broken up into smaller regions along the lines of specific functions (e.g., *dysize*() from *time* directory) or of specific functionality (i.e., distinct directories from the source code - see Table 4.1). One or more regions can, then, be loaded separately if and when an application requires it, without having to load the whole library.

Following the basic principle behind Figure 3.4 which still applies in this case, one *gate* is mapped per region, proving that this approach works seamlessly with what we present in Section 5. Parts of this work have been included in [Tsa21].

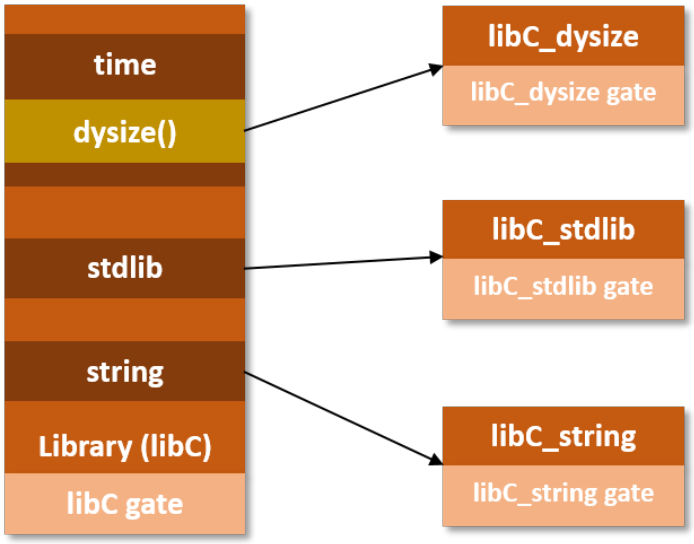


FIGURE 4.2: libc compartmentalized at directory/function granularity

4.2.1 Related Work

In [Har01], the author presents his approach towards building a custom minimal set of the glibc C libraries, using only the necessary objects required by a specific (group of) executable(s). He first determines which objects from libc library will be needed by the application and then builds a custom version of libc that includes only these objects. Then the application is executed after being linked with the minimal custom libc. This approach is similar to our own in the sense that it tries to minimize the memory footprint and size (i.e. attack surface) of libc, however it requires that every application be analyzed in order to determine the necessary objects and a separate version of libc be built for each specific application. This way, the user needs to interact with the tool and have technical knowledge in order to operate it, for each application they want to run with the minimal libc. Our approach is one-off, meaning that the library is built only once and can then be used by all applications automatically. Furthermore, it is totally transparent to the user, since they are not required to build libc themselves, but only link a specific extra library that they want to use at compile/execution time, similarly to other libraries of libc that need this by default (e.g., libm, libcrypt, etc.).

In Chapter 5, we show how to separate the memory of a running process into regions along the lines of loaded shared libraries, one of which is libc. Based on our work, by further compartmentalizing libc (or any other shared library), besides smaller memory footprint, we also manage to strengthen the security of a running application since there are more *gates* – meaning more security checks – that need to be passed successfully in order to execute code from an intended region. Additionally, we decrease the attack surface of a potentially malicious attempt, since libc contains only the bare minimum and all other code that the application requires is loaded in the form of extra

libraries, leading to much smaller size of the loaded libraries.

4.2.2 Implementation Specifics

This Section contains details on two approaches: (a) how to extract a whole directory and (b) how to extract specific functions from the source code of glibc - version 2.3.2. In both cases, an extra dynamic library is created that needs to be explicitly loaded at compile/execution time, if we want to use the specific functions in our program.

catgets directory

In the first case, we extract the whole *catgets* directory from glibc, which deals with some translation aspects. In order to do this and create an extra dynamic library containing the related functions, we need to modify the following files:

1. <glibc_source_code_directory>/Makeconfig
2. <glibc_source_code_directory>/shlib-versions

This can be extended to include other functions from other portions of libc, if needed.

Makeconfig: In this file, we need to remove the *catgets* directory from the list of subdirectories containing the libc source. This way, the subdirectory does not get built into libc.

- In line 1270, we delete *catgets*

shlib-versions: In this file, we need to "tell" the final libc build that there will be an extra shared library which we will be able to use. So, after line 75 (end of file), we add:

```
1 libmartsan_catgets = 1
```

catgets_build_script: Additionally, we need to run a custom script that takes care of some dependencies and compiles all *catgets*-related C files into a shared library, based on the normal-case build process of glibc.

Compile, link and run: Next, we build glibc normally [Glia; Glib]. Then, in order to use a *catgets*-related function in a program, we must load it explicitly, as shown in Figure 4.3. In point (1), we compile and link with the system libc, so the program runs as expected. When, in (2), we compile and link with the custom libc that does not contain the *catgets*-related functions (in this case *catopen*()), when trying to execute we get an error since these functions are missing. After linking in the extra library with `LD_PRELOAD` in (3), the program runs as in the normal case.

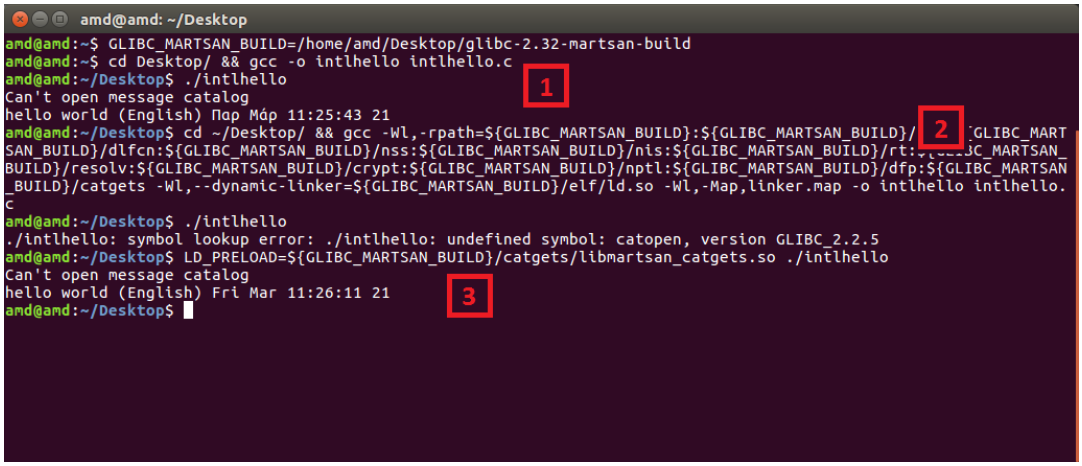


FIGURE 4.3: Compiling and using a separate *catgets*-related function

dysize function

There may be instances that even greater granularity is required, at a function level. In this second case, we extract a single function (*dysize*) from the time directory and create the extra dynamic library containing only this function. Three files need to be modified, in order to successfully do this:

1. <glibc_source_code_directory>/time/Makefile
2. <glibc_source_code_directory>/time/Versions
3. <glibc_source_code_directory>/shlib-versions

Makefile: In this file, we need to remove the *dysize*() function from the list of routines that will be built for the time part of glibc. We also need to state that we want an extra library to be built that contains only the *dysize*() function.

- In line 36, we delete *dysize*
- After line 41, we add:

```
1 extra-libs                = libmartsan_dysize
2 extra-libs-others         = $(extra-libs)
3 libmartsan_dysize-routines = dysize
```

Versions: In this file, after line 82 (end of file), we add:

```
1 libmartsan_dysize { GLIBC_2.0 { dysize; } }
```

```
1 libmartsan_dysize = 1
```

shlib-versions: In this file, after line 75 (end of file), we add:

Compile, link and run: Next, we build glibc normally [Glia; Glib]. Then, in order to use *dysize*() in a program, we must load it explicitly with $-l$ flag, as shown in Figure 4.4.

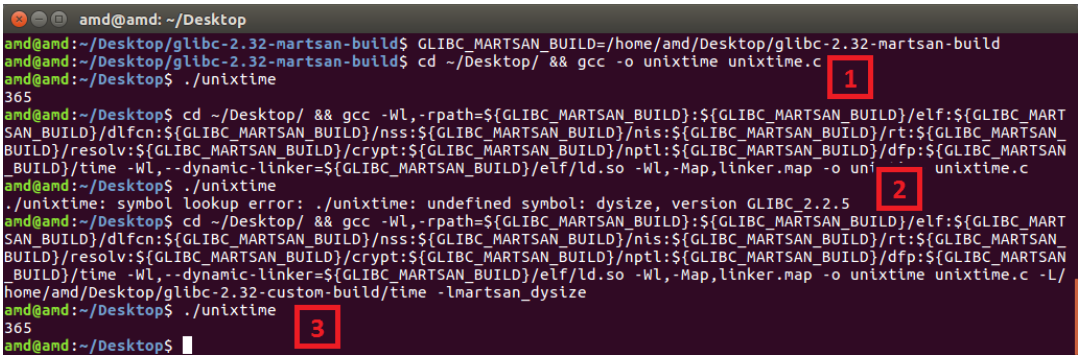


FIGURE 4.4: Compiling and running the separate *dysize* function

In point (1), we compile and link with the system libc, so the program runs as expected. When in (2), we compile and link with the custom libc that does not contain *dysize*(), when trying to execute we get an error since *dysize*() is missing. In point (3), after linking in the extra library, the program runs as in the normal case.

Effort

In each of the previous cases, it is evident that only minor changes are required to make our approach happen:

(a) *catgets*

- Deletion of one word
- Addition of 1 line in total
- Execution of a pre-made script

(b) *dysize*

- Deletion of one word
- Addition of 5 lines in total

In Table 4.1, we provide a list of portions of glibc with their respective final code space required that can be extracted and compiled as shared libraries.

argp	1 MB	intl	2.4 MB	signal	1.9 MB
assert	190 KB	io	5 MB	socket	1 MB
csu	968 KB	libio	15.3 MB	stdio-common	9 MB
ctype	606 KB	malloc	2.6 MB	stdlib	7.1 MB
dirent	2.5 MB	nscd	4.7 MB	string	8.5 MB
gmon	593 KB	posix	8.3 MB	sysvipc	717 KB
grp	1.5 MB	pwd	1.2 MB	termios	653 KB
gshadow	1 MB	resource	566 KB	time	3.3 MB
iconv	3.6 MB	setjmp	191 KB	wcsmbs	6.7 MB
inet	8.2 MB	shadow	1.2 MB	wctype	927 KB

TABLE 4.1: glibc portions and their size

4.3 Policy Generation

In order to implement access control at the library level, we rely on security policies that dictate to the monitoring mechanism what transitions are permitted at run-time. These policies can originate from various sources, e.g.:

- the library designer/developer,
- the monitoring mechanism presented in Section 4.1 operating in learning mode (i.e., in an isolated environment that ensures nominal execution without outside interference),
- a policy specification language (e.g., similar to [Ham+16; HP17] that uses the KeyNote engine [Bla+99], or to the one used by `systrace(8)`).

In this Section, we present our own approach in defining a policy, as part of an internal report at TU Braunschweig (TUBS), by examining the update procedure of the Sophos Anti Virus (AV) Linux application that is distributed in the academic community at TUBS.

We point out here that the ideal way is a combination of the above. Optimally, the application-library designer/developer creates the policy in a specification language, by leveraging our monitoring mechanism. They are the most suitable to provide it, since they know their product best and are in a position (e.g., have access to the source code, can run extensive testing, etc.) to create a comprehensive and complete security policy.

4.3.1 File Organization

In [Pre17], Prof. Prevelakis managed to uncover the credentials used to update the AV, as part of an internal technical report at TUBS. By using these credentials, we manage to download all the files from the original update

website and set up a local mirror of the server, which contains three distinct sets of files:

(a) The ones that exist in the local server that we set up (server files),

(b) the ones that are created when the Sophos application is installed in a directory (install files) and

(c) the exact copies of the server files that exist in a folder inside the install directory, called cache (cache files – *<directory>/update/cache*).

4.3.2 Code Analysis

In order to run the update, the command *<install directory>/bin/savupdate* needs to be executed. Then, the control is transferred to the python scripts responsible for the update procedure. These scripts are inside *cidrep.zip* package, located in *<install directory>/update*. After careful run-time analysis, the following diagrams depict the relevant parts of the flow of execution within and among the scripts (Figures 4.5, 4.6, 4.7, 4.8 and 4.9).

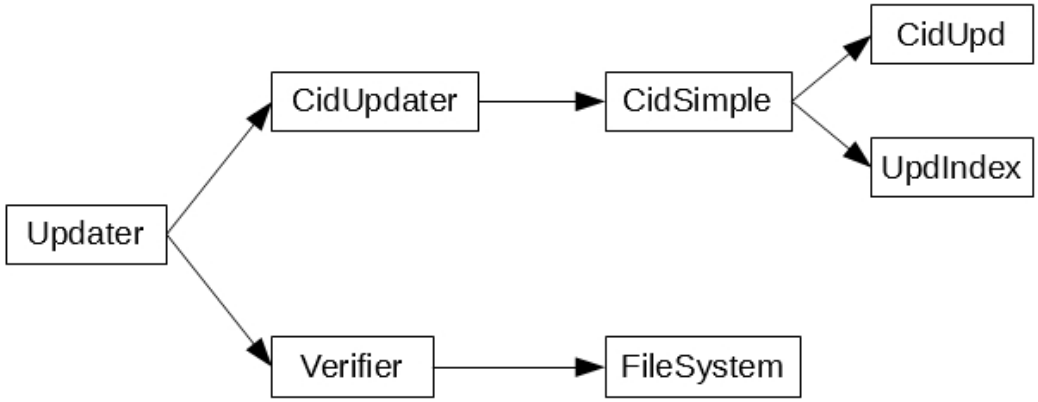


FIGURE 4.5: Overall sequence of the Sophos update procedure

Since, now, we have every possible execution path when Sophos AV receives an update, we can include them in a security policy that will be enforced at run-time during this procedure. For example, in Listing 4.1, we can see the security policy related to CidSimple (Figure 4.7). For a specific function in the file (in this case $replicate()$), there are a number of $next\{\}$ statements. Inside $next\{\}$, there can either be $file\{\}$ statements (which means execution is transferred to another file and subsequent functions), or more $function\{\}$ statements (referring to internal functions in the same file).

4.3.3 Compromisation Attempts

After producing the complete picture of the update sequence, as a step further, we experimented on if and how it was possible to compromise the application, by using what we learned so far.

