UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de disertație

TITLUL LUCRĂRII DE LICENȚĂ

Absolvent
Numele studentului

Coordonator științific
Titlul și numele profesorului coordonatorului

București, iunie 2024

Rezumat

Abstract

Contents

Chapter 1

Introducere

Chapter 2

Background Information

In this chapter we will cover concepts relevant for the rest of the paper.

2.1 Reverse Engineering

As stated on *Wikipedia*: *"Reverse engineering is a process or method through which one attempts to understand through deductive reasoning how a previously made device, process, system, or piece of software accomplishes a task with very little (if any) insight into exactly how it does so"* [2].

In this work we will focus on reverse engineering pieces of software in the context of performing security research or malware analysis. Typically, in such contexts, the goal is to understand the behaviour of a piece of software, without having access to its source code. Depending on the type of analysis, a reverse engineer might have different approaches.

They might focus on gaining a comprehensive understanding of specific parts of the software in order to identify weaknesses, or more commonly named *vulnerabilities*. The analysis could be restricted to specific parts because for multiple reasons which include, but are not limited to: the full program being to big to justify performing a full analysis, or the existence of prior knowledge which gives higher priority to the analysis of certain code regions. With the knowledge obtained from RE, the engineer can identify and prove the existence of attack vectors on a system that is running this software. They might then write a report which covers the risks to which entities running the software are exposed, describing in detail the finding and, eventually exemplifying how an attacker might abuse the discovered vulnerabilities. In this case, the end goal is to initiate the process of patching the vulnerable piece of software.

In other instances, the engineers might perform a full and comprehensive analysis of piece of software. This is typically done when dealing with malware. The malware analyst will first try to determine if the piece of software is in fact malicious or not. If the code is malicious, it is important to determine its behaviour, how it interacts with the system,

or with outside entities (possibly by creating network traffic). Further, analysts might study and document novel techniques employed by attackers. They might also integrate the newly find malware into a detection system to prevent future uses of the respective malware [3].

Regardless of the goal, RE falls in one of two broad categories which determine the typical approach and the tooling used: **static analysis** and **dynamic analysis**.

2.2 Static Analysis

Static analysis represents the multitude of techniques employed to analyse a program without executing it. These techniques range in difficulty and complexity starting from reading source code, to reading assembly, attempting to decompile the binary and ultimately to using very advanced tools and theoretical knowledge such as Symbolic Execution (SE) engines, SMT solvers or formal methods.

In its most basic form, static analysis is equivalent with reading the source code in order to understand what the program does. However, in the context of this paper, we are dealing with binary files, compiled to machine code. The source code is not available to us, so we must resort to other analysis techniques. One option is to convert the machine code into the human readable form, called assembly. The process is known as machine code disassembly. Assembly code is typically very hard to understand for humans, but from it the logic of the program can be successfully recovered, given enough time and effort. One advantage of static analysis through reading assembly is that the entry barrier is not high in terms of the tooling required. A very basic tool such as `objdump` [**TODO**] can be enough for simple programs, but most likely a more feature rich tool such as `radare2`/`cutter` [**TODO**] might be more suitable, as these are able to display basic blocks and how all such basic block(BB TODO) relate to each other and form the Control Flow Graph(CFG TODO).

Reverse engineers typically default to more advanced static analysis tools, such as IDA or Ghidra. These tools feature a suite of functionalities, out of which, probably the most prominent is the *decompiler*. Compilation is the process of converting source code into machine code. Decompilation is the opposite: the process of converting machine code, back into source code. Compared to the disassembly process, which is deterministic and corresponds exactly to the assembly process, the decompilation process will almost never yield back the original source code. This is the case because a lot of information useful to programmers, but useless for the CPU(TODO) is lost during the compilation process. This information includes, but is not limited to: variable and function names, type information, custom defined data types such as structures, or specific language features. This is why decompilers will always output an approximation of the original code.

Let us consider a concrete example and consider Listings 2.1, 2.2, 2.3. Listing 2.1

contains the implementation of the function `add`, which takes the end of linked list and a value, and creates a new node in the list with that value, also taking the necessary steps to update the list accordingly. The code is part of a slightly larger `C` program, which we compiled and imported into Ghidra. Looking at Listing 2.2 we can see the decompilation of the exact code in the previously mentioned listing. It is immediately obvious that: the type information related to `node` structure is completely lost and that the original function was inlined by the compiler. As a result, the decompilation is an obfuscated version of the original code. This is a well known fact and advanced tools such as Ghidra offer various features, which the reverse engineer can utilise in order to remove part of the obfuscation. Listing 2.3 contains the same segment of decompiled code after a minimal amount of manual intervention, which includes: variable renaming, custom type creation and type updates. Clearly, it is a lot more human readable and bears a closer resemblance to the original code in Listing 2.1.

