Automatic Exploit Generation: A modular approach for vulnerability validation

Alexandru Pena
alexandru.pena@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November, 2022

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, this 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. [14]). More recent research (Gadient et al. [9]) 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 proofs 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 [13]. 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 AVDs 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

1. Introduction

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 a lot of 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 [2] 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) [8] 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 [13]. 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 the vulnerability detection can be considered the core component of such systems, exploit generation can add a lot of value to the system. 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 [5], 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.

2. Background

We will use this Section to 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. 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. An attacker could possibly exploit a buffer overflow bug to gain control over a target machine, by making the application execute what he desires.

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.

2.2. 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. 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.

2.3. 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.

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.

2.4. 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.

2.5. Buffer Over-read

his 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.

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 [1], a security bug in the OpenSSL Cryptography library which allowed an attacker to obtain sensitive information through specially crafted packets.

2.6. Ret-2-Libc and Data Execution Prevention (DEP)

DEP [6], 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.

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.

2.7. Return Oriented Programming (ROP)

Return Oriented Programming (ROP) [12] 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 [7], 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.

3. Implementation

In this chapter we present the different components of the system, difficulties and challenges faced with the implementation. This work was implemented in around 2100 lines of python code.

One of the first steps when AVD starts is deciding how the AVD will try to exploit the target, for this purpose the module Exploit Strategy 3.8 will analyze what 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, the engine will execute the target instruction by instruction. The DSE engine will sometimes resource to the use of Summarization 3.13, this is meant to accelerate the analysis process of binaries.

During the analysis of the target, sometimes libc functions will be called, those are handled by the Libc Summaries. We have implemented in some libc summaries 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 3.2 system.

When the DSE engine detects that a safety policy is violated it will query the exploit strategy decided and verify if we have met all requirements to generate an exploit. If we have, then the Exploit Generation 3.9 module will be executed to attempt the generation of an exploit.

In the next chapters we start by introducing our

attacker and then discuss different forms of Memory Disclosures (3.2), this being how the system detects memory leaks using different vulnerabilities like format string. Then we discuss the exploit generation (3.9) its strategies and complete with Summarization (3.13).

3.1. Extending AVD's Attacker Model

In previous work [13], 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`. 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 current enabled 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, and 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 we require 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: ROP chains, and 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 protections 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 also 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 also 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.

3.2. Memory Disclosure

In order to to bypass protections ASLR, PIE, Stack canaries, and other 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 3.5), *buffer overreads* (Section 3.7), and *output functions* (Section 3.4) that print unsafe information such as addresses (although this last method is usually associated with unsafe programming by the developer).

In the following Section 3.3, we will present the newly implemented strategy to automatically verify the resulting contents of applying the strategies further discussed.

3.3. 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.

3.4. 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. Having knowledge of that, we can calculate the address of all the others loaded libc functions.

3.5. Format Strings

The detection of the format string vulnerability is implemented in the libc functions summaries' of the AVD, `printf`. The first thing we need to do is check if we have any user controlled characters. To this purpose we have an algorithm that will count how many consecutive symbolic variables we control.

The reason for that, is because we need to have at least two consecutive user controlled characters, like `"%x"`, in order to leak a value from the stack. With a string like `"<sym>H<sym>E<sym>L<sym>L<sym>O"`, we can't really do anything since we can't solve it to any payload.

To decide with what format string payload to solve these variables, we will run an algorithm that needs to verify the following things: the enabled protections of the binary; the strategy chosen by the exploit strategy module; and the size of the user controlled character.

It's important to know what protections are enabled since they limit our capacity to use more complex payloads.

The dependence of the strategy exists because different exploitation strategies require different information, some may not need an ASLR leak and vice versa.

After choosing the type of payload, we solve the symbolic variables with this payload. The format string is then passed to the `printf` parser and the algorithm discussed in Section 3.3 will execute to detect if we managed to obtain a leak.

3.6. 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).

3.7. 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 this 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 work. 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.