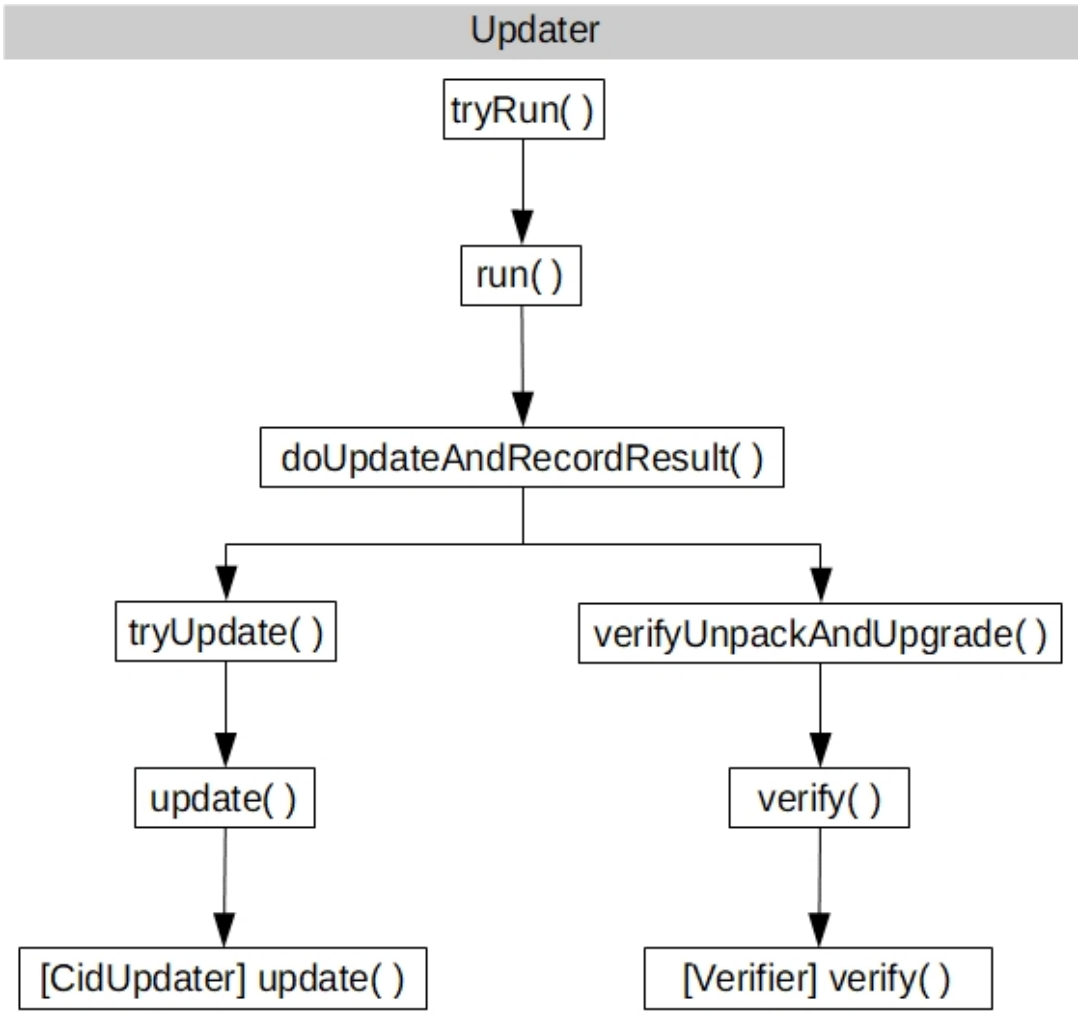


FIGURE 4.6: Execution paths in Updater.py file

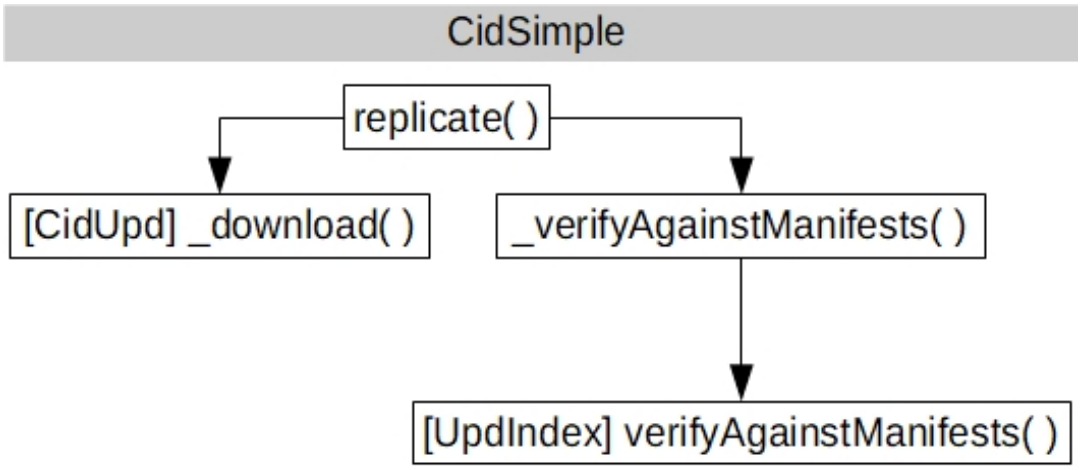
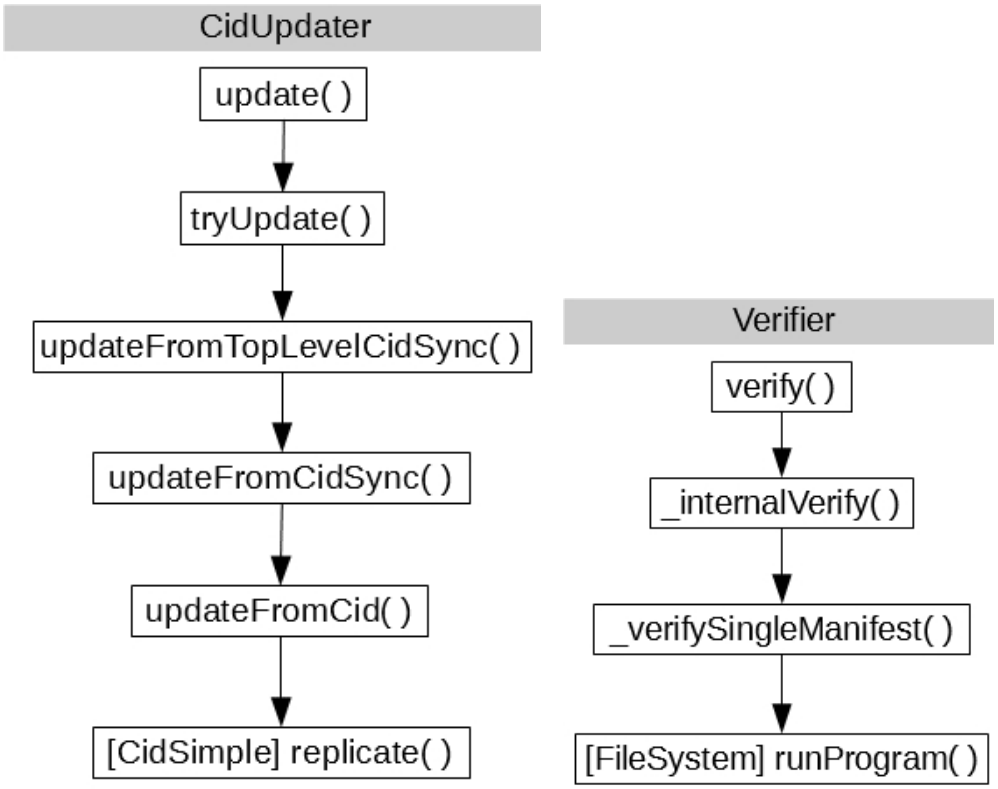
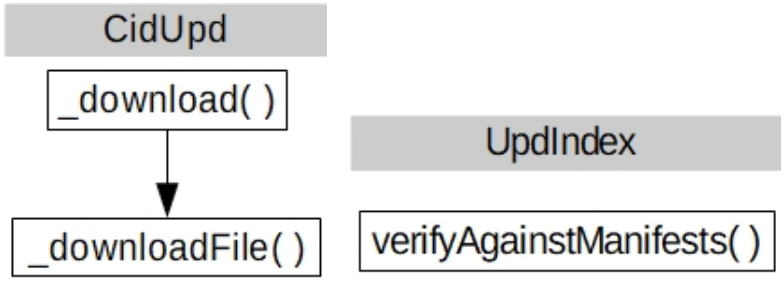


FIGURE 4.7: Execution paths in CidSimple.py file



(a) CidUpdater (b) Verifier

FIGURE 4.8: Execution paths in CidUpdater.py and Verifier.py files



(a) CidUpd (b) UpdIndex

FIGURE 4.9: Execution paths in CidUpd.py and UpdIndex.py files

When the update procedure starts, the application downloads the server file *cidsync.upd* which is a list of all the cache files that should exist. If one doesn't exist, it is downloaded from the server and the install file *<directory>/engine/versig* is executed to perform signature verification. The first approach was to identify the OpenSSL version that versig is using, to determine if there are any known vulnerabilities for it. Version 1.0.2j had some vulnerabilities, none of which were relevant to our cause.

The next step is to modify the *cidsync.upd* file that is downloaded when the update starts. The structure of the contents of this file, with respect to

```
1  file CidSimple{
2    function replicate{
3      next{
4        file CidUpd{
5          function _download{}
6        }
7      }
8
9      next{
10       function _verifyAgainstmanifests{
11         next{
12           file UpdIndex{
13             function verifyAgainstmanifests{}
14           }
15         }
16       }
17     }
18   }
19 }
```

LISTING 4.1: Example of a security policy

listed files, is as follows:

- four-byte number: length of entry in bytes [1]
- four-byte number: length of **file** in bytes
- four-byte number: checksum of file
- file path <00>

To change the contents of a listed file, the actions needed are:

1. After saving the modified file, we run the update once – instrumented with the custom library from Section 4.1 – to get the new file checksum with which we modify the respective four-byte number in *cidsync.upd* (the update fails)
2. Update once again to get the new checksum of *cidsync.upd* and change it in the file (last four bytes – update fails again)

With this in mind, we create a new *versig* file in order to bypass the signature verification step and perform an arbitrary action (e.g., create a file in */tmp*). Since the new *versig* file is smaller in size than the original one, its size inside *cidsync.upd* needs to be changed, too. Following the change procedure above, the update fails again because the size of the files is vetted against

[1] This is the number of bytes for the file length, file checksum and file path including the terminating NULL. It does NOT include the length-of-entry field itself

the manifests of the application. The manifests contain the path of each file, its size and its SHA1 hash and are digitally signed by Sophos, so cannot be modified. So, the custom *versig* file is padded to match the original's size and this works as expected. The signature verification is bypassed and a file is created in */tmp*. However, the *versig* file is located in the install directory, hence in order a potential attacker have access to it, they have to somehow gain root privileges (e.g., perform privilege escalation).

Another angle is to modify the python script responsible for the retrieval of the path of the *versig* file (these paths are hardcoded), in order to add a custom path of our own *versig*. The same principal as above (i.e., root privileges) applies here as well.

Other ideas were also entertained, but lead to non-applicable attempts. One of these ideas was to change the length of a file's entry in *cidsync.upd* to $0xFFFF$ to see if we could cause a reaction. This meant that the file path after the checksum would need to be (64k-8) bytes. But, the update procedure checks if all the files actually exist, so the file would need not only be real, but have a (64k-8) bytes–long path (besides, there is a hardcoded limitation of up to 512–byte paths).

Another direction was an SQL-injection-like approach. Changing the file names in order to execute python code did not lead to a potential attack. Before searching if a file exists, the update procedure stores all the file names in an array and then vets them against the manifests. If a name in the array does not correspond to any of the manifests, it is removed from the array. Then and only then, all the remaining files are checked. Parts of this work have been included in [Tsa18].

4.4 Summary

In this Chapter, we present an access control scheme that produces custom libraries and examines calls to functions within them along with their arguments, to ascertain if they adhere to specific security policies. Our approach improves important aspects of SecModule, in which it can be incorporated, simplifying and automating the generation of libraries and providing a seamless way of evaluating the arguments of each call.

Furthermore, we present a mechanism that compartmentalizes a library at an even greater granularity than normal, at a functionality/function level and allows users to load only specific parts of the library that are necessary for execution. This way, we do not only save resources, but increase security as well by performing more security checks, in conjunction with the approach detailed in Chapter 5.

Additionally, we present an approach on how to analyze the execution of an application, extract all information pertaining to the transitions at runtime and include them in a security policy.

5

Kernel-Level Monitoring of Library Call Invocation

The ability to monitor when user code invokes a library function offers numerous advantages. For example, during black-box testing of code, high-level CFI checking, run-time access-control policy enforcement and so on. However, for this technique to be useful it must be efficient and able to function even when the target application is provided only as a statically linked executable.

In Chapter 4 we demonstrated how library calls may be intercepted using wrappers. But this approach works only with dynamically linked code and requires user intervention - albeit a small one - and some technical expertise to process the function arguments. Furthermore, since it is essentially just another library of the user-space application, it is possible it can be bypassed by a technically-savvy attacker. Consequently, it cannot be used as a standalone access control/monitoring mechanism. However, its main advantage is that, after the shared custom library is produced, it offers the ability to monitor/sanitize the arguments of each call in detail.

In this Chapter, we present a more concrete technique which the previous one compliments. This approach operates on the secure kernel side, so it is much more difficult for an attacker to bypass it. It corresponds to Figure 3.4. Under this scheme, each library - either dynamically or statically linked - constitutes a separate code-region. At any point in time, only pages belonging to that one region are marked as executable, so when code branches to a page outside the "home" region, it lands in a non-executable page, a fault occurs and the kernel takes over. By adding suitable code to the kernel, we can determine (a) whether the call should go ahead, (b) whether the arguments are acceptable and (c) ensure that the kernel is informed when the code returns from the called function.

The rest of the Chapter is organized as follows: Section 5.1 provides the

motivation behind our approach to monitor the execution of a program. In Section 5.2, we present the design on top of which our idea is based. In Section 5.3, we delve into details on how we implement it. Section 5.4, contains a use-case study where we apply the mechanism on a vulnerable application. In Section 5.5, we present a training environment that includes parts of our framework. Finally, we summarize the Chapter in Section 5.6.

5.1 Execution Monitoring: Challenges

As the advancement of technology offers capabilities which result in attackers on every level getting more competent and effective, attacks have become more elaborate. Therefore, we need to establish an adequate level of security in software systems. Complete security of a program is unfeasible and so it becomes imperative to detect a situation where a program enters a possibly insecure state and take some action to respond to it. Our idea is to implement such actions at an abstract level, between the OS and a running application.

Mechanisms such as CFI [Aba+05], `systrace(8)` [Pro03], SecModule [KP06], etc., aim to control the behavior of a program based on some predetermined policy. When the program attempts to perform some action that is in conflict with its execution policy, the security mechanism detects this and takes corrective action. Unlike most runtime security mechanisms, where violations in most cases lead to the termination of the offending process, the call intercept technique offers a variety of options in dealing with the security breach. For example, `systrace(8)` may rewrite function arguments (e.g. truncate or replace strings), or return an error without making the call.

A key concern is the level at which the call intercept should take place. Some papers propose to carry out the checks at the machine language level, via call graphs, while others at the system call level. We believe that carrying out the analysis at an even higher level has many benefits, not least being close to the application logic. With this in mind, we chose to base our system on the monitoring of library calls. We, therefore, had to consider the performance penalty of carrying out our high-level monitoring.

Our compromise is to designate the main application and each library as separate areas and monitor program jumps from one area to another. We then program the MMU in such a way so as to cause a fault when the control flow moves from one area to another. We do this by designating all code pages outside the currently executing area as non-executable. In this way, a jump within the current area is carried out normally, but a jump to a different area causes a fault. This technique is very efficient, while at the same time it allows `systrace`-like control of function invocation, argument checking and modification etc. Parts of this work have been published in [TP19; TP20; TP21a; TP21b].

5.2 Design Overview

The goal of our approach is to thwart control-flow hijacking attacks by segregating a process's executable areas which correspond to external libraries or the main executable and by imposing strict control over any attempt to invoke such an area, through a policy enforcement engine. In order to achieve this, in Section 3.1 we set a number of requirements that our mechanism must fulfill. To abide by these requirements, we provide a customized Linux kernel that leverages the MMU in order to separate the memory of a process into regions, based on the libraries that are loaded upon a program's execution. When an untrusted, user-space application issues a call to a protected library, our custom kernel intercepts it and redirects it through a policy decision mechanism, before allowing it to continue.

In order to intercept all the calls inside the libraries that an application uses, our system loops through all the memory regions of a running process and identifies all the executable ones (e.g *.text*/code region) that correspond to a linked library or the main executable. These regions are where the actual executable code is found. For each of these regions, it maps a special custom library - which we designate as *gate* - in the running process's address space and associates it with the corresponding region. Then, it marks these regions as non-executable. When a program is running it issues, for example, a call inside *LibX*. After the transition, all the other executable regions (e.g., of *main*, *LibY*, etc.) have their NX-bit set. If from there the flow of execution is transferred to another library, the same procedure happens again. In a similar fashion, the regions change from executable to non-executable when the flow returns upon execution completion.

This procedure ends up causing a page fault when the process tries to call a library, since all of its functions are located in currently non-executable regions. The Page Fault Exception Handler (PFEH) [BC05] is, then, invoked to resolve the issue. In order to continue executing, we modify the PFEH so it redirects the flow inside the associated *gate* library that we had previously mapped in the process's memory area. If the policy check is passed, PFEH marks the specific non-executable region as executable and continues with the call. After it is completed normally, we may need to make the region non-executable again (more information about this in Sections 5.3.4 and 5.3.9) and the whole procedure starts from the beginning. Figure 5.1 shows this sequence of events.

In order to avoid being overly verbose, we choose to monitor the execution at an abstract level. This approach operates at the granularity of a library/main executable, meaning that we concern ourselves only with each external call to a specific library and not with how - internally - a library handles a call to one of its own functions, which is left to the library designer. For example, consider when a program calls the $printf(3)$ function from *libc* library. Internally $printf(3)$, first, parses the arguments of the call, i.e. output format (how) and printable arguments (what) and then passes them as input to $vfprintf(3)$, another function of *libc*. Then, $vfprintf(3)$ analyzes these inputs and displays the text in the desired format. We expect that the

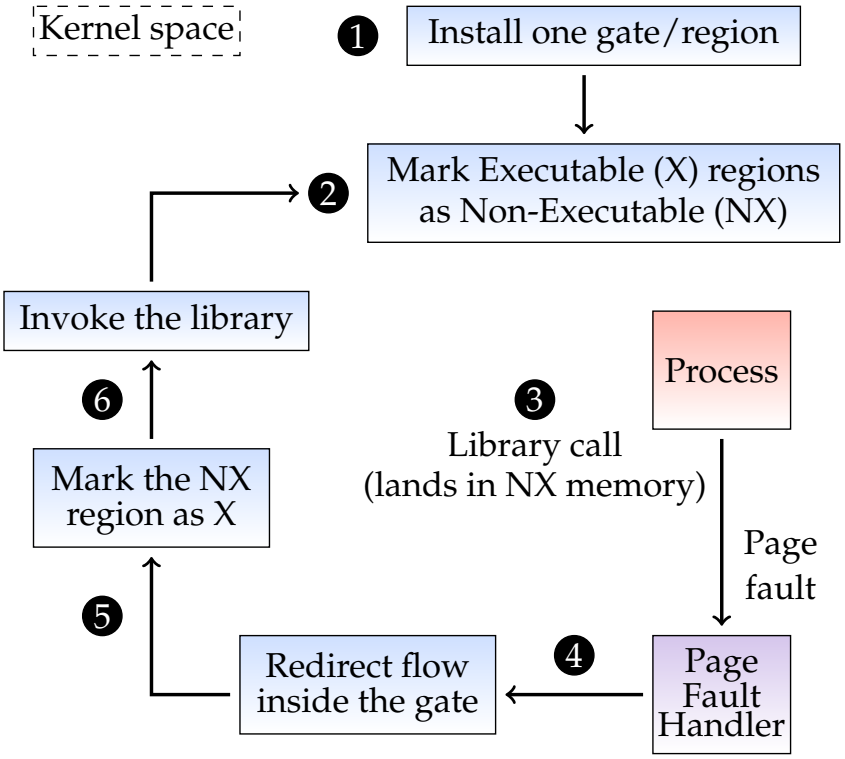


FIGURE 5.1: Kernel intercept design

library/application designer provide us with the correct and complete security policy associated with their product, in order to be able to identify the complete list of permissible function calls. In a different case, the list will not be as comprehensive (i.e., when the system administrator is responsible for creating it).

