Cyber Reasoning System

Claudiu-Florentin GHENEA[1] and George-Andrei IOSIF[2]
[1,2]*Politehnica University of Bucharest, Computer Science Department, Romania*
Email: [1]claudiu.ghenea@stud.acs.upb.ro, [2]george_andrei.iosif@stud.acs.upb.ro

I. Project Idea

A **cyber reasoning system** is an automated system for managing vulnerabilities inside executable programs, starting from discovery and continuing with exploiting, signature creation and self-healing.

The first viable prototypes were shown at a competition organized by DARPA back in 2016, Cyber Grand Challenge. Then, multiple solutions were installed on physical systems that were able to communicate with each, with a single purpose: to automatically attack other systems and to defend from them, earning points in a Capture The Flag competition.

From the top 3 finalists, that are still considered **state of the art** at the moment, Mayhem (the winner) became a commercial solution [1] and Shellphish was published on GitHub as an open source project [2] and became unmaintained. Since then, some others commercial software appeared, but the open source landscape remained the same.

In our thesis, we propose the creation of an **open source cyber reasoning system** using the latest advances in the binary analysis field. As the variety regarding the executables is high, we will niche our solution for ELF CLI executables, built from a C codebase on an x86 architecture.

II. Work This Semester

As the dataset module was finished in the first semester, together with one for attack surface approximation (part of the module for vulnerability discovery), the efforts this semester focused on researching and developing the above-presented pipeline of the cyber reasoning system.

A. Fuzzing Standard Input and Files

1) Introduction: The purpose of the vulnerability detection module is to use all the previously-detected input streams of the given executable in order to detect its vulnerabilities. The latter consists in the path of the control flow graph (abbreviated CFG) which are not implemented correctly.

The issue that can emerge is the **path explosion**, a problem specific for techniques such as symbolic execution. As symbolic values are propagated on each path of the graph, the complexity of finding vulnerabilities in such a large set of data is unfeasible considering the limitations in memory and computation power.

We proceed to overcome this by using **fuzzing**. The sending of random inputs on each input stream make the execution flow on various paths, with various input variables. The probability of triggering a vulnerability residing in the bytecode could be then increased.

Another important constraint for fuzzing is the **blackbox aspect** of the CRS: the vulnerability discovery module does not have access to the source code of the executable, but only at the binary itself. This limited our search as multiple fuzzers uses source code instrumentation,

[1]https://forallsecure.com/mayhem-for-code
[2]https://github.com/shellphish

namely introducing small pieces of code before the compilation stage such that, at execution, the fuzzer can know which path of the CFG was reached by the execution.

Firstly, we considered american fuzzy lop (abbreviated AFL) [6] as a state-of-the-art blackbox fuzzer that we can integrate in the cyber reasoning system, but the Google's upstream has no development activity since June 2021. A promising alternative is a fork of it, **afl++** [7], which is continuously developed by the open source community and supports features like:

- QEMU emulation for deducing relevant information (for example, coverage) on runtime, without a hard requirement for source code instrumentation before compilation;
- Persistent mode for specifying a section of the code segment that should be executed in a loop by the fuzzer, without the need of creating a process each time new inputs should be provided to the executable; and
- Greater speed, resulting in more mutations in the fuzzing engine.

2) Architecture. Implementation: Our implementation started by considering a subset of all possible **input streams** that an executable could have:

- Opened and written/read files;
- Standard input; and
- Arguments, that are used to specify startup information such as filenames and generic strings.

The architecture in this section will cover only the first two, the arguments being detailed in a next subchapter.

As previously mentioned, afl++ was chosen as a de-facto blackbox fuzzing tool. We created a custom Docker image to be its environment, the purpose being isolation and easy regeneration. It is **orchestrated via Python** code to be automatically built, instantiated, attached to volumes (for target executable, sample inputs and analysis).

Furthermore, the same Python code started `afl-fuzz` inside the created **container** to fuzz the provided target executable with the sample inputs (if any). These are mutated and passed to standard input or placed in files (two input streams that are supported by default by afl++), data that is consumed by the binary.