For example, 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.

3.8. Exploit Strategy

The exploit strategy module decides what exploit strategy to try based on the protections that are enabled on the binary and system, it tracks the status of said exploit strategy and starts the exploit generation when all conditions are met.

Each implemented exploit strategy has knowledge about the information it requires depending on the enabled protections. A strategy is decided at the start of the execution and tracks the status of the vulnerability discovery process, when all requirements are the AVD will stop searching for vulnerabilities and exploit generation will start.

This module has the ROP and Ret-2-Libc techniques implemented, though is very flexible and can be extended with more strategies to increase the chances of being able to generate a working exploit.

3.9. Exploit Generation

The exploit generation module is responsible for using all 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 requirements and an overflow/write out of bounds vulnerability is detected by the memory safety policies.

The first thing the exploit generation algorithm needs to do, is to distinguish between the leak trace and 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 execu-

tion 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 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 ret-2-libc technique will be briefly explained in Section 3.10, and the exploit generation function corresponding to the `ROP` strategy will be described in greater detail in Section 3.11.

3.10. 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.

3.11. 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.

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 3.2.

After this decision process is complete, we use the tool ROPGadget [10] to obtain the rop chain payload: `ROPgadget --binary ./target --ropchain`, where the target is either the binary or the libc.

3.12. 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.

To reach the desired state faster we present in Section 3.13 a strategy that attempts at relieving the problem created by path explosion.

3.13. 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.

With the goal to improve the execution times in these cases, we implement a Summarization technique. Summarization will try to merge the different possible states generated by a loop into a single state and with such avoid the execution of code outside of the loops multiple times for each possible path.

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.

3.14. 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 [11]. 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.

After this process completes, it will save the information of the loops identified in a file for caching.

3.15. 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 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. This process will be described in Section 3.16.

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 3.18. 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.

3.16. 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.

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, this value needs to be adapted according to the target code and the user may have to test different combinations.

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.

3.17. 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, this is done for later on to merge all states into a single expression.

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.

3.18. 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 or more) 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.

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.

4. 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 attempt the exploitation of 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), amount of exploits generated that are able to obtain a shell, leak and control-flow hijacks.

4.1. 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, though the system architecture permits for more implementations. If the target binary follows the attacker model as described in section 3.1, 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.

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.

4.2. 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).

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_game_of_chance` (Stack); `03_format1_aslr`, `04_format2_aslr` and `lab6B` (ASLR) for which we do not have libc functions summaries' yet implemented. The challenges `09_format_picoctf` and `10_format_rpisec` 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.

4.3. Summarization

In terms of the summarization technique, the results depend on the target code. In some cases we have obtained significant execution time improvements.

In a adapted version of the code presented by [3], with only 10 iterations instead of 100, 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)`) outside of the loop 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.

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`.

Our strategy however falls short by a large margin when compared to MergePoint and their Static Symbolic Execution technique it.

4.4. 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. 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

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`.

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

5. Conclusions

5.1. 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 exploitation technique in order to exploit binaries, although the architecture implementation allows with few changes to extend it with more techniques.

There were 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 very small binaries when the conditions are right with modern protections like ASLR enabled or PIE.

A Summarization algorithm was implemented, achieving significantly lower execution times for some code cases discussed in 4, 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 enriching the AVD vulnerability detection system allowing for more vulnerabilities to be detected and faster which in turn permitting the sys-

tem to generate more exploits.

5.2. Future Work

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

The AVD also requires for more libc functions summaries to be implemented, if we thrive to achieve a more complete analysis of binaries.

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, many improvements can be made 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 no doubtly hold better results if implemented a strategy similar to MergePoint [4] 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.

Acknowledgements

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 it's 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.

References

[1] Cve-2014-0160.

[2] Cve-2021-44228.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] D. F. DARPA. Cyber grand challenge (cgc) (archived).

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

[10] 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.

[11] T. Mowry. Loop invariant computation and code motion.

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

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

[14] 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.