5.3 Implementation Specifics

To actualize our approach, we used Linux kernel version 4.16.7, the latest one at the start of the development phase. We customized the $\_do\_fork()$ function which is responsible for creating new processes/threads. Upon process creation, our mechanism identifies the regions of contiguous virtual memory (Virtual Memory Areas - VMAs) that are executable and correspond to a linked library or the main executable (similarly to the `/proc/<pid>/maps` kernel utility) and maps the custom *gate* library in the process's memory, one for each identified VMA (Figure 5.1 (1)). Then these VMAs are marked as non-executable (Figure 5.1 (2)). This is done by changing the flags of a VMA. These flags designate the VMA access rights (e.g., if it is read-only, writable, executable, etc.). In our case, we are interested in the `VM_EXEC` flag. The modification of it can be one of two actions: (a) clear, which means the flag becomes zero "0" and execution is forbidden from the VMA and (b) set, which means `VM_EXEC` becomes one "1" and the VMA becomes executable.

5.3.1 Page Table Entries

Additionally, we modify all the Page Table Entries (PTEs) that correspond to the identified VMAs. In memory, there is a mapping between virtual and physical addresses. Each of these "translations" is stored as an entry at an in-memory table called Page Table (PT). Dedicated to it is a cache of the most common PTEs, the Translation Lookaside Buffer (TLB), which is used to boost performance when accessing a memory location. Every time we want to modify a PTE, we perform a page table walk (go through the multi-level PT [Shu16] in its entirety) (Figure 5.2) to find a specific PTE and then clear the processor cache and the TLB from any record of it, in order the modification take effect. Modifying all the PTEs of a VMA is computationally expensive, however it is performed only once per process/thread, which keeps the performance overhead to a minimum.

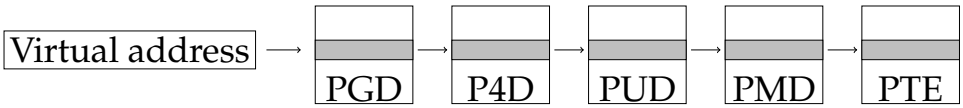


FIGURE 5.2: Page table walk

5.3.2 Deliberate Page Fault

After a VMA and corresponding PTEs are marked as non-executable, when a process tries to perform a library call, it will land on an address in a non-executable page (Figure 5.1 (3)), which will result in a page fault. An error code is associated with every fault that represents what kind it is. The bits of an error code can be found in Table 5.1 [Err]. In our case, the error code produced has a value of $15_{16} = 21_{10} = 010101_2$. When we correlate this number with Table 5.1, we can see that the bits X86_PF_PROT, X86_PF_USER and X86_PF_INSTR have a value of one "1". This means that this was a protection fault caused by a user-mode read access (X86_PF_WRITE is zero "0") when an instruction fetch was attempted. In other words, a user-space application tried to read from a page that is protected against this action.

When a page fault occurs, the PFEH [BC05] intervenes to handle it. This is done through the interrupt service function `do_page_fault()` (Intel x86-64 architecture). At this point and before the function has a chance to address the fault by itself, we step in and redirect the execution flow inside our policy enforcement engine (Figure 5.1 (4)), which we had previously installed for each VMA.

5.3.3 NX-bit

After the *gate* checks the security policy and allows the call to continue (Figure 5.1 (5)), we need to make the memory executable again, which we accomplish based on the virtual address of the function that caused the page fault. Both the PTE and the VMA that correspond to a specific virtual address need

Name	Value	Description
X86_PF_PROT	1 << 0	0: no page found 1: protection fault
X86_PF_WRITE	1 << 1	0: read access 1: write access
X86_PF_USER	1 << 2	0: kernel-mode access 1: user-mode access
X86_PF_RSVD	1 << 3	1: use of reserved bit detected
X86_PF_INSTR	1 << 4	1: fault was an instruction fetch
X86_PF_PK	1 << 5	1: protection keys block access

TABLE 5.1: Page fault error code bits

63	62	...	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	...	0	0	1	1	1	1	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1	1	0	0	0	0	0	0	0	1	0	0	1	0	1

(a) Bit values of an executable page

63	62	...	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	...	0	0	1	1	1	1	0	0	0	0	1	0	0	0	0	0	1	0	1	1	1	1	0	0	0	0	0	0	0	1	0	0	1	0	1	1

(b) Bit values of a non-executable page

FIGURE 5.3: PTE value comparison of executable and non-executable pages

to be executable if code from that address is to be executed, so we perform the inverse procedure than before (i.e., set `VM_EXEC` flag and clear NX-bit).

In the Intel x86-64 architecture, the most significant bit of a PTE is the NX bit [WX03], which is used to designate a page where execution of code is not permitted. Inversely from the case of the `VM_EXEC` flag, the modification of the NX-bit can be: (a) set, which means the NX-bit becomes one "1" and execution is forbidden from the page and (b) clear, which means the NX-bit becomes zero "0" and the page becomes executable. For example, in our implementation, when the NX-bit is clear, the value of a specific PTE that represents an executable page can be seen in Figure 5.3(a). After we set it, the value becomes that of Figure 5.3(b). Comparing the two values, we can see that the only difference is in the most significant (64th) bit which is the NX-bit. Bit numbers 36 through 61 have a zero value and have been omitted for convenience.

5.3.4 Shadow Stack

After clearing the NX-bit and setting the `VM_EXEC` flag, the library call continues (Figure 5.1 (6)). Following its completion, we may need to make the region non-executable again (Figure 5.1 (2)). In order to determine this, at

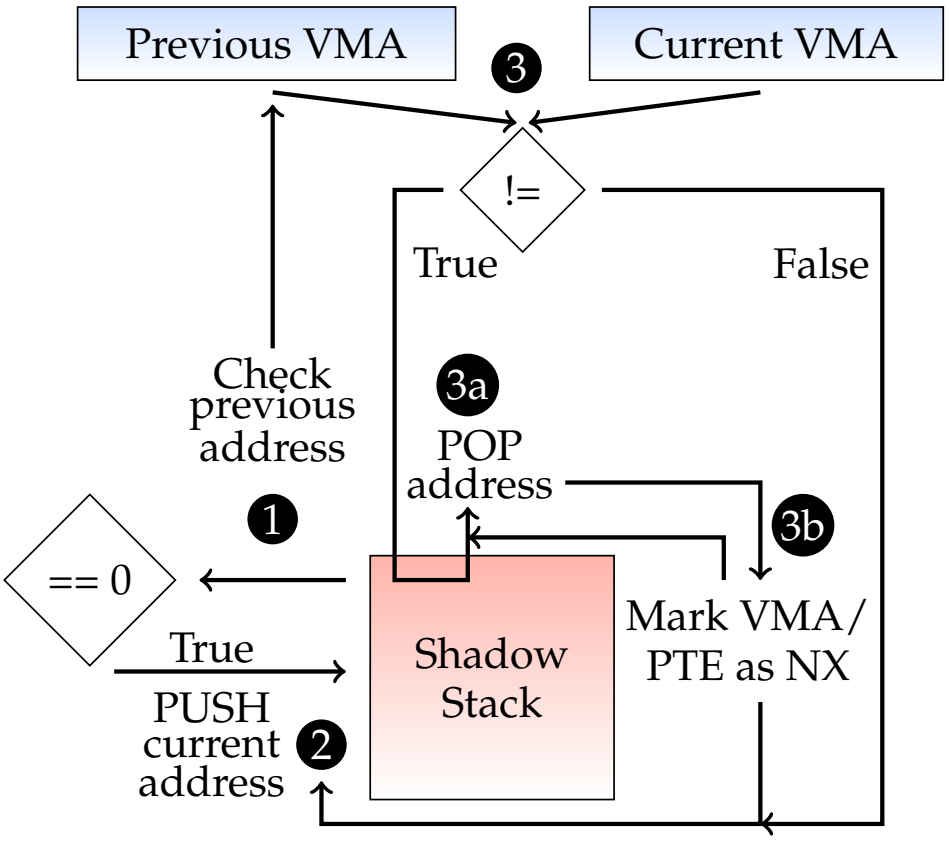


FIGURE 5.4: Leveraging a compact-like shadow stack

first we leveraged the idea of shadow stacks [CH01; DMW15; BZP18]. Originally, a shadow stack is a separate protected memory region where an attacker cannot have access. When a function is called, the return address of that call is stored in the shadow stack. When the function returns, the saved return address is either compared against the program's return address on the main stack, or it is placed directly on the main stack overwriting that return address. In our case, we implemented a compact-like form of shadow stack [BZP18] to save the virtual addresses where the page faults occurred.

In order to adhere to the library-level granularity of our design, we first check whether there is a previous faulting address stored in the shadow stack (Figure 5.4 (1)). If there is none, we store the current one (Figure 5.4 (2)). If there is, we check to determine if it corresponds to the same VMA as the current one (meaning same executable/library) (Figure 5.4 (3)), in which case we leave the VMA/specific PTE as executable and push the current faulting address in the shadow stack. Even if the VMA remains the same in the next iteration, the PTE will have changed to a new non-executable one which will produce the next page fault to be intercepted.

If the previous and current VMAs are different (meaning different executables/libraries by extension), all the saved addresses in the shadow stack are popped and their corresponding PTEs and VMA (i.e., the whole previous VMA) are marked as NX (Figure 5.4 (3a and 3b)), before pushing the current faulting address in the shadow stack. At this stage all the addresses of our

process are in a non-executable state, but the execution is in the context of the PFEH. From there it is redirected inside the *gate*, where the security policy is checked. If permission to continue execution is granted, which means that the current faulting address is inside the permissible addresses list, the VMA and PTE that correspond to that address are marked as X and the whole sequence starts again.

This procedure is performed automatically on the kernel side, without requiring access to the source code/binary of the application or linked libraries, thus making our approach completely transparent.

5.3.5 Kernel Modifications

In order to implement our system, we had to customize three key data structures inside the kernel, as well as the fault handling and process creation routines. However, these modifications are minor and incur only minimal increase in the Trusted Computing Base (∼500 LoC) and run-time overhead.

vm_area_struct defines a Virtual Memory Area (VMA) [Vma]. *A VMA is any part of the process's virtual memory space that has a special rule for the page fault handlers (e.g. a shared library, the main executable etc.)*. We added an extra field that designates a special VMA, where our *gate* library is mapped in. We chose to add the extra field here so it is easier to associate the special vma with the executable vma.

mm_struct stores information on the MMU state and the address space that the running process belongs to [Mms]. We added an extra field to store the offset of the executable area of our library, which we calculate at the mapping stage. We chose to add the extra field here so it is easier to retrieve the offset when we intercept the page fault and redirect execution to the proper address inside the *gate*.

task_struct stores information about a process [Tas]. There is one such data structure associated with every running process in the system. We added two extra fields, one to associate a shadow stack with the process and one to designate if the process has the *gates* already mapped in. We chose to add these fields here, because they are required on a per-process basis.

__do_page_fault() is the function responsible for handling the page faults when they occur [Dop] - essentially the PFEH. We customized it to intercept the specific page fault caused by our approach and perform the redirection.

_do_fork() is the function responsible for creating all the processes/threads [Dof]. This is where the initial NX marking takes place. We customized it in order to be able to instrument all the instances of a process in the system.

ShadowStackNode is a custom structure that represents a shadow stack node. It is used to hold a faulting address and point to the next node in the shadow stack. Adding one, dynamically enlarges the shadow stack, while removing one shrinks it.

5.3.6 Custom Variables

In order to instrument a process that we are interested in, we iterate through the list of the kernel's processes and select the specific one. This is done after the process starts. There is also the option to do it as it starts, by implementing a *kprobe* in the $\_do\_fork()$ function.

There may be several instances of a process, so after having selected a specific one, we first check to see if it has already been instrumented, in which case we simply ignore it. Continuing, we set a custom variable of the process's address space (*mm_struct*) to the start of the executable area of *gate_library.o* (see Section 5.3.7 - we'll use this later). We, also, initialize a custom variable of the process (*task_struct*) which we will use to check if the current and previous memory areas are different, so we can adhere to the library-level granularity.

5.3.7 Gate Library

As the first step in our approach (Figure 5.1 (1)), we need to identify all executable VMAs and for each one map a special VMA that corresponds to our *gate* library and associate it with the related VMA. The *gate* is essentially the policy enforcement engine and includes a list of allowed executable addresses (offsets) inside the library. These addresses mostly correspond to the start of library functions, although there may be unrelated addresses where the program jumps to under normal execution. This list is the security policy that needs to be checked in order to allow the call to continue or not.

According to our security assumptions (see Section 3.3), we expect the library designer to provide us with the complete and correct list. However, for testing purposes, we leverage our previous approach in [TP17] to extract the relevant addresses from the OpenSSL library. The format of the policy is `"<function>", <offset>`. In order to perform the policy check, we traverse the list and compare the calculated offset of the faulting address against the offset in each element.

In order to compile the *gate* library, we leverage the way the kernel compiles the *vDSO* library [Ker]. The vDSO (virtual Dynamic Shared Object) is a Linux kernel mechanism that maps a special library in the address space of each process automatically. This library exports specific kernel-space read-only functionality to user-space (e.g. read a variable), in order to dispense with the otherwise necessary system call and reduce overhead. Whenever the kernel is built, it compiles the vDSO and when a process starts, the kernel maps the vDSO in its address space dynamically, so it can use the exported functionality. We compile our custom library in a similar way, using also a customized version of the special linker script that the kernel uses for

the vDSO. Based on the assumptions that we make about the security of the underlying system (see Section 3.3), we can digitally sign the policy enforcement engine, to prevent it from being tampered with. The compilation as a shared library is done with the command in Listing 5.1.

```
1 gcc -nostdlib -o gate_library.o -fPIC -shared
2 -Wl,-T,gate-layout.lds.S -Wl,--hash-style=both
3 -Wl,--build-id -Wl,-Bsymbolic -m64
4 -Wl,-soname=gate_library.so -Wl,--no-undefined
5 -Wl,-z,max-page-size=4096 -Wl,-z,common-page-size=4096
6 -e policy_enforcement gate_library.c &&
7 objcopy -S gate_library.o
```

LISTING 5.1: Compiling the *gate*

Mapping and Inserting Gates

After opening the gate library object file (*gate_library.o*), we read it and copy it to memory. Then, we perform some simple consistency checks regarding its type, architecture, and program headers. Following, we map it to a kernel memory area. In order to do that, we first loop through all the VMAs and select the executable ones based on their flags (`vma_mmap->vm_flags & VM_EXEC`). However, there are two cases in which we ignore a selected VMA:

- If it is a *gate* VMA (one that we have previously inserted). We don't want to change anything there.
- If we have already associated it with a *gate* VMA. With each executable VMA we associate one *gate* VMA, where we redirect execution.

In order to be able to perform these checks, we modify the *vm_area_struct* (see Section 5.3.5) by adding an extra field (*special_vma*) that designates the special VMA (where our *gate* is mapped in) that is associated with the executable VMA. Furthermore, we add another field (*is_special_vma*) to show if the VMA is a *gate* or not.

If these checks are passed, we get the starting address of a free (unmapped) area from *mm_struct*, just big enough to fit the *gate*. After some sanity checks on the address, we allocate some space in the kernel *vm_area* slab cache (*kmem_cache_zalloc*()), we insert the *gate* in the process's address space and finally remap it to user-space, in order the appropriate physical pages be associated with it.

5.3.8 Private Memory Mapping

Separation of data used by the libraries from data used by the running application is another significant aspect of our security framework. Originally, the

application and library code share their stack and heap spaces, which provides a breeding ground for interfering with the execution of library code. If a security mechanism controls entry to a particular region (e.g., a library), it can also enable access to private data pages associated with that region. When the CPU executes code within a given region, its private pages are mapped. When execution is transferred outside the region, the associated pages get unmapped and the private memory becomes inaccessible. In this way, each region can maintain state (e.g., which part of the program tried to access it, how many times a given routine has been called, or the sequence of calls to its various functions).

In this Section, we refer to our effort to implement the notion of a TEE at the memory space of a user application. Hardware vendors (e.g., Intel) have already implemented the concept of TEE into their products (e.g., SGX technology). Virtual TEEs (e.g., Open-TEE [Lim21]) allow developers to create trusted applications using the GlobalPlatform TEE specification [Glo21]. Related work proposes mechanisms that push sensitive objects into isolated memory regions, called *safe regions* [Kon+17], that can only be accessed by privileged program instructions. These regions protect sensitive program data, such as code pointers [Kuz+14], cryptographic keys [Gua+15], or programmer-defined structures [CP17; Dau+15]. Access to these regions is achieved in two ways: either (a) monitor all unsafe accesses (e.g., sandboxing them using SFI or MPX [CP17; Kon+17; Wah+93a; Seh+10]), or (b) switch between protection domains (as in SGX, TrustZone, MPK, VMFUNC [CP17; Kuv+17]). Other implementations enforce policies at instruction/word granularity [Dha+15; Son+16; CCH06], but require significant hardware enhancements [MRD18].

In our case, since we already have a mechanism that allows us to rewrite the Page Table whenever a library boundary is crossed, we can now extend it by programming the MMU in such a way so as to map protected private pages for every library into the address space of a running program, which are accessible only by specific functions inside the *gate* libraries of said program. Upon interception of a library call, our system - after redirecting the call through the *gate* - determines if the call can access the information stored securely in the newly-mapped private memory. These pages are only mapped when the CPU executes code within the associated library. When execution is transferred outside the library, the pages get unmapped.

In this way, we protect sensitive code/data inside a secure enclosure, which need protection against disclosure, tampering, execution, etc. Consequently, we minimize what can access them, as we limit their exposure to only a specific set of legitimate functions found in the *gate* library, imposing serious limitations on what actions can be performed on the protected data, by what part of the program and at which point in execution time. Figure 5.5 shows a representation of this approach. This procedure is also performed automatically and transparently to the execution.