When a new crash is detected by afl++, a new file is created in the analysis directory, which is continuously monitored by a Python watchdog. The crash is processed to create a new instance of a data structure that represents a proof of vulnerability. In the future implementations, it will be passed to the next modules (for example, the vulnerability analysis, that will analyze the root cause and the affected internals of the executable).

B. Fuzzing Arguments

1) Introduction: Another topic that we addressed this semester is the arguments fuzzing. This is required for two different components of the cyber reasoning system architecture:

- **Discovering the names and order of the arguments** in the attack surface discovery module; and
- **Generating random arguments** to be passed directly as argument in the vulnerability discovery module.

Due to difficulties in choosing a technology, we only implemented the first sub-task.

To detect that an argument (or a set of arguments) is used by the executable, the **dynamic approach** is recommended, as the static one could lead to false results due to shallow inspection of the control flow graph. On the other hand, by analyzing the program dynamically, the input makes the program execution flows on a different path, reaching different basic blocks (predictable sequences of instructions, seen as nodes in a CFG) and, eventually, calling different system API methods.

The goals of the first sub-task is essentially the extraction of coverage information that can be further used to differentiate between two executions of the same executable, but with different arguments. There are multiple dynamic binary analysis techniques that can be used to extract information of this kind:

- **Debugging**: By placing hardware or software breakpoints on each basic block or system call, a debugger can detect the execution flow. But being a tool for humans (programmers, vulnerability researchers, etc.), it is not properly optimized for automations in which no manual intervention is required. The performance cost came either from user - kernel spaces switches (when calling the `ptrace` API from the debugger process) or by a form of dynamic binary instrumentation, by rewriting the code with debugging-related calls or interrupts.
- **Static instrumentation and execution**: This approach consists in lifting the program into a high-level abstraction language, insertion of instrumentation code and compilation. The newly created executable is then ran and can report coverage information for its execution. Being an alternative for compiler instrumentation, this approach can lead to unexpected behavior due to a lack of standardized technologies in this field of research.
- **Dynamic binary instrumentation** (abbreviated DBI): A DBI instrument tries to solve the issues of a debugger. Taking QuarkslaB's engine, **QBDI** [11], as an example (that is actually used in our implementation), it allows the injection of instrumentation code inside the binary, at runtime. The optimization is that the analysis tool and the analyzed program runs under the umbrella of the same process, reducing the friction.

From our tests, we concluded that the last technique applied in QBDI could be used, taking in consideration its **advantages**:

- Efficiency: The slowdown is less than as in other techniques.
- Compatibility: There are multiple APIs available: C, C++, Python and even JavaScript (when used with Frida).

2) Architecture. Implementation: The first approach for integrating QBDI consisted in the following steps:

1) Transform the executable into a library by using the LIEF [10] Python library to remove the PIE flag (that is present in executables) and mark the `main()` function as exported.
2) Load the executable into a program memory using `dlopen()`.
3) Instrument the code using the C API of QBDI.

It failed due to the QBDI's mode of functioning: it considers non-reentrant all the calls to dynamic linked libraries, and it switches the execution from instrumented to native (with no instrumentation at all). After the function return, it turns back the instrumentation. This whole mechanism is called execution transfer.

This limitation made us retry with a different API of QBDI, **QBDIPreload**. It is a utility library in which all the callbacks exported by the QBDI API are overwritten and called at runtime by the DBI engine. The instrumentation library is injected in the process memory by using the Linux loader's `LD_PRELOAD`.

This second approach used a series of different steps:

1) Create a callback function for basic block call. It applies an address normalization technique to discard the variances introduced by Address Space Layout Randomization. This relativization of the start address of the basic block is stored inside a list.
2) Create a callback function for execution transfer. It checks if `close()` is called on a canary file, provided as an argument to the program.
3) On exiting, create a non-cryptographic DJB2 hash over the first 1000 addresses in the CFG and dump it inside a file.

In this manner, it can be said that if two arguments produces two different DJB2 hashes, then they produced a different sequence of nodes in the control flow graph.

The last step here is the **generation of a dictionary with arguments**, which is created by parsing all the manuals present in a Linux operating system:

1) Read the `man` configuration files to detect folders where manuals reside.
2) Get each gzip archive from this kind of folder.
3) Unarchive its content and search with Regex for common patters.