Decompilers and disassemblers are very powerful tools, which aid significantly in the process of static analysis. However, these tools have their shortcomings as highlighted above. Moreover, there are certain program behaviours which cannot, or are significantly harder to understand only by *looking* at the code.

```
1  void add(lnode** node, int v) {
2      // allocate memory for a new node in the liked list
3      lnode* new_node = (lnode*) malloc(sizeof(lnode));
4      new_node->val = v; // set the value
5      if (*node != NULL) {
6          (*node)->nxt = new_node; // link to the new node from the end of
       the list
7          *node = new_node; // move the list end to the new node
8      } else {
9          *node = new_node; // TODO fix comments set it as the end of the
       list, it it is the first one
10     }
11 }
```

Listing 2.1: TODO

```
1 piVar1 = (int *)malloc(0x10);
2 *piVar1 = iVar3;
3 if (piVar5 != (int *)0x0) {
4     *(int **)(piVar5 + 2) = piVar1;
5 }
6 piVar5 = piVar1;
7 if (piVar4 == (int *)0x0) {
8     piVar4 = piVar1;
9 }
```

Listing 2.2: TODO

```
new_node = (node *)malloc(0x10);
new_node->val = v;
if (last_node != (node *)0x0) {
    last_node->nxt = new_node;
}
last_node = new_node;
if (root == (node *)0x0) {
    root = new_node;
}
```

Listing 2.3: TODO

2.3 Dynamic Analysis

As described by T. Ball in his 1999 paper [1], *"dynamic analysis is the analysis of a running program"*. This type of analysis is desirable in different situations where static analysis could not extract sufficient information, or when acquiring extra information depends the program to be running Dynamic analysis is an umbrella term which covers many powerful techniques used for program analysis. A taxonomy of these techniques has been presented in a comprehensive survey by Ori et al. in 2019 [4]. We will shortly cover some of them.

2.3.1 Debugging

Debugging is a very well known technique, especially popular among developers who use it mainly to identify bugs or errors in their code. It is however a very effective and reliable form of analysing unknown programs (eg. malware). Also called *single stepping*, it involved using a tool, intuitively called a *debugger*, in order to run the program one instruction at a time. After each instruction, the analyst can inspect the state of the registers, the memory and what instructions follow. This process can also help in determining any relevant changes in the operating system itself, caused or related to the debugged program.

Debuggers use the CPU's trap flag in order to trigger an interrupt after each instruction, or only certain desired instructions. The interrupt causes a context switch from the execution of the debugged program to the debugger. To continue execution, the trap flag is set again and the context switches back to the program. The high number of context switches means that debugging is a very resource intensive analysis technique. It is also very easy to detect by the running malware, which can check the state of the trap flag and hide its behaviour in case it is debugged [4].

2.3.2 Function Call Analysis

Any type of meaningful action that a malware can take, will ultimately rely on system calls. It can be the case that these system calls are performed through function calls from an external library, such as the standard `libc` library, or from an internally defined function. Analysing function calls, the state of the program before, during and after the function call, as well as the parameters used can provide valuable information about the behaviour of the analysed program. Techniques for approaching this goal vary. For instance, we could use command line (cli TODO) programs such as `strace`, or `ltrace`, which track system calls and library calls respectively. We could also use more advanced techniques, such as function hooking. An analyst can extract more information from a function call by *hooking* (i.e. linking) a piece of code to the targeted function. What will happen is that upon the function call, the *hooked* code will run as well, which can just print debugging messaged to inform the analyst that the function has just been called, or access the state of the program at that time and save it for further inspection. [4]

Function calls can also be used as a powerful side-channel. More specifically, one can monitor the amount of `calls` which have been made since reference point in order to determine if progress has been made or not in the execution. Let's consider Listing 2.4. We're running the `crackme` program through `ltrace` to monitor function calls, with a randomly chosen input string. We notice a length check with `strlen` 4, after which the program crashes. By selecting the correct input length of 70 bytes, we can pass the length check 14. This `crackme` is a particularly good example for using this technique, because it is heavily obfuscated. We cannot effectively use static analysis on this binary, so employing dynamic analysis techniques enables us to make progress and recover the secret [5].

```
>_ python -c "print('a␣*␣42)" | ltrace ./crackme
memset(0x8625ae8, '\0', 10000)                              = 0x8625ae8
fgets("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"..., 10000, 0xf22e9700) = 0x8625ae8
strlen("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"...)                 = 43
puts("WROOONG!"WROOONG!
)                                          = 9
exit(1 <no return ...>
+++ exited (status 1) +++

>_ python -c "print('a␣*␣70)" | ltrace ./crackme
memset(0x8625ae8, '\0', 10000)                              = 0x8625ae8
fgets("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"..., 10000, 0xedcf4700) = 0x8625ae8
strlen("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"...)                 = 71
strstr("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"..., "zihldazjcn") = nil
puts("WROOONG!"WROOONG!
exit(1 <no return ...>
```