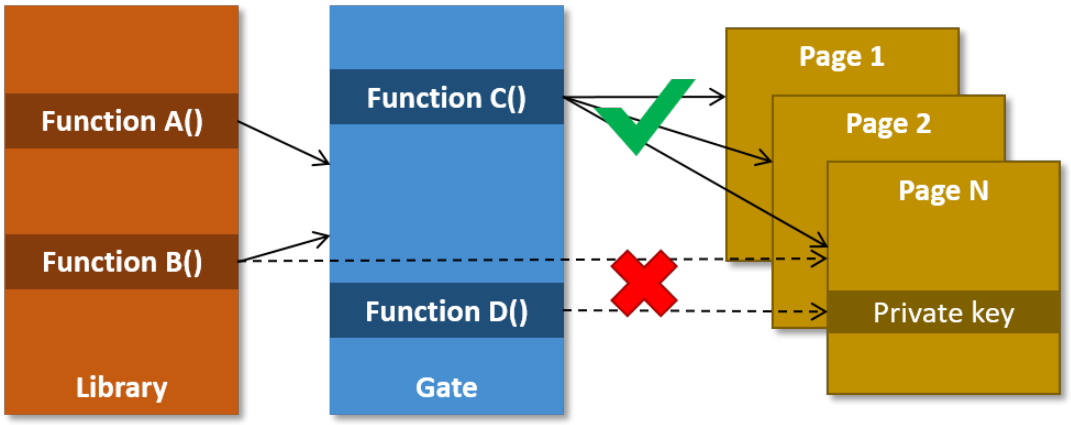


FIGURE 5.5: Secure memory mapping

Application Programming Interface

In order to facilitate the use of this extended capability, we propose an Application Programming Interface (API) analogous to the one used for shared memory [Ker08]. Listing 5.2 showcases a sample of our API, where the code has the ability to allocate a private memory space to a specific region.

```
1  ...
2  char *addr;
3  int fd;
4  fd = scrm_open(PAGE_SIZE, <FLAGS>);
5  addr = mmap(NULL, PAGE_SIZE,
6              PROT_READ | PROT_WRITE,
7              MAP_PRIVATE, fd, 0);
8  scrm_assoc(<caller>, fd, addr,
9             addr + PAGE_SIZE);
10 ...
11 scrm_unlink(fd);
12 ...
```

LISTING 5.2: Usage example of Secure API

First, we create a secure memory (scrm) object with specific flags and its size set to that of a page (line 4). Then we map the object into the process's address space (line 5). Following, we associate the object with the caller (a given region) in line 8. Finally, after some processing we return the memory to the system, by unlinking the scrm object. By using this interface, library designers can take advantage of private memory to protect internal data structures e.g., making them completely inaccessible (no read/write/execute rights) to the rest of the program.

5.3.9 Optimizations

During the course of development two major issues arose that caused crashes when instrumenting complex applications: (a) the speed of our version of a shadow stack and (b) synchronization when marking PTEs as NX, as well as when instrumenting all subsequent processes of a specific process.

Shadow Stack

Initially, we were concerned about the large number of PTEs that needed to be updated every time we cross a library boundary. However, we assumed - and our experiments confirmed - that only a small subset of pages of a given library need to be executable (and will consequently produce a page fault) when execution moves from one region to another. For this reason, at first, we decided to use the shadow-stack-like structure (described in Section 5.3.4), in order to identify this subset of PTEs that actually need to be modified during the transition. In this way, we achieved marking each previous library as non-executable without incurring the overhead of having to update thousands of entries in the page table; only the ones where a page fault occurred.

Furthermore, in order to adhere to the library-level granularity of our design, we employ *lazy* clearing of the NX-bit i.e., we perform it only for the current faulting address. As an artifact of this approach in our implementation, we had additional faults while execution remained within the same library in-between calls (which resulted in a number of faulting addresses being added in the shadow stack as described before), in which case we refrained from going through the *gate*.

While the shadow stack approach works perfectly well for simple applications with small memory footprint, after extensive testing, we identified that there were issues when dealing with more complex applications. In the implementation of our initial version (Figure 5.4), we would dynamically allocate (using $kmalloc()$) some kernel memory when inserting an item and free it (using $kfree()$) when removing an item, from the shadow stack. These functions entail allocating and freeing memory in the slab CPU cache among other things, but they were performed for every page fault that we caused (meaning in high numbers). This led to a bottleneck, since the cache operations are much faster. Consequently, it caused continuous crashes when trying to free memory from the slab cache (specifically in $\_\_slab\_free()$) and resulted in the mechanism not being functional. We realized that only one item was added to/removed from the shadow stack in each cycle. Hence, we modified the custom shadow stack field in *task_struct* into a simple variable and removed the *ShadowStackNode* representation from the kernel (see Section 5.3.5), dispensing with the overhead of the extra memory operations and the subsequent crashes.

The main takeaway from this realization, is that we manage to lower the performance/memory overhead even more, since we now need space and CPU execution time to save only one faulting address and not tens of them. Secondarily, our framework takes up even less LoC than before. Figure 5.6

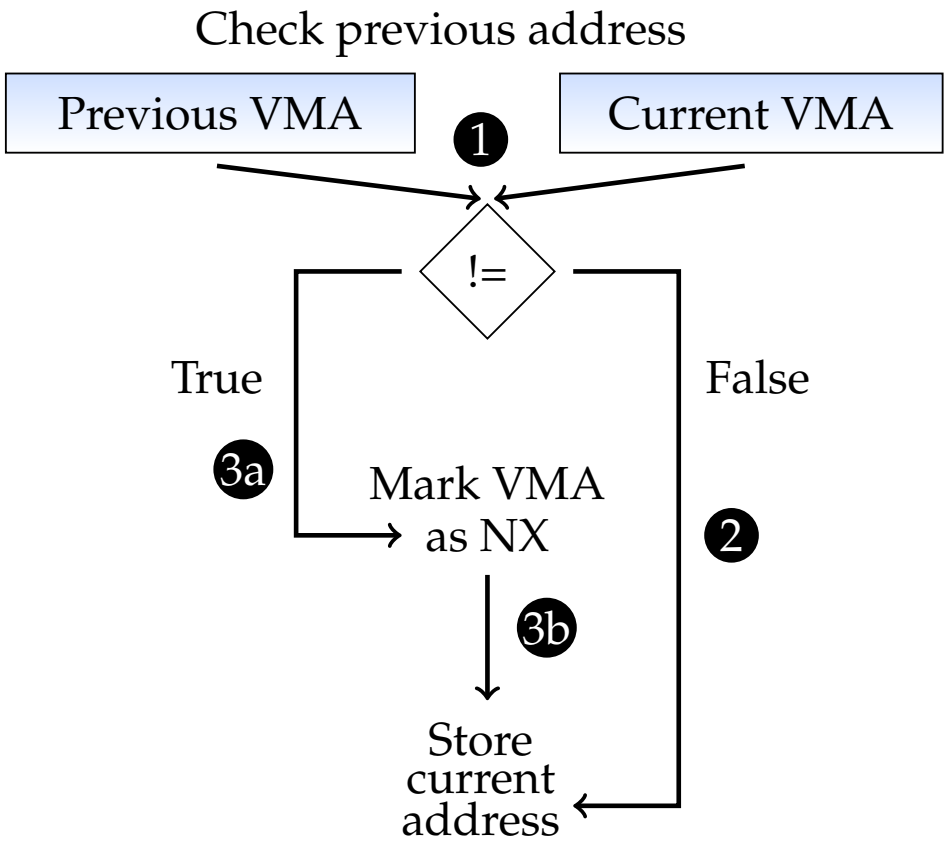


FIGURE 5.6: Compartmentalizing an application at library-level granularity, without a shadow stack

shows the updated approach. As is evident, the basic idea behind compartmentalizing an application at library-level granularity is the same as in Section 5.3.4, having eliminated however the use of the shadow stack.

Synchronization

Initially, we followed the procedure described in Section 5.3.1 (marking both VMAs as well as corresponding PTEs as NX) at the starting phase of a process. However this caused several issues. To mark a VMA as NX, it is sufficient to change its flags (`vma_mmap→vm_flags &= ~VM_EXEC`), which is done in a single function call (*vma_set_page_prot*()). On the other hand, a VMA comprises of a very high number of addresses which we must loop through individually, if we want to mark the corresponding PTEs as NX. At first, we were performing both of these sequences to mark a VMA as NX. However, at the PTE marking stage, as we started to iterate the addresses, the corresponding PTEs became NX in each iteration. So, when a call was issued to an address before this procedure was completed, if the corresponding PTE was already NX then the page fault was caused, as it should be. However, if the PTE was still X, there wasn't a page fault thrown and the execution continued as if there was no instrumentation. This led to undefined behavior, which resulted in crashes in some cases. More importantly, we managed to

identify that this is a typical race condition (which can be taken advantage of under specific circumstances), which is a difficult task in itself.

Another synchronization issue when marking all relevant PTEs as NX, has to do with the spawned processes (children) of a specific process (parent). When a child is initialized, it inherits the parent's memory layout. However, in most cases the PTE NX marking in the parent was not finished by the time the child was initialized. This left the child in an inconsistent state and resulted in erratic behavior and crashes.

For these reasons, we decided to not perform the PTE NX marking in the starting phase, since we detected that it is not mandatory for our mechanism to function correctly. This resolved all aforementioned issues.

5.4 Use-Case Study

In this section, we present a scenario where a vulnerability of an application is exploited to affect the underlying system. In our use-case, we use a vulnerable version of NGINX HTTP server, where a buffer overflow is triggered under specific circumstances to launch a Denial-of-Service (DoS)/arbitrary code execution attack, in order to compromise the application. By using our custom kernel to observe the calls the server issues to external library functions (and the subsequent internal library calls), we can better understand the behavior of the attack and provide a signature of the way it works. Hence, we can recognize it when it happens and block it before causing any damage.

As mentioned in Section 3.5, CVE-2013-2028 [Com13] is a stack-based buffer overflow vulnerability that affected NGINX server versions 1.3.9 through 1.4.0 with the default setup. Although it was addressed in later versions, we can use our prototype to examine the chain of calls that NGINX makes to its external libraries (as well as the internal library calls), which result in a crash or remote shell when the vulnerability is exploited.

5.4.1 Applying the custom kernel/MMU

There are several exploits available online [sor13; Mer13; Mhs13; w0013; dan13] implementing either a DoS attack or a CRA as proof-of-concept. In order to prove the applicability and effectiveness of our custom kernel, we chose a ROP attack described in [w0013] due to the simplicity of the exploitation script and its impact in real-world applications.

From the excerpt of the output of our approach in Figure 5.7, we can see what happens when the exploit is run against the vulnerable NGINX server. We can distinguish both cases where the current executable/library is both different and the same as the previous one. For example, in lines 8-16 the flow of execution is inside the main executable. In lines 9 and 10, we can see that when the fault occurs, the VMA is non-executable ($r--p$), as well as the PTE (NX-bit 1, SET). Continuing, we mark the current PTE and VMA as executable (NX-bit 0, CLEAR and $r-xp$). Since previously (lines 1-7) the flow was inside a different library (*libpthread*), that library is now marked

```
1  [88.812529] ===[2578] start __do_page_fault 7f358be3e890===
2  [88.812531] 7f358be38000-7f358be50000 r-xp /lib/x86_64-linux-gnu/libpthread-2.23.so
3  [88.812533] NX bit before: 1, SET
4  [88.812538] NX bit after:  0, CLEAR
5  [88.812539] 7f358be38000-7f358be50000 r-xp /lib/x86_64-linux-gnu/libpthread-2.23.so
6  [88.812540] prev_addr: 7f358be3db30
7  [88.812540] ===[2578] end__do_page_fault 7f358be3e890===
8  [88.812556] ===[2578] start __do_page_fault 41d2d0===
9  [88.812558] 400000-495000 r--p /usr/local/nginx/sbin/nginx
10 [88.812560] NX bit before: 1, SET
11 [88.812560] NX bit after:  0, CLEAR
12 [88.812561] 400000-495000 r-xp /usr/local/nginx/sbin/nginx
13 [88.812562] prev_addr: 7f358be3e890
14 [88.812563] prev NX bit before: 0, CLEAR
15 [88.812567] prev NX bit after:  1, SET
16 [88.812568] ===[2578] end__do_page_fault 41d2d0===
17 [88.812579] ===[2578] start __do_page_fault 41ea5e===
18 [88.812580] 400000-495000 r-xp /usr/local/nginx/sbin/nginx
19 [88.812582] NX bit before: 1, SET
20 [88.812583] NX bit after:  0, CLEAR
21 [88.812584] 400000-495000 r-xp /usr/local/nginx/sbin/nginx
22 [88.812584] prev_addr: 41d2d0
23 [88.812585] ===[2578] end__do_page_fault 41ea5e===
```

FIGURE 5.7: Custom kernel output after exploitation

as non-executable. However, in the next iteration (lines 17-23), although the VMA remains the same so it is executable as before (lines 18 and 21 $r-xp$), the PTE changes (line 19), so we get a page fault which we intercept and rectify (line 20) in order to continue executing.

As the exploitation unfolds, we monitor its progress and we manage to identify a characteristic sequence of calls to executables/libraries (including the internal ones) and produce a trail of it (Figure 5.8). Consequently, the next time our mechanism recognizes the same fingerprint, it will be able to intercept and mitigate the attack.

5.4.2 Performance Evaluation

We leverage the Phoronix Test Suite (PTS) [Pts] to evaluate the efficiency of our mechanism. Specifically, we run the OpenSSL benchmark test, since we have already instrumented the relevant library (Section 5.3.7). Our test-bed is mentioned in Section 3.4.

We run the benchmark test for both cases: (a) default untouched kernel, (b) kernel with customized MMU. The outcome reports on the number of RSA 4096-bit sign operations per second. Figure 5.9 shows details of the collected data (rounded numbers) for 5 repetitions of the test, while Figure 5.10 summarizes the results.

The default kernel is consistent in its performance, as is evident, since it is optimized by-default. It outputs 696 signs per second in each iteration (the same number occurred in other test runs, as well).

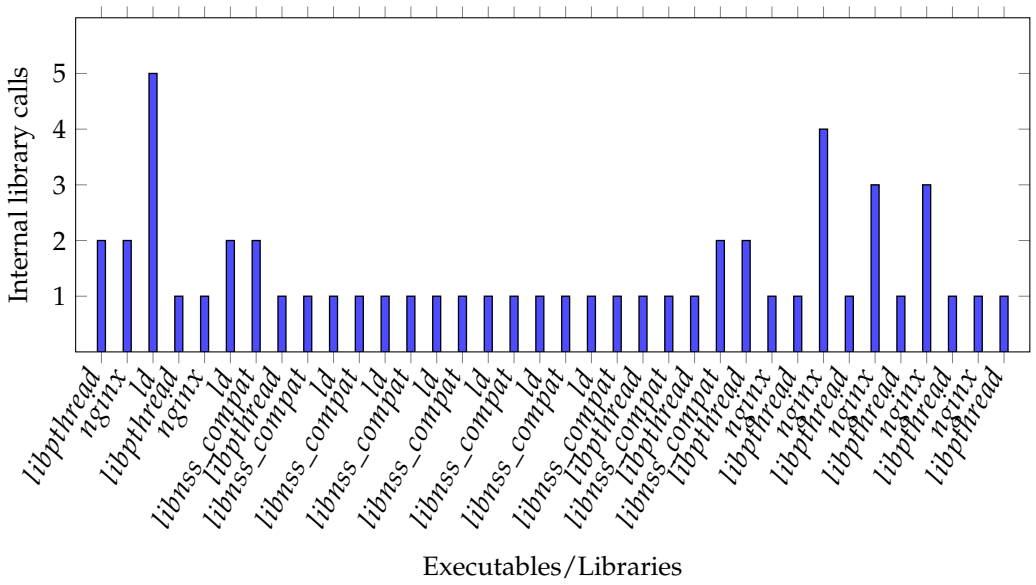


FIGURE 5.8: Sequence of libraries and number of calls inside each library

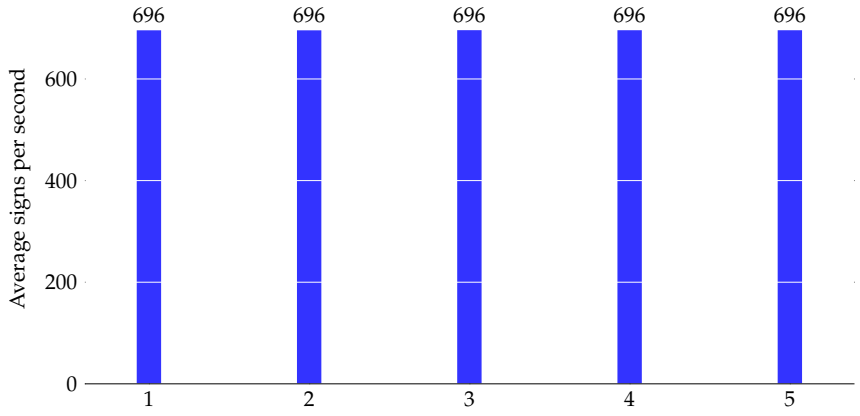
The customized kernel, on the other hand, presents some minor fluctuations, especially at the start of the testing. The first iteration is always higher in output (690 signs/second). After that, the rest of the phases are normalized in lower numbers (680-681 signs/second). The same behavior was exhibited in other test runs, as well. We attribute this behavior to the fact that our mechanism is a prototype that customizes critical parts of the kernel. Consequently, these parts may not be operating optimally, which requires further research and development.