Combining the coverage information with the arguments from the dictionary, we can apply a bruteforce strategy to discover the arguments that are used by the provided executables.

C. Taint Analysis

1) Introduction: In our first period of documentation regarding the chosen subject for the master's thesis, we have considered useful the utilization of a dynamic taint analysis (abbreviated DTA) engine in our project for discovering how a certain input that generated a crash inside a binary propagates, what parts of it are involved in what functions, what common weaknesses are present from Common Weakness Enumeration (abbreviated CWE) and based on those information to be able to build the next submodules.

Dynamic taint analysis is a technique used for tracking information flow between sources and sinks during a program's execution. It is generally used in developing security related application, including automatic vulnerability detection. In general, a DTA tool labels data originating from outside of the program called **sources** (for example, program arguments, standard input, input from sockets, generally any input operation). Those sources are marked as **tainted**, after that it keeps track of the propagation of tainted data as a program executes and detects when such tainted data is used in the program, such usages or operation that include tainted data are usually called **sinks**. With this mechanism, a DTA tool provides the ability to reason about the actual execution and the ability to perform precise security analysis based on runtime information (for example, a jump to an address provided by the user).

During the semester we have identifies and tested (from an implementation and adaptation to our needs standpoint) a series of open source state of the art dynamic taint analysis tools and frameworks, that we will be presenting the following section.

2) Saluki's Tests: Saluki [3] is a tool for checking security properties in a binary by using data dependency (taint checking) and static analysis. This tool is said to be able to generate context and path sensitive data dependency relations on values that depend on a tainted source by using security policies (rules written in OCaml programming language), those policies can be used to define CWE vulnerabilities such as command injection and SQL injection. It provides a powerfull engine (called uFlux) that can execute any parts of a program within an emulator, the emulator catches and reads all instructions before they are executed in order to match the given serucity policies. It's unique approach is that of using mixed taint analysis based on random execution as it is much faster that the conventional static approach and it is said to find more vulnerabilities than soley dynamic analysis, it achieves that by using Binary Analysis Platform (abbreviated BAP) [8] as engine to derive the control flow graph in order to discover as much code as possible.

In order to implement and test this tool we have deployed it inside a docker container and tried to modify it in order to suit our needs, during this we have noticed the following limitations:

- Defining a new security policy was pretty cumbersome and it had a limited set already implemented, this would have required to write specific rulles for detecting CWEs for each analyzed binary.
- Saluki is not a standalone tool, it uses BAP as a core and acts like a plugin for it.
- Saluki's unique approach of using mixed taint analysis based on random execution did not suit or needs in the end as we are trying to find the flow of provided data and the functions/basic blocks it reaches.

3) Triton's Tests: Triton [2] is a dynamic binary analysis framework that provides a variety of components such like Dynamic Symbolic Execution, Taint Engine, SMT Solvers and a lot more. Based on the components it offers one could build program analysis tools, perform software verification and even Dynamic Tain Analysis. Triton has an active community, but no so much when it comes to their DTA engine as it does not support the abilty to track multiple sources, and an Issue regarding the rewriting of this engine is opened since May 2020.

During our testing of this framework we have encountered a limited ability to track tainted data as you have to manually set a certain register or both a register and memory address in order to keep track of tainted data as it flows through the program. Those remarks pushing us to find a better fitted solution for our needs.

4) TaintGrind's Tests: TaintGrind [13] [4] is a taint-tracking plugin for the popular Valgrind [1] dynamic binary instrumentation framework that gives full instrumentation coverage of user code. It is based on Valgrind's MemChecker and Flayer [9].

TaintGrind requires access to the source code in order to manually indicate tainted variables that will then be followed at runtime, where do they propagate, if any function calls depend on them and so on. This project is available on github and the latest commit dates back from late 2021 this project does not have a paper tied to it and no proper documentation either. The strong point of this project is that it generates a list of tainted variables based on the source and it also flags any jumps or calls using such variables, thus being really close to what we needed.