Listing 2.4: TODO

2.3.3 Fuzzing

2.3.4 Dynamic Taint analysis

Dynamic Taint analysis is a technique used to track data flow from sources to sinks. In order to achieve this goal, data considered important is given a label (*a taint*), based on a *taint introduction policy*. Typically, we would taint untrusted user input or data arriving over the network. This *tainted* data is propagated through the system based on execution and how the code interacts with the data at the opcode level. When an operation is performed on tainted data, memory locations used during the respective operation are also tainted, based on a *taint propagation policy*. Some memory areas, or code sections are also marked as *sinks*. When tainted data arrived at a sink, the path it took through the code can be traced back. In the context of malware analysis, the flow of tainted data is valuable, as it gives valuable insights about the ways the malware interacts with the user and the operating system. Taint analysis is also valuable for exploit detection, and was initially used specifically for this goal. By tainting untrusted user input one can detect unusual data flows and detect attempts at exploiting a system. In such cases, a *taint checking policy* might be used to determine further behaviour (e.g. halting execution) [4] [6].

2.3.5 Symbolic Execution

SE is a powerful program analysis technique, and one of the core techniques which the idea of this paper is based on. As such, we will cover SE in more detail compared to the other dynamic analysis approaches.

SE is typically discussed in relation with *Concrete Execution (CE)*. CE is the formal term for what we refer to as normal program execution. That is, executing a program with a concrete input until the end of a single execution path. When every possible external value (user input, response from a system call, return value of a function) has a concrete value, we're dealing with CE. Let us consider Listing 2.3.5. An input is taken at Line 2.

```
1 byte c;
2 read(c);
3
4 if (c < 1) {
5     print("too␣small");
6 } else if (c > 15) {
7     print("too␣big");
8 } else if (c % 3 == 0 || c % 5 == 0) {
```

```
    if (c % 3 == 0)
        print("fizz");
    if (c % 5 == 0)
        print("buzz");
} else {
    print(c);
}
```

SE, on the other hand, aims to explore multiple paths of execution, in order to determine what classes of inputs lead to which execution paths.

The key aspect of SE is the use of symbolic values, as opposed to concrete values. Initially, the symbolic values are unconstrained, meaning that they can represent any possible input value associated to that type. The program is executed in a controlled environment by a SE engine. The engine keeps track of a logic formula which describes the constraints that the input must satisfy, in order to follow each specific execution path that is being tracked. It also keeps trace of the symbolic memory store, which keeps track of symbolic values and memory areas, and the expressions or concrete values which these hold. As the program executes, each conditional branch splits the symbolic state into two and adds new constraints to each branch, based on the respective conditional.

2.3.6 Mixed Approaches

Chapter 3

Concluzii

Acronyms

AC Arithmetic Cromatics. *Glossary:* AC

SE Symbolic Execution. 6

Bibliography

[1] Thoms Ball. "The concept of dynamic analysis." In: *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-7. Toulouse, France: Springer-Verlag, 1999, pp. 216–234. ISBN: 3540665382.

[2] Contributors to Wikimedia projects. *Reverse engineering - Wikipedia*. [Online; accessed 21. May 2024]. Apr. 2024. URL: https://en.wikipedia.org/w/index.php?title=Reverse_engineering&oldid=1221145420.

[3] *Malware Analysis: Steps & Examples - CrowdStrike*. [Online; accessed 21. May 2024]. Apr. 2024. URL: https://www.crowdstrike.com/cybersecurity-101/malware/malware-analysis.

[4] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. "Dynamic Malware Analysis in the Modern Era—A State of the Art Survey." In: *ACM Comput. Surv.* 52.5 (Sept. 2019). ISSN: 0360-0300. DOI: 10.1145/3329786. URL: https://doi.org/10.1145/3329786.

[5] *RE: Reverse Engineering*. [Online; accessed 27. May 2024]. Apr. 2023. URL: https://cs.unibuc.ro/~crusu/re/labs.html.

[6] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)." In: *2010 IEEE Symposium on Security and Privacy*. 2010, pp. 317–331. DOI: 10.1109/SP.2010.26.