In summary, the default setup is performing better, as can be expected. 696 signs per second on average, compared to around 682 signs per second on average. However, there is only minimal decrease in performance when using our custom kernel, of about 2%.

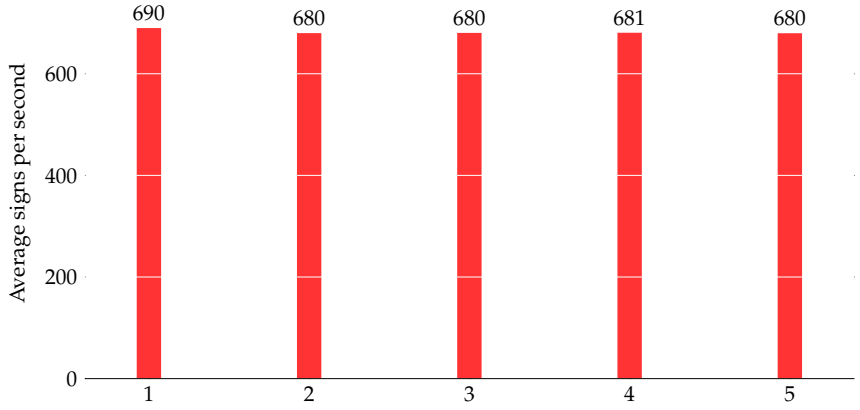
5.5 Training Environment

Aspects of the approach described in this Section, have been incorporated in the training platform developed as part of the THREAT-ARREST EU H2020 research project [1]. There, two training scenarios are offered to the trainees that represent two skill levels: (a) the "Software Exploration" scenario corresponds to the beginner-level scenario, while (b) the second one "Run-time analysis and modification" corresponds to the more advanced level.

[1] https://www.threat-arrest.eu/



(a) Default kernel



(b) Custom kernel

FIGURE 5.9: Signs/second (rounded) of the OpenSSL benchmark of PTS

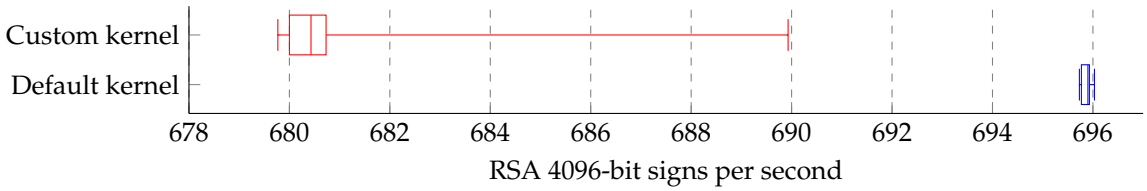


FIGURE 5.10: Performance evaluation of our mechanism using the OpenSSL benchmark of PTS

5.5.1 Software Exploration

In the "Software Exploration" scenario, a user that is trained on using our system must observe the occurrence, the number and the sequence of calls and try to identify if and when an attack happens. In this case, by leveraging the NGINX HTTP server vulnerability as described in Section 5.4 to observe the calls the server issues to external library functions (and the subsequent internal library calls), the trainee can better understand the behavior of the attack and provide a signature of the way it works. Hence, they can recognize

it when it happens and report it.

As the exploitation unfolds, our platform updates two files with the output results of both user- (similarly to the excerpt in Section 4.1.3) and kernel-side (similarly to Figure 5.7) mechanisms. At first, the trainee must monitor the contents of both files. On one hand they are expected to identify which functions are called inside the OpenSSL library (used by NGINX as well) and what these functions perform. On the other hand they must identify not only the occurrence, but a characteristic sequence of calls to executables/libraries as well as the number of consecutive internal calls that correspond to the exploit and produce a trail of it (similarly to Figure 5.8). Next, based on their observations, they must develop a small program in a language of their preference that monitors the files as they are updated by our platform and whenever it intercepts the characteristic signature from the previous steps, it produces an alarm to notify the trainee that an attack was attempted. Consequently, the next time that the trainee will run the scenario, they will be expected to report the alarm to the trainer.

5.5.2 Run-time Analysis and Modification

After the initial introduction to software exploration, the trainee is given the opportunity to continue to the more advanced level. Initially, they are presented with a piece of software and allowed to install it, read available documentation and run some examples of its use. Then the trainee is shown the block diagram of the program and related libraries and is asked to create a call graph showing the control flow paths between elements of the system. Further analysis may be used to identify items of interest (IoI) in the design (e.g., the transfer of an encryption key to a routine, the creation of a network connection, access to a file, etc.). For each identified IoI, a decision needs to be made as to whether it is worth monitoring and if so to which extent (just the invocation of a function, examination of one or more of its arguments, and so on).

After running the program a couple of times under our framework to collect invocation data and confirm some of the initial hypotheses related to the design of the program, the trainee can start the process of active interference in the execution of the program by modifying arguments to functions, or return values etc. Further runs of the program will demonstrate the impact of the modifications, and hopefully shed light to its internal design.

Finally, the trainee must produce a report with their findings, on which they are evaluated at the end of the scenario.

5.6 Summary

In this Chapter, we present the counterpart of the work explained in the previous Chapter, this time on the side of the Linux kernel, which intercepts calls to external libraries/executables (irregardless of them being statically or dynamically linked), as well as internal calls within them, in an efficient

manner. This way, we are able to manipulate the flow of execution, monitor access to executables/libraries and change their functionality when there is suspicion of foul play. We, also, exploit a known NGINX vulnerability and produce a characteristic trail of calls that shows the high-level behavior of the application under attack.

Furthermore, we present a training platform that leverages the two approaches combined: (a) a library wrapper that is inserted between a program and the original library code and (b) a kernel modification that intercepts all calls to libraries/executables. The platform is included in a training environment of the THREAT-ARREST EU H2020 project, which shows that it can be applied to a real-life scenario. As part of this environment, we present two training exercises where we give the trainee the opportunity to explore how a program behaves and modify its execution when under attack. Our approaches are transparent and can be used on binary/legacy applications and existing environments, as well as serve as a complimentary measure of defense alongside already implemented mechanisms.

Part III

THE END

6 Requirements, Evaluation and Future Directions

As is evident from Chapter 2, which contains a small sample of related work, over the last several years there has been a lot of effort to counter Code Injection/Reuse Attacks and the arms race is never-ending. This list of publications contains protection techniques that can be divided into two main categories: probabilistic and deterministic. Probabilistic solutions (see Section 2.2.1) build on randomization or encryption. All other approaches (see Sections 2.2.2 and 2.2.3) offer deterministic protection by implementing a low-level reference monitor [Sch00]. A reference monitor observes the program execution and halts it whenever it is about to violate a given security policy. Traditional reference monitors enforce higher-level policies (such as file system permissions) compared to CFI-like approaches, and are implemented in the kernel (e.g., system calls) [Sze+13].

Reference monitors enforcing lower-level policies, e.g., Control Flow Integrity, can be implemented efficiently in two ways: in hardware or by embedding the reference monitor into the code. For instance, the W$\oplus$X/NX-bit [WX03; DEP04; AA04] mechanism is now enforced by the hardware, as modern processors support both non-writable and non-executable page permissions. Hardware support for a protection mechanism results in negligible overhead [Cop+19]. The alternative to hardware support is adding the reference monitor to the code. Simple reference monitors can be implemented with low overhead (e.g., a shadow stack version costs around 6.5% in performance on average in [TP19]), whereas more sophisticated reference monitors like taint checking [BSB11] or ROP detectors [DSW11] result in overheads that exceed 100% and are unlikely to be deployed in practice [Sze+13].

In this Chapter, we present a systematic way of evaluating our overall approach and comparing it with the state of the art. In Section 6.1, we position our work with respect to the related research contained in Chapter 2, based

on the type of protection that we offer. In Section 6.2, we set the requirements that we believe a security mechanism must abide by, in order to be practical and widely adopted. Section 6.3 presents the comparison of our approach against the state of the art. In Section 6.4, we discuss open issues that future research can tackle. Finally, we summarize the Chapter in Section 6.5.

6.1 Placement

Our approach observes a program's execution and applies a predetermined security policy, for every call that it intercepts. Evidently, based on the previous, it is classified as a reference monitor. Furthermore, taking into account the way that we design and implement our mechanism, we place our approach on the intersection of the two types of deterministic protection, i.e., access/behavior control and execution monitoring. Additionally, with regards to the level at which we enforce the security policies, our approach's position is two-fold: (a) the work in Chapter 4 interjects between an application and the kernel at run-time, while (b) the work detailed in Chapter 5 is embedded into the kernel and observes execution from one level below. Hence, our overall mechanism enforces higher-level protection compared to lower-level approaches (e.g., CFI), similar to a traditional reference monitor.

Taking our placement into consideration in order to evaluate and compare our work against the related work in Section 2, we only focus on work that is most relevant to our own, i.e., control and monitoring mechanisms (Sections 2.2.2 and 2.2.3 respectively). The probabilistic techniques are related to ours only in the sense of defending against C[IR]As and are included for the sake of completeness. However, since we offer no support for randomization or encryption, we consider this category out of scope and do not cover it in the comparison. Furthermore, the rest of the mechanisms (i.e., hardware approaches and shadow stacks) are notions that are implemented in some way in the literature as well as our approach, and are thus compared only as part of the overall respective related work.

6.2 Requirements

When designing a framework for securing a computer system, some objectives need to be reached. We have briefly mentioned the main properties and requirements that our solution meets, in Section 1.2. Here, we discuss them in more detail. These properties determine the practicality of a proposed method, more specifically, whether or not it is suitable for widespread acceptance and adoption.

(R1) Effectiveness: The security mechanism should be able to monitor all communication paths during execution and allow it to continue, only when strict and specific circumstances are met. Compared to other approaches that operate at a lower level, our approach is based on the fact that at some point a high-level call - either benign or malicious -

will be made to a function. So, it intercepts all calls - both local (within a specific region) and remote (between two regions) - before enforcing a security policy in order to decide if the execution flow is allowed to continue or not down the intended path.

(R2) Accuracy is an important attribute, too. The aforementioned circumstances should only lead to benign, nominal execution paths. Furthermore, they should not allow either false positives/alarms (e.g., unnecessary crashes) or false negatives (protection failures) to manifest. The accuracy of an approach can determined by the way the enforced security policy is provided and how comprehensive it is. In our case, the policy can either be generated automatically, e.g., by the system administrator (less comprehensive), or ideally be supplied by the application/library designer (most comprehensive).

(R3) Transparency is of paramount importance for the security layer. Applications must continue to work as originally intended by the developer without interrupting execution, but the security mechanism underneath should deliver a secure run-time environment. In our case, for example, if an untrusted application is hijacked and requests to write in a privileged file (e.g., */etc/passwd* that contains user account information), this would constitute a suspicious action. The mechanism can, then, create a temporary, empty file and modify the arguments of the call so that the application has access only to the dummy file.

(R4) Compatibility: A security system should not require any kind of access or manual modifications to an application's source code, which in most cases is not available. The need of even minimal human intervention or effort makes a solution not only unscalable, but too costly as well, in terms of both time and money. Additionally, the security layer should be compatible with applications distributed in binary form, irregardless of the development language. Our solution has no need for access to source code and is compatible with binary/legacy applications, as well.

(R5) Efficiency: Since there is an extra layer introduced that monitors/alters the execution of an application, there is bound to be some performance overhead associated with it. One fundamental requirement of this layer is to leave as little run-time performance and memory footprint as possible, in order to secure widespread acceptance and adoption. Otherwise, if it impedes or completely prevents the execution of a program, the user will disable the security feature. Some believe the average overhead should be no more than roughly 10% on average, in order the security mechanism gain wide adoption in production environments [Sze+13]. Our mechanism incurs only minimal overhead of about 2% on average.

(R6) Automation: Besides avoiding to impede/prevent the execution of a program, the end user usually requires that any security measure takes little to no extra effort on their part to be implemented. In a different

case, they may choose to ignore the mechanism, if it takes more of their time. Hence, the mechanism should strengthen the system's security ideally automatically, without requiring any kind of intervention on the user's part. Under our scheme, after the initial instrumentation by the system administrator (custom kernel build), the end user need not take any other action.

(R7) Collaboration/coexistence of the new security layer with other existing solutions is also pivotal to its practicality and adoption, as well as the security and availability of the overall system. As already established, absolute security of a computer system cannot be achieved, so there are several practices and measures employed to make sure the system is protected to the highest degree possible. Approaches such as ASLR [PaX01], W⊕X/NX-bit [WX03; DEP04; AA04] and stack canaries [Cow+98; WC03] are enabled by default in most modern systems and protect against a range of vulnerabilities and subsequent attacks. Our solution is orthogonal to these approaches and works well in conjunction with them, so that all together can better protect the underlying system.

6.3 Evaluation and Comparison

In this Section, we present a cumulative list of the several solutions presented in Section 2 and denote if the requirements set in the previous Section are met by each one (Table 6.1). More specifically, as already mentioned (Section 6.1), we focus only on research in Sections 2.2.2 and 2.2.3. Furthermore, we include the performance overhead for each of these solutions, in order to compare them with our own. The last three entries represent our own work.

As we can clearly see, most of the related work falls short in some aspect compared to our framework. That been said, we stress that, comparing our proposed mechanism with existing ones is a challenging task due to the difficulty of reproducing in our testbed the previous results of most of the research. The properties and performance values represent implementations in different environments, different configurations and sometimes measurements with different sets of benchmark programs, so they only provide rough estimates. Additionally, due to the aforementioned reasons, we rely on information reported by the authors/developers of the tools, to populate the table. Whenever information for a requirement cannot be derived from the authors themselves or from our understanding of their work, the corresponding entry is marked as not available (a dash in the Table). Lastly, some approaches are implemented in hardware, so they cannot be directly and completely compared to ours.

Strictly speaking, there is a number of mechanisms directly comparable to our own. Based on some aspects of the implementation (i.e., kernel modifications), most relevant to our work are the approaches authored by Gionta, Enck and Ning [GEN15] and Bakes et al. [Bac+14]. These mechanisms are also relevant to each other, since they both try to apply some kind of *Execute*

and *Read* permissions in the kernel. Furthermore, relevant to our work is the work published by Lin et al. [Lin+21], which (a) divides the executable (*.text*) section of a program into smaller segments (at function granularity) and (b) diverts execution to an added memory block (Trampoline - similar to out *gate*) that computes and validates the correct call target address at runtime. Moreover, comparable to our approach is the work presented in [BR21]. There, the authors isolate each library of a program in its own space and control the interactions between them and the main program. The other approaches either have a hardware/hypervisor component and/or generally try to force execution to adhere to a precomputed graph or to provide execution of multiple variants of the same application.

In [GEN15], the authors calculate a probability of less than 16% for a specific set of applications where an adversary could potentially circumvent their defense, which leads to requirements R1 and R2 being partially fulfilled. On the contrary, our approach traps all execution attempts and since a complete security policy is provided by the application developer, R1 and R2 are both fully met. Furthermore, they rely on fine-grained binary randomization techniques (in order to prevent adversaries from gaining knowledge of code protected in memory), which are not enabled by default in most systems (so R7 is partially fulfilled).

Similarly, in [Bac+14], the authors leave a scenario open where an attack could bypass their mechanism under specific circumstances and execute successfully. Additionally, they assume the use of a more fine-grained variant of ASLR [Dav+13; His+12; War+12] not used by default, in order their approach work as expected. Our framework, however, works seamlessly with the by-default enabled mechanisms without requiring any additional layer, fulfilling requirement R7.

In [Lin+21], Lin et al. identify the instructions to be instrumented inside a *red area* which must conform to a number of conditions. This means they do not replace all fetch instructions, but only the ones located in a *red area* (i.e. they do not cover all execution paths). This course of action leaves a chance open that false positives/negatives will be presented, leading to requirements R1 and R2 being partially fulfilled. Furthermore, the application's binary needs to be statically analyzed and instrumented, which is not fully in-line with requirement R6.

In [BR21], Bauer and Rossow rely on static analysis to produce the PDG which helps them infer which library calls the program will potentially make. Consequently, this leaves a chance that some paths may not be included in the PDG leading to false negatives, thus requirements R1 and R2 are partially fulfilled. Additionally, when implementing their approach they need access to the source code of the main program in order to rewrite all calls to library functions with calls to replacement functions that they have created. Thus, requirement R4 is not fulfilled. The required static analysis and instrumentation also leaves requirement R6 partially fulfilled.