During the testing and validation period for this tool we have identified the following limitations and problems:

- TaintGrind requires modification over the source code of the binary, by introducing function calls that will mark a certain source (variables) as tainted. This requires of course recompilation and it is not quite a solution that fits our needs.
- TaintGrind only propagates explicit data flows, meaning that it will not propagate taint in control structures such as if statements, and loops (for and while).
- In order to be able to use this tool in a real case scenario we need to find a way to do binary rewriting or lifting at some level, in order to be able to add the proper calls for the taint engine to mark our interest sources as tainted and then follow them as the binary executes.

5) Preliminary Results: During the current semester we have searched and struggled with finding and modeling a dynamic taint analysis module that best suits our needs as we have tried multiple solutions. We have idenfitied multiple implementations and learned about their inner workings, learned about what they can and what they can not do. This research period benefic was fructful as we identified a potentially valid solution (TaintGrind) and a couple more leads to further investigate (angr and rex [12]).

We also have reached the conclusion that in general, almost every existing DTA tool uses a set of predefined rules for defining taint propagation from the taint source to the sink while a program executes [5]. This rule-based propagation has some fundamental limitations, limitations from which we consider worth-while mentioning the following "Specifying accurate proparation rules" [5] as it is hard to define rules even for simple operations because there are many different cases to consider. As an example, the correct propagation rule for `sink = tainted * constant` might vary for the different taint labels of variable `tainted` and for values of variable `constant` (for example, if constant is 0, our sink will not be influenced by the tainted value).

III. Planned Work

Our work was postponed by issues with finding proper technologies that could be integrated in the cyber reasoning system. Despite this, we want to continue our work in summer so that the started modules will be finished before next semester.

Firstly, the arguments fuzzing logic that was presented needs to be integrated in the attack surface discovery module. This will allow the latter to centrally report the used input streams (discovered via static analysis techniques) and arguments (only if the arguments are used at all, discovered via arguments bruteforcing).

In the vulnerability detection module, we need to add support for arguments fuzzing. afl++ already implements a method of this kind [3], but we think that it will be enhanced if correlated with the arguments bruteforcing presented in the last chapter. Additionally, the container for standard input will be slightly modified to support file fuzzing.

Regarding the dynamic tainting, we need to further test the available technologies and find a suitable one for the blackbox constraints that the CRS presents. The aforementioned solutions angr and rex, a solution used by one of the DARPA's finalists, could be a possible approach. We will take a peak in their implementation and usages. We are also discussing the idea of implementing a binary rewriting or lifting tool to be able to integrate TaintGrind in our architecture and to complete the DTA module.

The final goal here is to develop a module that will take the proof of vulnerability from the vulnerability discovery one and to perform root cause analysis (what characteristic of the input triggered the vulnerability) and impact analysis (what parts of the executable are impacted by the executable).

Lastly, we will enhance our presence on GitHub and publish our work in repositories, one for each developed module.

IV. Bibliography

[1] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[2] Florent Saudel and Jonathan Salwan. "Triton: A Dynamic Symbolic Execution Framework". In: *Symposium sur la sécurité des technologies de l'information et des communications*. SSTIC. Rennes, France, June 2015, pp. 31–54.

[3] Ivan Gotovchits, Rijnard Van Tonder, and David Brumley. "Saluki: finding taint-style vulnerabilities with static property checking". In: *Proceedings of the NDSS Workshop on Binary Analysis Research*. Vol. 2018. 2018.

[4] Wei Ming Khoo. *Taintgrind: a Valgrind taint analysis tool*. 2018.

[5] Dongdong She et al. "Neutaint: Efficient dynamic taint analysis with neural networks". In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1527–1543.

[6] *AFL's Website*. https://lcamtuf.coredump.cx/afl/.

[7] *AFL++'s Website*. https://aflplus.plus/.

[8] *BAP's Repository*. https://github.com/BinaryAnalysisPlatform/bap.

[9] *flayer's Code Archive*. https://code.google.com/archive/p/flayer/.

[10] *LIEF's Website*. https://lief-project.github.io/.

[11] *QBDI's Website*. https://forallsecure.com/mayhem-for-code.

[12] *rex's Repository*. https://github.com/angr/rex.

[13] *Taintgrind's Repository*. https://github.com/wmkhoo/taintgrind.

[3] https://github.com/AFLplusplus/AFLplusplus/tree/stable/utils/argv_fuzzing