TABLE 6.1: Comparison with other approaches from literature

Study	Requirements							
	R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**Overhead**
Provos [Pro03]	●	◑	○	●	◑	◑	–	0.2% - 31% *
Kim et al. [KP06]	◑	◑	●	◑	○	◑	–	10x
Abadi et al. [Aba+05]	◑	◑	●	●	○	◑	●	<45% *
Kayaalp et al. [Kay+12]	◑	◑	●	●	●	●	●	<7% *
Das et al. [DZL16]	◑	●	●	●	●	●	◑	<1%
Kanuparthi et al. [KRK16]	●	◑	●	◑	●	●	●	4.7%
Niu and Tan [NT14a]	◑	◑	●	◑	●	●	●	4% - 6%
Niu and Tan [NT14b]	●	●	●	◑	◑	●	●	14.6%
Niu and Tan [NT15]	●	●	●	◑	●	●	●	3.2%
Gionta, Enck and Ning [GEN15]	◑	◑	●	●	●	●	◑	1.49%
Backes et al. [Bac+14]	◑	◑	●	●	●	●	◑	2.2%
Zhang et al. [Zha+13]	◑	◑	●	●	●	●	●	0.4%
Habibi et al. [Hab+15b]	●	◑	●	●	●	●	●	2.85%
Kanuparthi et al. [Kan+12]	●	●	●	●	●	●	–	1.66%
Kayaalp et al. [Kay+15]	●	●	●	●	●	●	●	<2%
Tian et al. [Tia+14]	◑	◑	●	◑	◑	●	●	9.5% - 12% *
Crane et al. [Cra+13]	◑	◑	●	◑	–	●	●	–
Liu et al. [Liu+14]	●	●	●	●	◑	●	●	20% *
Volckaert et al. [VCS16]	◑	◑	●	●	●	◑	●	6.37% - 8.94% *
Zeng et al. [ZZL15]	●	●	●	◑	●	◑	●	6.2%
Lin et al. [Lin+21]	◑	◑	●	●	●	◑	●	6.74%
Bauer and Rossow [BR21]	◑	◑	●	○	●	◑	●	0% - 6.5% *
[TP17]	●	●	●	◑	–	●	●	–
[TP19]	●	●	●	●	◑	●	●	1.3% - 11.4%
[TP21a; TP21b; Tsa21]	●	●	●	●	●	●	●	2%

– Not available | ○ Not fulfilled | ◑ Partially fulfilled | ● Fulfilled

* Depending on the system configuration / benchmark / application

6.4 Future Directions

Moving forward, we have identified a number of open issues that point to promising directions in research efforts.

First and foremost, we mention policy generation. In the effort towards

delivering this thesis, we provided our approach to produce security policies, by analyzing a well-known application (Section 4.3). However, we followed this path only to support the development of our prototype. It was not our intention to provide a policy generation mechanism, which is left out of the scope of the thesis. As already mentioned, we expect to be supplied with a correct and complete security policy by the app/library developers. Although research in the area in various domains has been ongoing [FW01; ASCW19; Li+21], we envision additional future research to tackle with the important issue of automatic policy generation, in order to minimize interaction with the developers and enable users to describe the required functionality in an intuitive manner.

Another direction is to port our mechanism to other platforms. Our development efforts were performed on the Intel x86-64 architecture and the Linux OS where specifically we modified the MMU. Future efforts can focus on different platforms (e.g. ARM, Windows, etc.) and possibly create a portable version to provide cross-platform interoperability.

Finally, an important aspect on which to perform research, is optimization. During development, we identified and implemented two important improvements (Section 5.3.9) that optimized the functionality and efficiency of our initial approach. More research can be performed in this respect, in order to further improve our mechanism.

6.5 Summary

In this Chapter, we place our framework among the related work described in Sections 2.2.2 and 2.2.3. We, then, set a number of attributes that a mechanism needs to have in order to be practical and evaluate the state of the art, as well as our approach, based on them. From this comparison - which is not complete due to several differences - it is evident that our approach is among the most concrete solutions to date. Finally, we set a course for future research directions in order to improve our design and implementation.

A PUBLICATIONS

This Appendix lists my publications. The first section includes publications directly related to the main topic of this thesis, while the second one contains other publications. The lists are ordered by the date of appearance in reverse chronological order.

A.1 Related to the Thesis

1. Marinos Tsantekidis and Vassilis Prevelakis, **"Securing Runtime Memory via MMU manipulation"**, in the *15th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE)*, Athens - Greece, 2021 [TP21b]

2. Marinos Tsantekidis and Vassilis Prevelakis, **"MMU-based Access Control for Libraries"**, in the *18th International Conference on Security and Cryptography (SECRYPT)*, Virtual, 2021 [TP21a]

3. Marinos Tsantekidis, **"libC compartmentalization How-To"**, *Technical report at ICS-FORTH*, Crete - Greece, 2021 [Tsa21]

4. Marinos Tsantekidis and Vassilis Prevelakis, **"Software System Exploration using Library Call Analysis"**, in the *2nd Workshop on Model-driven Simulation and Training Environments for Cybersecurity (MSTEC)*, Virtual, 2020 [TP20]

5. Marinos Tsantekidis and Vassilis Prevelakis, **"Efficient Monitoring of Library Call Invocation"**, in the *2nd IEEE International Symposium on Future Cyber Security Technologies (FCST)*, Granada - Spain, 2019 [TP19]

6. Marinos Tsantekidis and Vassilis Prevelakis, **"Report on efforts on compromising Sophos AV"**, *Internal report at TU Braunschweig*, Germany, 2018

7. Marinos Tsantekidis and Vassilis Prevelakis, **"Library-Level Policy Enforcement"**, in the *11th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE)*, Rome - Italy, 2017 [TP17]

A.2 Others

1. Mohammad Hamad, Marinos Tsantekidis and Vassilis Prevelakis, **"Intrusion Response System for Vehicles: Challenges and Vision"**, in *Smart Cities, Green Technologies and Intelligent Transport Systems (SMARTGREENS/VEHITS)*, Communications in Computer and Information Science, vol 1217, pp 321-341, 2019 [HTP21]

2. Marinos Tsantekidis, Mohammad Hamad, Vassilis Prevelakis and Mustafa R. Agha, **"Security for Heterogeneous Systems"**, in *Heterogeneous Computing Architecture - Challenges and Vision*, chapter 10, pp. 221-232. Taylor & Francis Ltd, 1st Edition ed., 2019 [Tsa+19]

3. Mohammad Hamad, Marinos Tsantekidis, and Vassilis Prevelakis, **"Red-Zone: Towards an Intrusion Response Framework for Intra-Vehicle System"**, in the *5th International Conference on Vehicle Technology and Intelligent Transport Systems (VEHITS)*, Crete - Greece, 2019 [HTP19]

List of Figures

List of Tables

List of Abbreviations

AES	Advanced Encryption Standard
API	Application Programming Interface
ASLR	Address Space Layout Randomization
AV	Anti Virus
BB-CFI	Basic Block CFI
BROP	Blind Return Oriented Programming
BR	Branch Regulation
CCC	Controlling Concurrent Change
CET	Control Enforcement Technology
CFC	Control Flow Checker
CFG	Control Flow Graph
CFIN	Control Flow Instruction
CFI	Control Flow Integrity
CIA	Code Injection Attack
COTS	Commercial Off-The-Shelf
CPI	Code Pointer Integrity
CPU	Central Processing Unit
CRA	Code Reuse Attack
DCL	Disjoint Code Layouts
DDoS	Distributed Denial-Of-Service
DEP	Data Execution Prevention
DFG	Deutsche Forschungsgemeinschaft
DMA	Direct Memory Access
DOP	Data Oriented Programming
DRM	Digital Rights Management
DSC	Dynamic Sequence Checker
ECU	Electronic Control Unit
EU	European Union
FFRR	Function Frame Runtime Randomization

FPGate	Function Pointer Gate
FPGA	Field Programmable Gate Array
ILR	Instruction Location Randomization
ISA	Instruction Set Architecture
ISR	Instruction Set Randomization
JIT-ROP	Just-In-Time Return Oriented Programming
JOP	Jump Oriented Programming
KB	Kilobyte
LFSR	Linear Feedback Shift Register
MB	Megabyte
MCFI	Modular Control-Flow Integrity
MMU	Memory Management Unit
NOP	No Operation
NX	No eXecute
OS	Operating System
PBOD	Program Buffer Overflow Defect
PC	Program Counter
PDG	Program Dependence Graph
PFEH	Page Fault Exception Handler
PICFI	Per-Input Control-Flow Integrity
PIE	Position Independent Executable
PTE	Page Table Entry
PTS	Phoronix Test Suite
PT	Page Table
PVM	Process-level Virtual Machine
RFOR	Record Field Order Randomization
ROP	Return Oriented Programming
SAOR	Subroutine Argument Order Randomization
SCRAP	Signature-based CRA Protection
SGX	Software Guard eXtensions
SMAP	Supervisor Memory Access Protection
SMEP	Supervisor Memory Execute Protection
SQL	Structured Query Language
TA	Target Address
TEE	Trusted Execution Environment
TLB	Translation Lookaside Buffer

TUBS	Technical University of Braunschweig
TU	Technical University
UAV	Unmanned Aerial Vehicle
VCFR	Virtual Control Flow Randomization
vDSO	virtual Dynamic Shared Object
VMA	Virtual Memory Area
XOR	eXclusive Or

Bibliography

[AA04] Starr Andersen and Vincent Abella. *Data Execution Prevention*. https://technet.microsoft.com/en-us/library/bb457155.aspx. 2004.

[Aba+05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-Flow Integrity". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. Alexandria, VA, USA: ACM, 2005, pp. 340–353. ISBN: 1-59593-226-7. DOI: 10.1145/1102120.1102165.

[AC19] Hala Assal and Sonia Chiasson. "'Think Secure from the Beginning': A Survey with Software Developers". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. CHI '19. Glasgow, Scotland Uk: Association for Computing Machinery, 2019, 1–13. ISBN: 9781450359702. DOI: 10.1145/3290605.3300519.

[All18] Secure Technology Alliance. *Trusted Execution Environment (TEE) 101: A Primer*. [Retrieved: 02-2022]. 2018. URL: https://www.securetechalliance.org/wp-content/uploads/TEE-101-White-Paper-FINAL2-April-2018.pdf.

[Ana+13] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. "Innovative Technology for CPU Based Attestation and Sealing". In: 2013. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/innovative-technology-for-cpu-based-attestation-and-sealing.html.

[ARM09] ARM. *ARM Security Technology. Building a Secure System using TrustZone Technology*. 2009. URL: https://documentation-service.arm.com/static/5f212796500e883ab8e74531?token=.

[ASCW19] Mohammed Al-Shaboti, Aaron Chen, and Ian Welch. "Automatic Device Selection and Access Policy Generation Based on User Preference for IoT Activity Workflow". In: *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. 2019, pp. 769–774. DOI: 10.1109/TrustCom/BigDataSE.2019.00111.

[Aso+14] N. Asokan, Jan-Erik Ekberg, Kari Kostiainen, Anand Rajan, Carlos Rozas, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. "Mobile Trusted Computing". In: *Proceedings of the IEEE* 102.8 (2014), pp. 1189–1206. DOI: 10.1109/JPROC.2014.2332007.

[Aso+18] N. Asokan, Thomas Nyman, Norrathep Rattanavipanon, Ahmad-Reza Sadeghi, and Gene Tsudik. "ASSURED: Architecture for Secure Software Update of Realistic Embedded Devices". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.11 (2018), pp. 2290–2300. DOI: 10.1109/TCAD.2018.2858422.

[Bac+14] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. "You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, pp. 1342–1353. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660378. URL: http://doi.acm.org/10.1145/2660267.2660378.

[Bar+05] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. "Randomized Instruction Set Emulation". In: *ACM Trans. Inf. Syst. Secur.* 8.1 (2005), 3–40. ISSN: 1094-9224. DOI: 10.1145/1053283.1053286. URL: https://doi.org/10.1145/1053283.1053286.

[BC05] Daniel P. Bovet and Marco Cesati. "Page Fault Exception Handler". In: *Understanding the Linux Kernel, 3rd Edition*. O'Reilly, 2005. Chap. 9.4. ISBN: 0-596-00565-2.

[Bit+14] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. "Hacking Blind". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 227–242. DOI: 10.1109/SP.2014.22.

[BJ21] Jim Bird and Eric Johnson. *A SANS Survey: Rethinking the Sec in DevSecOps: Security as Code*. https://info.veracode.com/sans-survey-rethinking-the-sec-in-devsecops-asset.html. 2021.

[Bla+99] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. *The KeyNote Trust-Management System Version 2*. RFC 2704. http://www.rfc-editor.org/rfc/rfc2704.txt. 1999. URL: http://www.rfc-editor.org/rfc/rfc2704.txt.

[Ble+11] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. "Jump-oriented Programming: A New Class of Code-reuse Attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '11. Hong Kong, China: ACM, 2011, pp. 30–40. ISBN: 978-1-4503-0564-8. DOI: 10.1145/1966913.1966919.

[BN14] Michael Backes and Stefan Nürnberger. "Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014, 433–447. ISBN: 9781931971157.

[Boy+10] Stephen W. Boyd, Gaurav S. Kc, Michael E. Locasto, Angelos D. Keromytis, and Vassilis Prevelakis. "On the General Applicability of Instruction-Set Randomization". In: *IEEE Transactions on Dependable and Secure Computing* 7.3 (2010), pp. 255–270. DOI: 10.1109/TDSC.2008.58.

[BR21] Markus Bauer and Christian Rossow. "Cali: Compiler-Assisted Library Isolation". In: *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. ASIA CCS '21. Virtual Event, Hong Kong: Association for Computing Machinery, 2021, 550–564. ISBN: 9781450382878. DOI: 10.1145/3433210.3453111. URL: https://doi.org/10.1145/3433210.3453111.

[Bra+17] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. "DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization". In: *CoRR* abs/1709.09917 (2017). arXiv: 1709.09917. URL: http://arxiv.org/abs/1709.09917.

[BSB11] Erik Bosman, Asia Slowinska, and Herbert Bos. "Minemu: The World's Fastest Taint Tracker". In: *Recent Advances in Intrusion Detection*. Springer Berlin Heidelberg, 2011, pp. 1–20. ISBN: 978-3-642-23644-0.

[BZP18] Nathan Burow, Xinping Zhang, and Mathias Payer. "Shining Light On Shadow Stacks". In: *CoRR* abs/1811.03165 (2018). arXiv: 1811.03165.

[CCH06] Miguel Castro, Manuel Costa, and Tim Harris. "Securing software by enforcing data-flow integrity". In: *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2006. URL: https://www.microsoft.com/en-us/research/publication/securing-software-by-enforcing-data-flow-integrity/.

[Cer+20] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. "SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems". In: *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, pp. 1416–1432. DOI: 10.1109/SP40000.2020.00061.

[Ces] Juan Cespedes. *ltrace*. URL: https://linux.die.net/man/1/ltrace.

[CH01] Tzi-cker Chiueh and Fu-Hau Hsu. "RAD: a compile-time solution to buffer overflow attacks". In: *Proceedings 21st International Conference on Distributed Computing Systems*. 2001, pp. 409–417. DOI: 10.1109/ICDSC.2001.918971.

[Che+05] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. "Non-Control-Data Attacks Are Realistic Threats". In: *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*. SSYM'05. Baltimore, MD: USENIX Association, 2005, p. 12.

[Che+10] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. "Return-oriented Programming Without Returns". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: ACM, 2010, pp. 559–572. ISBN: 978-1-4503-0245-6. DOI: 10.1145/1866307.1866370. URL: http://doi.acm.org/10.1145/1866307.1866370.

[Che+13] Li-Han Chen, Fu-Hau Hsu, Yanling Hwang, Mu-Chun Su, Wei-Shinn Ku, and Chi-Hsuan Chang. "ARMORY: An Automatic Security Testing Tool for Buffer Overflow Defect Detection". In: *Comput. Electr. Eng.* 39.7 (Oct. 2013), pp. 2233–2242. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2012.07.005.

[Che+14] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Huijie Robert Deng. "Ropecker: A generic and practical approach for defending against rop attacks". In: *In Symposium on Network and Distributed System Security (NDSS)*. 2014.

[Che+16] Yue Chen, Zhi Wang, David Whalley, and Long Lu. "Remix: On-demand Live Randomization". In: *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. CODASPY '16. New Orleans, Louisiana, USA: ACM, 2016, pp. 50–61. ISBN: 978-1-4503-3935-3. DOI: 10.1145/2857705.2857726.

[Com13] Common Vulnerabilities and Exposures. *CVE-2013-2028*. [Retrieved: 02-2022]. 2013. URL: https://www.cvedetails.com/cve/CVE-2013-2028/.

[Con+15] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. "Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, 952–963. ISBN: 9781450338325. DOI: 10.1145/2810103.2813671. URL: https://doi.org/10.1145/2810103.2813671.

[Cop+19] Luigi Coppolino, Salvatore D'Antonio, Giovanni Mazzeo, and Luigi Romano. "A comprehensive survey of hardware-assisted security: From the edge to the cloud". In: *Internet of Things* 6 (2019). [Retrieved: 07-2022], p. 100055. ISSN: 2542-6605. DOI: https://doi.org/10.1016/j.iot.2019.100055. URL: https://www.sciencedirect.com/science/article/pii/S2542660519300101.

[Cor19] Intel Corporation. "Control-flow Enforcement Technology Specification". In: 2019.

[Cow+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks". In: *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. SSYM'98. San Antonio, Texas: USENIX Association, 1998, pp. 5–5.

[CP17] Scott A. Carr and Mathias Payer. "DataShield: Configurable Data Confidentiality and Integrity". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ASIA CCS '17. Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2017, 193–204. ISBN: 9781450349444. DOI: 10.1145/3052973.3052983. URL: https://doi.org/10.1145/3052973.3052983.

[Cra+13] Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. "Booby Trapping Software". In: *Proceedings of the 2013 Workshop on New Security Paradigms Workshop*. NSPW '13. Banff, Alberta, Canada: ACM, 2013, pp. 95–106. ISBN: 978-1-4503-2582-0. DOI: 10.1145/2535813.2535824.

[Cui+21] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. "SmashEx: Smashing SGX Enclaves Using Exceptions". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, 779–793. ISBN: 9781450384544. DOI: 10.1145/3460120.3484821. URL: https://doi.org/10.1145/3460120.3484821.

[CVE16a] CVE-2016-7054. *ChaCha20/Poly1305 heap-buffer-overflow*. [Retrieved: 02-2022]. 2016. URL: https://www.openssl.org/news/secadv/20161110.txt.

[CVE16b] CVE_2016_7054. *ChaCha20/Poly1305 heap-buffer-overflow*. [Retrieved: 02-2022]. 2016. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-7054.

[dan13] danghvu. *For the analysis of CVE-2013-2028*. 2013. URL: https://github.com/danghvu/nginx-1.4.0.

[Dau+15] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation". In: *SIGARCH Comput. Archit. News* 43.1 (2015), 191–206. ISSN: 0163-5964. DOI: 10.1145/2786763.2694386. URL: https://doi.org/10.1145/2786763.2694386.

[Dav+13] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. "Gadge Me If You Can: Secure and Efficient Ad-Hoc Instruction-Level Randomization for X86 and ARM". In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ASIA CCS '13. Hangzhou, China: Association for Computing Machinery, 2013, 299–310. ISBN: 9781450317672. DOI: 10.1145/2484313.2484351. URL: https://doi.org/10.1145/2484313.2484351.

[Dav+14] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection". In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. SEC'14. San Diego, CA: USENIX Association, 2014, 401–416. ISBN: 9781931971157.

[Dav+15] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Snow, and Fabian Monrose. "Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming". In: *Proc. of 22nd Annual Network & Distributed System Security Symposium (NDSS)*. 2015. URL: https://www.internetsociety.org/sites/default/files/05_3_2.pdf.

[DEP04] Microsoft DEP. https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in. 2004.

[Des97] Solar Designer. *Getting around non-executable stack (and fix)*. http://seclists.org/bugtraq/1997/Aug/63. 1997.

[Dha+15] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M. Smith, Thomas F. Knight, Benjamin C. Pierce, and Andre DeHon. "Architectural Support for Software-Defined Metadata Processing". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. Istanbul, Turkey: Association for Computing Machinery, 2015, 487–502. ISBN: 9781450328357. DOI: 10.1145/2694344.2694383. URL: https://doi.org/10.1145/2694344.2694383.

[DHM08] Mark Daniel, Jake Honoroff, and Charlie Miller. "Engineering Heap Overflow Exploits with JavaScript". In: *Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies*. WOOT'08. San Jose, CA: USENIX Association, 2008.

[DMW15] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. "The Performance Cost of Shadow Stacks and Stack Canaries". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2015, pp. 555–566.

[Dof] *Linux kernel source code – _do_fork*. `https://elixir.bootlin.com/linux/v4.16.7/source/kernel/fork.c#L2056`.

[Dop] *Linux kernel source code – __do_page_fault*. `https://elixir.free-electrons.com/linux/v4.16.7/source/arch/x86/mm/fault.c#L1238`.

[DSW11] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. "ROPdefender: a detection tool to defend against return-oriented programming attacks". In: *ASIACCS '11*. 2011.

[DZL16] S. Das, W. Zhang, and Y. Liu. "A Fine-Grained Control Flow Integrity Approach Against Runtime Memory Attacks for Embedded Systems". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24.11 (2016), pp. 3193–3207. ISSN: 1063-8210. DOI: 10.1109/TVLSI.2016.2548561.

[EPAG16] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Jump over ASLR: Attacking Branch Predictors to Bypass ASLR". In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016.

[Err] *Linux kernel source code – Page fault error code bits*. `https://elixir.bootlin.com/linux/v4.16.7/source/arch/x86/include/asm/traps.h#L158`.

[Eva+15] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. "Missing the Point(er): On the Effectiveness of Code Pointer Integrity". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 781–796. DOI: 10.1109/SP.2015.53.

[Fis11] Stephen Fischer. *Supervisor Mode Execution Protection (SMEP)*. `https://web.archive.org/web/20160803075007/https://www.ncsi.com/nsatc11/presentations/wednesday/emerging_technologies/fischer.pdf`. 2011.

[Fit+15] Andreas Fitzek, Florian Achleitner, Johannes Winter, and Daniel Hein. "The ANDIX research OS — ARM TrustZone meets industrial control systems security". In: *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*. 2015, pp. 88–93. DOI: 10.1109/INDIN.2015.7281715.

[Fra12] I. Fratric. *ROPGuard: Runtime prevention of return-oriented programming attacks*. `http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf`. 2012.

[Fra+18] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. "IMIX: In-Process Memory Isolation EXtension". In: *USENIX Security Symposium*. 2018.

[FW01] Zhi (Judy) Fu and S. Felix Wu. "Automatic Generation of IPSec/VPN Security Policies In an Intra-Domain Environment". In: 2001. URL: http://proceedings.utwente.nl/24/.

[GEN15] Jason Gionta, William Enck, and Peng Ning. "HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities". In: *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. CODASPY '15. San Antonio, Texas, USA: ACM, 2015, pp. 325–336. ISBN: 978-1-4503-3191-3. DOI: 10.1145/2699026.2699107.

[Glia] *Building glibc without installing*. [Retrieved: 02-2022]. URL: https://sourceware.org/glibc/wiki/Testing/Builds.

[Glib] *Compile against glibc build tree*. [Retrieved: 02-2022]. URL: https://sourceware.org/glibc/wiki/Testing/Builds.

[Glic] *Glibc source code*. [Retrieved: 02-2022]. URL: https://ftp.gnu.org/gnu/glibc/glibc-2.3.2.tar.gz.

[Glo21] GlobalPlatform. *Trusted Execution Environment (TEE) Committee*. [Retrieved: 02-2022]. 2021. URL: https://globalplatform.org/technical-committees/trusted-execution-environment-tee-committee/.

[Goo18] Google. *CVE-2018-9411 Android Security Bulletin*. https://source.android.com/security/bulletin/2018-07-01. 2018.

[Gru+16] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: Association for Computing Machinery, 2016, 368–379. ISBN: 9781450341394. DOI: 10.1145/2976749.2978356. URL: https://doi.org/10.1145/2976749.2978356.

[Gru+17] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. "Strong and Efficient Cache Side-Channel Protection Using Hardware Transactional Memory". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. Vancouver, BC, Canada: USENIX Association, 2017, 217–233. ISBN: 9781931971409.

[Gua+15] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. "Protecting Private Keys against Memory Disclosure Attacks Using Hardware Transactional Memory". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 3–19. DOI: 10.1109/SP.2015.8.

[Hab+15a] J. Habibi, A. Gupta, S. Carlsony, A. Panicker, and E. Bertino. "MAVR: Code Reuse Stealthy Attacks and Mitigation on Unmanned Aerial Vehicles". In: *2015 IEEE 35th International Conference on Distributed Computing Systems*. 2015, pp. 642–652. DOI: 10.1109/ICDCS.2015.71.

[Hab+15b] Javid Habibi, Ajay Panicker, Aditi Gupta, and Elisa Bertino. "DisARM: Mitigating Buffer Overflow Attacks on Embedded Devices". In: *Network and System Security: 9th International Conference, NSS 2015, New York, NY, USA, November 3-5, 2015, Proceedings*. Cham: Springer International Publishing, 2015, pp. 112–129. ISBN: 978-3-319-25645-0. DOI: 10.1007/978-3-319-25645-0_8. URL: http://dx.doi.org/10.1007/978-3-319-25645-0_8.

[Ham+16] Mohammad Hamad, Johannes Schlatow, Vassilis Prevelakis, and Rolf Ernst. "A communication framework for distributed access control in microkernel-based systems". In: *12th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT16)*. Toulouse, France, 2016, pp. 11–16. URL: http://www.cs.hs-rm.de/~kaiser/events/ospert16/.

[Ham+18] Mohammad Hamad, Zain A. H. Hammadeh, Selma Saidi, Vassilis Prevelakis, and Rolf Ernst. "Prediction of Abnormal Temporal Behavior in Real-Time Systems". In: *The 33rd ACM/SIGAPP Symposium On Applied Computing (SAC 2018)*. 2018.

[Har01] Darren Hart. "Building a Minimal Glibc with Componentization". In: *Linux Journal* (2001). [Retrieved: 02-2022]. URL: https://www.linuxjournal.com/article/5457.

[Hir03] Etoh Hiroaki. "ProPolice: GCC Extension for Protecting Applications from Stack-Smashing Attacks". In: (Jan. 2003).

[His+12] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. "ILR: Where'd my gadgets go?" In: *2012 IEEE Symposium on Security and Privacy*. 2012, pp. 571–585. DOI: 10.1109/SP.2012.39.

[Hoe15] Matthew Hoekstra. *Intel® SGX for Dummies (Intel® SGX Design Objectives)*. https://software.intel.com/en-us/blogs/2013/09/26/protecting-application-secrets-with-intel-sgx. 2015.

[Hom+13] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. "Profile-guided automated software diversity". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2013, pp. 1–11. DOI: 10.1109/CGO.2013.6494997.

[Hom+15] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz. "Large-scale Automated Software Diversity — Program Evolution Redux". In: *IEEE Transactions on Dependable and Secure Computing* PP.99 (2015), pp. 1–1. ISSN: 1545-5971. DOI: 10.1109/TDSC.2015.2433252.

[HP17] Mohammad Hamad and Vassilis Prevelakis. "Secure APIs for Applications in Microkernel-based Systems". In: *3rd International Conference on Information Systems Security and Privacy*. Vol. 1. 2017.

[HTP19] Mohammad Hamad., Marinos Tsantekidis., and Vassilis Prevelakis. "Red-Zone: Towards an Intrusion Response Framework for Intra-vehicle System". In: *Proceedings of the 5th International Conference on Vehicle Technology and Intelligent Transport Systems - VEHITS*, INSTICC. SciTePress, 2019, pp. 148–158. ISBN: 978-989-758-374-2. DOI: 10.5220/0007715201480158.

[HTP21] Mohammad Hamad, Marinos Tsantekidis, and Vassilis Prevelakis. "Intrusion Response System for Vehicles: Challenges and Vision". In: *Smart Cities, Green Technologies and Intelligent Transport Systems*. Springer International Publishing, 2021, pp. 321–341. ISBN: 978-3-030-68028-2.

[Hu+15] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. "Automatic Generation of Data-Oriented Exploits". In: *Proceedings of the 24th USENIX Conference on Security Symposium*. SEC'15. Washington, D.C.: USENIX Association, 2015, 177–192. ISBN: 9781931971232.

[Hu+16] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks". In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016, pp. 969–986. DOI: 10.1109/SP.2016.62.

[Hua+17] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. "VTZ: Virtualizing ARM Trustzone". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. Vancouver, BC, Canada: USENIX Association, 2017, 541–556. ISBN: 9781931971409.

[Inc05] Red Hat Inc. *CVE-2005-2096*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-2096. 2005.

[Inc83] Honeywell Information Systems Inc. *Multics Data Security*. http://multicians.org/multics-data-security.html. 1983.

[Int21] Intel. *11th Generation Intel Core Processor Datasheet*. 2021, p. 65. URL: https://cdrdv2.intel.com/v1/dl/getContent/634648.

[Int22] Intel. *12th Generation Intel Core Processor Datasheet*. 2022, p. 51. URL: https://cdrdv2.intel.com/v1/dl/getContent/655258.

[Jel+21] Christopher Jelesnianski, Jinwoo Yom, Changwoo Min, and Yeongjin Jang. "Securely Sharing Randomized Code That Flies". In: *Digital Threats* (2021). Just Accepted. ISSN: 2692-1626. DOI: 10.1145/3474558. URL: https://doi.org/10.1145/3474558.

[Kan+12] A. K. Kanuparthi, R. Karri, G. Ormazabal, and S. K. Addepalli. "A high-performance, low-overhead microarchitecture for secure program execution". In: *2012 IEEE 30th International Conference on Computer Design (ICCD)*. 2012, pp. 102–107. DOI: 10.1109/ICCD.2012.6378624.

[Kar15] Hubert Kario. *TLS test suite and fuzzer*. 2015. URL: https://github.com/tomato42/tlsfuzzer.

[Kay+12] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. "Branch Regulation: Low-overhead protection from code reuse attacks". In: *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*. 2012, pp. 94–105. DOI: 10.1109/ISCA.2012.6237009.

[Kay+15] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. A. Ghazaleh. "Signature-Based Protection from Code Reuse Attacks". In: *IEEE Transactions on Computers* 64.2 (2015), pp. 533–546. ISSN: 0018-9340. DOI: 10.1109/TC.2013.230.

[Ker] Michael Kerrisk. http://man7.org/linux/man-pages/man7/vdso.7.html.

[Ker08] Michael Kerrisk. *shm_overview(7) — Linux manual page*. [Retrieved: 02-2022]. 2008. URL: https://www.man7.org/linux/man-pages/man7/shm_overview.7.html.

[Kim+15] S. H. Kim, L. Xu, Z. Liu, Z. Lin, W. W. Ro, and W. Shi. "Enhancing Software Dependability and Security with Hardware Supported Instruction Address Space Randomization". In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2015, pp. 251–262. DOI: 10.1109/DSN.2015.48.

[KK14] K. S. Kumar and N. R. Kisore. "Protection against Buffer Overflow Attacks through Runtime Memory Layout Randomization". In: *2014 International Conference on Information Technology*. 2014, pp. 184–189. DOI: 10.1109/ICIT.2014.57.

[KKP03] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. "Countering Code-Injection Attacks with Instruction-Set Randomization". In: *Proceedings of the 10th ACM Conference on Computer and Communications Security*. CCS '03. Washington D.C., USA: Association for Computing Machinery, 2003, 272–280. ISBN: 1581137389. DOI: 10.1145/948109.948146. URL: https://doi.org/10.1145/948109.948146.

[Koc+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy*. 2019.

[Kon+17] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. "No Need to Hide: Protecting Safe Regions on Commodity Hardware". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: Association for Computing Machinery, 2017, 437–452. ISBN: 9781450349383. DOI: 10.1145/3064176.3064217. URL: https://doi.org/10.1145/3064176.3064217.

[Kon17] Andrey Konovalov. *Exploiting the Linux kernel via packet sockets*. https://googleprojectzero.blogspot.com/2017/05/exploiting-linux-kernel-via-packet.html. 2017.

[KP06] Jason W. Kim and Vassilis Prevelakis. "Base Line Performance Measurements of Access Controls for Libraries and Modules". In: *Proceedings of the 20th International Conference on Parallel and Distributed Processing*. IPDPS'06. Rhodes Island, Greece: IEEE Computer Society, 2006, pp. 356–356. ISBN: 1-4244-0054-6.

[KRK16] A. Kanuparthi, J. Rajendran, and R. Karri. "Controlling your control flow graph". In: *2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2016, pp. 43–48. DOI: 10.1109/HST.2016.7495554.

[KT13] M. Kanter and S. Taylor. "Attack Mitigation through Diversity". In: *MILCOM 2013 - 2013 IEEE Military Communications Conference*. 2013, pp. 1410–1415. DOI: 10.1109/MILCOM.2013.239.

[Kuv+17] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. "SGXBOUNDS: Memory Safety for Shielded Execution". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: Association for Computing Machinery, 2017, 205–221. ISBN: 9781450349383. DOI: 10.1145/3064176.3064192. URL: https://doi.org/10.1145/3064176.3064192.

[Kuz+14] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. "Code-Pointer Integrity". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, 147–163. ISBN: 9781931971164.

[Lan96] Gerard Le Lann. *The Ariane 5 Flight 501 Failure - A Case Study in System Engineering for Computing Systems*. Research Report RR-3079. Projet REFLECS. INRIA, 1996. URL: https://hal.inria.fr/inria-00073613.

[Lar+14] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. "SoK: Automated Software Diversity". In: *2014 IEEE Symposium on Security and Privacy*. 2014, pp. 276–291. DOI: 10.1109/SP.2014.25.

[Lar19] Michael Larabel. *Linux Kernel To Better Fend Off Exploits That Disable SMAP/SMEP/UMIP Protections*. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Protect-Special-CR4-Bits. 2019.

[Lee+17] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. "Hacking in Darkness: Return-Oriented Programming against Secure Enclaves". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. Vancouver, BC, Canada: USENIX Association, 2017, 523–539. ISBN: 9781931971409.

[Li+21] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, and Jim Hao Chen. "Automatic Policy Generation for Inter-Service Access Control of Microservices". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3971–3988. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-xing.

[Lim21] Linaro Limited. *Open Portable Trusted Execution Environment*. [Retrieved: 02-2022]. 2021. URL: https://www.op-tee.org/.

[Lin+21] Kunli Lin, Haojun Xia, Kun Zhang, and Bibo Tu. "AddrArmor: An Address-based Runtime Code-reuse Attack Mitigation for Shared Objects at the Binary-level". In: *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. 2021, pp. 117–124. DOI: 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00029.

[Lip] Lipika. *Think like a hacker to protect your organization from security breaches – Key takeaways from Nuix Black Market Report 2018*. [Retrieved: 02-2022]. URL: https://www.dailyhostnews.com/key-takeaways-from-nuix-black-market-report-2018.

[Liu+14] Z. Liu, W. Shi, S. Xu, and Z. Lin. "Programmable decoder and shadow threads: Tolerate remote code injection exploits with diversified redundancy". In: *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2014, pp. 1–6. DOI: 10.7873/DATE.2014.064.

[LRL15] Samuel Laurén, Sampsa Rauti, and Ville Leppänen. "Diversification of System Calls in Linux Kernel". In: *Proceedings of the 16th International Conference on Computer Systems and Technologies*. CompSysTech '15. Dublin, Ireland: ACM, 2015, pp. 284–291. ISBN: 978-1-4503-3357-3. DOI: 10.1145/2812428.2812447.

[LSL15] Sixing Lu, Minjun Seo, and Roman Lysecky. "Timing-based Anomaly Detection in Embedded Systems". In: *20th Asia and South Pacific Design Automation Conference, ASP-DAC 2015*. Jan. 2015. DOI: 10.1109/ASPDAC.2015.7059110.

[Ltr] *ltrace*. URL: http://www.ltrace.org/.

[Lu+15] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. "ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: Association for Computing Machinery, 2015, 280–291. ISBN: 9781450338325. DOI: 10.1145/2810103.2813694. URL: https://doi.org/10.1145/2810103.2813694.

[McK+13] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. "Innovative Instructions and Software Model for Isolated Execution". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '13. Tel-Aviv, Israel: Association for Computing Machinery, 2013. ISBN: 9781450321181. DOI: 10.1145/2487726.2488368. URL: https://doi.org/10.1145/2487726.2488368.

[McK+16] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. "Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave". In: *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. HASP 2016. Seoul, Republic of Korea: Association for Computing Machinery, 2016. ISBN: 9781450347693. DOI: 10.1145/2948618.2954331. URL: https://doi.org/10.1145/2948618.2954331.

[Mer13] Mert Sarica. *Nginx 1.3.9 < 1.4.0 - Denial of Service (PoC)*. 2013. URL: https://www.mertsarica.com/nginx-dos-istismar-kodu/.

[MGR14] Hector Marco-Gisbert and Ismael Ripoll. "On the Effectiveness of Full-ASLR on 64-bit Linux". In: *In-depth Security Conference, DeepSec*. Vienna, Austria, 2014. URL: http://cybersecurity.upv.es/attacks/offset2lib/offset2lib-paper.pdf.

[MGRR19] Hector Marco-Gisbert and Ismael Ripoll Ripoll. "Address Space Layout Randomization Next Generation". In: *Applied Sciences* 9.14 (2019). ISSN: 2076-3417. DOI: 10.3390/app9142928. URL: https://www.mdpi.com/2076-3417/9/14/2928.

[Mhs13] Greg MacManus, hal, and saelo. *Nginx HTTP Server 1.3.9-1.4.0 Chunked Encoding Stack Buffer Overflow*. 2013. URL: https://www.rapid7.com/db/modules/exploit/linux/http/nginx_chunked_size.

[Mms] *Linux kernel source code – mm_struct*. https://elixir.free-electrons.com/linux/v4.16.7/source/include/linux/mm_types.h#L350.

[MRD18] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. "MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation". In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis. Cham: Springer International Publishing, 2018, pp. 359–379. ISBN: 978-3-030-00470-5.

[MSW17] Patrick Morrison, Benjamin H. Smith, and Laurie Williams. "Surveying Security Practice Adherence in Software Development". In: *Proceedings of the Hot Topics in Science of Security: Symposium and Bootcamp*. HoTSoS. Hanover, MD, USA: Association for Computing Machinery, 2017, 85–94. ISBN: 9781450352741. DOI: 10.1145/3055305.3055312.

[Mula] *Multics*. http://www.cse.psu.edu/~trj1/cse443-s12/docs/ch3.pdf. 2012.

[Mulb] *Multics Rings*. http://www.cs.unc.edu/~dewan/242/f96/notes/prot/node11.html. 1996.

[Mul15] David Mulnix. *Supervisor Mode Access Protection (SMAP)*. https://software.intel.com/en-us/articles/intel-xeon-processor-d-product-family-technical-overview#_Toc419802869. 2015.

[Mur+20] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Frank Piessens, and Daniel Gruss. "Plundervolt: How a Little Bit of Undervolting Can Create a Lot of Trouble". In: *IEEE Security Privacy* 18.5 (2020), pp. 28–37. DOI: 10.1109/MSEC.2020.2990495.

[NT14a] Ben Niu and Gang Tan. "Modular Control-flow Integrity". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom: ACM, 2014, pp. 577–587. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594295.

[NT14b] Ben Niu and Gang Tan. "RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: ACM, 2014, pp. 1317–1328. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660281.

[NT15] Ben Niu and Gang Tan. "Per-Input Control-Flow Integrity". In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 914–926. ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813644. URL: http://doi.acm.org/10.1145/2810103.2813644.

[O'K+15] Dan O'Keeffe, Divya Muthukumaran, Pierre-Louis Aublin, Florian Kelbert, Christian Priebe, Josh Lind, Huanzhou Zhu, and Peter Pietzuch. *Spectre attack against SGX enclave*. `https://github.com/lsds/spectre-attack-sgx`. 2015.

[OWA] OWASP. *OWASP Top 10 - 2021*. `https://owasp.org/Top10/`.

[Pap+13] Antonis Papadogiannakis, Laertis Loutsis, Vassilis Papaefstathiou, and Sotiris Ioannidis. "ASIST: Architectural Support for Instruction Set Randomization". In: *ACM-CCS*. 2013, pp. 981–992.

[PaX01] Team PaX. *Address Space Layout Randomization*. `https://pax.grsecurity.net/docs/aslr.txt`. 2001.

[PK10] Georgios Portokalidis and Angelos D. Keromytis. "Fast and Practical Instruction-Set Randomization for Commodity Systems". In: *ACSAC*. 2010.

[Pog] Chris Pogue. *The Black Report 2018*. [Retrieved: 02-2022]. URL: `https://www.nuix.com/white-papers/black-report-2018`.

[Pom+17] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. "kRX: Comprehensive Kernel Protection against Just-In-Time Code Reuse". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: Association for Computing Machinery, 2017, 420–436. ISBN: 9781450349383. DOI: `10.1145/3064176.3064216`. URL: `https://doi.org/10.1145/3064176.3064216`.

[Pom20] Marios Pomonis. "Preventing Code Reuse Attacks On Modern Operating Systems". In: *Ph.D. Thesis*. 2020.

[PPK13] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing". In: *Proceedings of the 22Nd USENIX Conference on Security*. SEC'13. Washington, D.C.: USENIX Association, 2013, pp. 447–462. ISBN: 978-1-931971-03-4.

[Pre17] Vassilis Prevelakis. *Use of HTTP protocol by the TU-BS Sophos Repository*. Tech. rep. TU Braunschweig, 2017.

[Pro03] Niels Provos. "Improving Host Security with System Call Policies". In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. SSYM'03. Washington, DC: USENIX Association, 2003, pp. 18–18.

[Pts] *Phoronix Test Suite*. `https://www.phoronix-test-suite.com`.

[Qiu+19] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. "VoltJockey: Breaching TrustZone by Software-Controlled Voltage Manipulation over Multi-Core Frequencies". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, 195–209. ISBN: 9781450367479. DOI: 10.1145/3319535.3354201. URL: https://doi.org/10.1145/3319535.3354201.

[Roe+12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. "Return-Oriented Programming: Systems, Languages, and Applications". In: *ACM Trans. Inf. Syst. Secur.* 15.1 (Mar. 2012), 2:1–2:34. ISSN: 1094-9224. DOI: 10.1145/2133375.2133377.

[SBYZ21] R. Stajnrod, R. Ben Yehuda, and N.J. Zaidenberg. "Attacking TrustZone on devices lacking memory protection". In: *Journal of Computer Virology and Hacking Techniques* (2021). URL: https://doi.org/10.1007/s11416-021-00413-y.

[Sch00] Fred B. Schneider. "Enforceable Security Policies". In: *ACM Trans. Inf. Syst. Secur.* 3.1 (2000), 30–50. ISSN: 1094-9224. DOI: 10.1145/353323.353382. URL: https://doi.org/10.1145/353323.353382.

[Sch+17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Malware guard extension: Using SGX to conceal cache attacks". In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 14th International Conference, DIMVA 2017*. Vol. 10327 LNCS. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer-Verlag Italia, 2017, pp. 3–24. ISBN: 9783319608754. DOI: 10.1007/978-3-319-60876-1_1.

[Seh+10] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. "Adapting Software Fault Isolation to Contemporary CPU Architectures". In: *Proceedings of the 19th USENIX Conference on Security*. USENIX Security'10. Washington, DC: USENIX Association, 2010, p. 1. ISBN: 8887666655554.

[Sha+04] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. "On the Effectiveness of Address-space Randomization". In: *Proceedings of the 11th ACM Conference on Computer and Communications Security*. CCS '04. Washington DC, USA: ACM, 2004, pp. 298–307. ISBN: 1-58113-961-6. DOI: 10.1145/1030083.1030124. URL: http://doi.acm.org/10.1145/1030083.1030124.

[Sha07] 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. Alexandria, Virginia, USA: ACM, 2007, pp. 552–561. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315313.

[Shu16] Kirill A. Shutemov. *5-level paging*. http://lkml.iu.edu/hypermail/linux/kernel/1612.1/00383.html. 2016.

[Sno+13] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 574–588. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.45. URL: http://dx.doi.org/10.1109/SP.2013.45.

[Son+16] *HDFI: Hardware-Assisted Data-Fow Isolation*. San Jose, CA, 2016.

[sor13] sorbo. *Nginx 1.4.0 (Generic Linux x64) - Remote Overflow*. 2013. URL: https://www.exploit-db.com/exploits/32277.

[SS72] Michael D. Schroeder and Jerome H. Saltzer. *A Hardware Architecture for Implementing Protection Rings*. ftp://ftp.stratus.com/vos/multics/tvv/protection.html. 1972.

[SWG19] Michael Schwarz, Samuel Weiser, and Daniel Gruß. "Practical Enclave Malware with Intel SGX". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Lecture Notes in Computer Science. Springer International, 2019, pp. 177–196. ISBN: 978-3-030-22037-2. DOI: 10.1007/978-3-030-22038-9_9.

[SXS13] D. M. Stanley, D. Xu, and E. H. Spafford. "Improved kernel security through memory layout randomization". In: *2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC)*. 2013, pp. 1–10. DOI: 10.1109/PCCC.2013.6742768.

[Sze+13] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[Tas] *Linux kernel source code – task_struct*. https://elixir.bootlin.com/linux/v4.16.7/source/include/linux/sched.h#L524.

[Tea22] The 2022 CWE Top 25 Team. *The 2022 CWE Top 25*. [Retrieved: 08-2022]. 2022. URL: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

[Tia+14] Donghai Tian, Xi Xiong, Changzhen Hu, and Peng Liu. "Defeating buffer overflow attacks via virtualization". In: *Computers & Electrical Engineering* 40.6 (2014), pp. 1940 –1950. ISSN: 0045-7906. DOI: http://dx.doi.org/10.1016/j.compeleceng.2013.11.032. URL: http://www.sciencedirect.com/science/article/pii/S0045790613003145.

[TP17] Marinos Tsantekidis and Vassilis Prevelakis. "Library-Level Policy Enforcement". In: *The Eleventh International Conference on Emerging Security Information, Systems and Technologies (SECURWARE)*. [Retrieved: 02-2022]. 2017. URL: http://www.thinkmind.org/index.php?view=article&articleid=securware_2017_2_20_30034.

[TP19] Marinos Tsantekidis and Vassilis Prevelakis. "Efficient Monitoring of Library Call Invocation". In: *Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*. [Retrieved: 02-2022]. Granada, Spain, 2019. ISBN: 978-1-7281-2949-5. DOI: 10.1109/IOTSMS48152.2019.8939203. URL: https://doi.org/10.1109/IOTSMS48152.2019.8939203.

[TP20] Marinos Tsantekidis and Vassilis Prevelakis. "Software System Exploration Using Library Call Analysis". In: *Model-driven Simulation and Training Environments for Cybersecurity*. Springer International Publishing, 2020, pp. 125–139. ISBN: 978-3-030-62433-0.

[TP21a] Marinos Tsantekidis and Vassilis Prevelakis. "MMU-based Access Control for Libraries". In: *Proceedings of the 18th International Conference on Security and Cryptography - SECRYPT*. [Retrieved: 02-2022]. INSTICC. SciTePress, 2021, pp. 686–691. ISBN: 978-989-758-524-1. DOI: 10.5220/0010536706860691. URL: https://www.scitepress.org/Link.aspx?doi=10.5220/0010536706860691.

[TP21b] Marinos Tsantekidis and Vassilis Prevelakis. "Securing Runtime Memory via MMU Manipulation". In: *The Fifteenth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE)*. [Retrieved: 02-2022]. 2021, pp. 76–81. ISBN: 978-1-61208-919-5. URL: https://www.thinkmind.org/index.php?view=article&articleid=securware_2021_1_120_30059.

[Tsa18] Marinos Tsantekidis. *Report on efforts on compromising Sophos AV*. Tech. rep. TU Braunschweig, 2018.

[Tsa+19] Marinos Tsantekidis, Mohammad Hamad, Vassilis Prevelakis, and Mustafa R. Agha. "Security for Heterogeneous Systems". In: *Heterogeneous Computing Architecture - Challenges and Vision*. Taylor & Francis Ltd, 2019, pp. 221–232. URL: https://doi.org/10.1201/9780429399602.

[Tsa21] Marinos Tsantekidis. *libC compartmentalization How-To*. Tech. rep. ICS-FORTH, 2021.

[TSS17] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management". In: *Proceedings of the 26th USENIX Conference on Security Symposium*. SEC'17. Vancouver, BC, Canada: USENIX Association, 2017, 1057–1074. ISBN: 9781931971409.

[VB+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, 2018.

[VCS16] S. Volckaert, B. Coppens, and B. De Sutter. "Cloning Your Gadgets: Complete ROP Attack Immunity with Multi-Variant Execution". In: *IEEE Transactions on Dependable and Secure Computing* 13.4 (2016), pp. 437–450. ISSN: 1545-5971. DOI: 10.1109/TDSC.2015.2411254.

[Ven00] Vendicator. "Stack Shield". In: (2000).

[Ven+16] Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M. Tullsen. "HIPStR: Heterogeneous-ISA Program State Relocation". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 727–741. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872408.

[Ven+19] Elaine Venson, Reem Alfayez, Marília M. F. Gomes, Rejane M. C. Figueiredo, and Barry Boehm. "The Impact of Software Security Practices on Development Effort: An Initial Survey". In: *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2019, pp. 1–12. DOI: 10.1109/ESEM.2019.8870153.

[Vij21] Jai Vijayan. *Memory Corruption Issues Lead 2021 CWE Top 25*. [Retrieved: 08-2022]. 2021. URL: https://www.darkreading.com/application-security/memory-corruption-issues-lead-2021-cwe-top-25.

[Vma] *Linux kernel source code – vm_area_struct*. https://elixir.free-electrons.com/linux/v4.16.7/source/include/linux/mm_types.h#L274.

[w0013] w00d. *Analysis of nginx 1.3.9/1.4.0 stack buffer overflow and x64 exploitation (CVE-2013-2028)*. 2013. URL: https://www.vnsecurity.net/research/2013/05/21/analysis-of-nginx-cve-2013-2028.html.

[Wah+93a] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient Software-Based Fault Isolation". In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. SOSP '93. Asheville, North Carolina, USA: Association for Computing Machinery, 1993, 203–216. ISBN: 0897916328. DOI: 10.1145/168619.168635. URL: https://doi.org/10.1145/168619.168635.

[Wah+93b] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. "Efficient Software-Based Fault Isolation". In: *SIGOPS Oper. Syst. Rev.* 27.5 (Dec. 1993), 203–216. ISSN: 0163-5980. DOI: 10.1145/173668.168635. URL: https://doi.org/10.1145/173668.168635.

[War+12] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. "Binary Stirring: Self-Randomizing Instruction Addresses of Legacy X86 Binary Code". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, 157–168. ISBN: 9781450316514. DOI: 10.1145/2382196.2382216. URL: https://doi.org/10.1145/2382196.2382216.

[Wat07] Robert N. M. Watson. "Exploiting Concurrency Vulnerabilities in System Call Wrappers". In: *Proceedings of the First USENIX Workshop on Offensive Technologies*. WOOT '07. Boston, MA: USENIX Association, 2007, 2:1–2:8.

[WC03] Perry Wagle and Crispin Cowan. "Stackguard: Simple stack smash protection for GCC". In: *Proc. of the GCC Developers Summit*. 2003, pp. 243–255.

[Wei+19] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. "SGXJail: Defeating Enclave Malware via Confinement". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Sept. 2019, pp. 353–366. ISBN: 978-1-939133-07-6.

[WX03] OpenBSD i386 WX. http://marc.info/?l=openbsd-misc&m=105056000801065. 2003.

[YSX16] Ruan Yefeng, Kalyanasundaram Sivapriya, and Zou Xukai. "Survey of return-oriented programming defense mechanisms". In: *Security and Communication Networks* 9.10 (2016), pp. 1247–1265. DOI: 10.1002/sec.1406. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.1406. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1406.

[ZB18] Tamir Zahavi-Brunner. *CVE-2018-9411: New critical vulnerability in multiple high-privileged Android services*. `https://blog.zimperium.com/cve-2018-9411-new-critical-vulnerability-multiple-high-privileged-android-services/`. 2018.

[Zha+13] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Stephen McCamant, and Laszlo Szekeres. "Protecting Function Pointers in Binary". In: *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*. ASIA CCS '13. Hangzhou, China: ACM, 2013, pp. 487–492. ISBN: 978-1-4503-1767-2. DOI: `10.1145/2484313.2484376`. URL: `http://doi.acm.org/10.1145/2484313.2484376`.

[Zha+21] Chaochao Zhang, Amro Awad, Rui Hou, and Mazen Alwadi. "Punchcard: A Practical Red-Zone Based Scheme for Low-Overhead Heap Protection". In: *2021 IEEE 23rd Int Conf on High Performance Computing & Communications; 7th Int Conf on Data Science & Systems; 19th Int Conf on Smart City; 7th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys)*. 2021, pp. 405–414. DOI: `10.1109/HPCC-DSS-SmartCity-DependSys53884.2021.00078`.

[ZZL15] Q. Zeng, M. Zhao, and P. Liu. "HeapTherapy: An Efficient End-to-End Solution against Heap Buffer Overflows". In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2015, pp. 485–496. DOI: `10.1109/DSN.2015.54`.