Thesis Dissertation

EXPLORING MEMORY SAFETY GUARANTEES IN GO

Nektarios Nikolaou

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

May 2024

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Exploring Memory Safety Guarantees In Go

Nektarios Nikolaou

Supervisor

Dr. Elias Athanasopoulos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus.

May 2024

Acknowledgments

First, I want to extend my sincere gratitude to my thesis supervisor Mr. Elias Athanasopoulos for his exceptional guidance, unwavering support, and encouragement throughout this thesis. His expertise and dedication have been crucial in shaping my research journey.

My gratitude to all my university professors, here at the University of Cyprus, who these past four years tutored us and provided us with comprehensive knowledge and increased my curiosity about the Computer Science field.

I also want to acknowledge Andonis Louca for his valuable contributions and feedback, which have significantly improved this work.

My heartfelt thanks go to my family and friends for their continuous support and understanding during this academic endeavor.

Abstract

In the modern world where we live the biggest amount of our work depends on a personal computer. The usage of applications and programming systems is necessary for the completion of every task that an employee or an individual handles. This fact increases the need for having safe programming systems. Unfortunately, no one can ensure that every programming system is safe, and it is globally known that there are both safe and unsafe programming systems. Secure systems ensure safe access to memory, while unsafe systems lack security assurances. The vulnerability of unsafe systems is programming bugs which gives the opportunity of exploitation to the attackers. On the other hand, safe systems depend on their run-time environment to ensure that they are secure and avoid bugs. While safe systems are perfect for use due to their security, their run-time environment costs a lot of performance overhead, which makes the usage of unsafe systems the only solution on building high performance applications. In these situations, it is necessary to find other ways to ensure the safety of the system such as hardening techniques. Even though, those systems are still insecure. Newly created systems and languages can be categorized somewhere in the middle of the old safe and unsafe systems. Their goal is to be memory safe using as lighter run-time guarantees as they can to produce better performance than the old safe systems. An example of these middle category systems and languages is Go programing language also known as Golang.

In this thesis, we try to find vulnerabilities in a compiled program written in Go programming language, despite the safety provided by the language. If the program succeeds in the compilation without any errors, then it is considered memory safe. The goal of this investigation is to convert a valid binary into an invalid one by creating bugs after the compilation. In the following thesis, we provide different examples and scenarios of violated binaries, whose bugs get inserted into binary level code artificially.

Contents

Chapter 1

Introduction

Most work nowadays is achieved through the usage of computer or mobile applications, which mainly perform mathematical calculations in their low-level implementation. The basic and fundamental tools for creating applications are programming languages which are sets of commands and instructions that are given to the computer machines to perform specific tasks depending on the set of instructions are given and their order. Programming languages pass through many stages until today in order to gain the form we know. They started form low-level languages, which are closer to the machine language and through all these years they become high-level and much easier to the programmers to understand, learn, and have the ability to use them when designing a programming system and developing computer and mobile applications. Languages also can be categorized in compiled and interpreted. To get in more details, compiled languages are translated directly into machine code that the computer's processor can execute. On the other hand, interpreted languages are not directly transformed in machine code but another program, the interpreter, reads and executes the code. As the time passed, languages' usage was getting bigger and bigger, and the result was the appearance of the first programming bugs.

Mostly compiled languages suffer till now from bugs, especially those who don't offer run-time safety support. Those bugs can create enormous system holes in applications if they get exploited. Those holes can cost a lot of money and loss of information to companies that got attacked. This fact created a wave of inspiration both for the art of exploiting and destroying programming systems and the need of building strong defenses for the existing systems written in compiled languages like C/C++. It also fired the need to develop new programming languages that tried to follow the speed of C/C++ and offer at the same time run-time security guarantees. These languages tried to be the lighter they can while holding these safety guarantees. Such examples are Rust and Go languages.

This thesis will focus on the Go language [2]. Go language is designed to prevent common bugs such as buffer overflows and memory leaks using its light goroutines on runtime.

Another key feature of Go, which makes the language safe, is the solid memory management that contains, which is focused on memory concurrency, and it is based on multiple goroutines and channels. Golang has a garbage collector that automatically frees the allocated memory, which is no longer needed, preventing memory leaks, race conditions and deadlocks. An also important characteristic that Go language contains is the type of safety, a feature that helps a lot Go developers to avoid null pointer and type mismatch errors preventing generating security vulnerabilities.

The main concern of this thesis is if someone with malicious purpose can overcome those safety measures after the compilation of a program written in Go language. In more details, is there any possibility a valid executable, which succeeded the compilation process, get violated on binary level by manual modifications? Can bugs, we mostly meet in C/C++, be created also in a correct executable with no impact on the output interface of an application?

The importance of these questions is very high in order to make optimizations both in language configuration and also on validating any application that may be released and has malicious behavior with purpose to be exploited on users' devices. Bugs will always exist due to the nature of programming and computing systems but the more we search them the more we can figure out a solution to harden the security features of every language and avoid as much threads as possible, by making it harder for hackers to take advantage of a system.

Contributions

1. Explore and exploit spatial safety features by modifying checks which prevent buffer overflows.
2. Explore the possibility of creating integer overflow bugs.

Thesis Structure

In the following chapters of this thesis, we emphasize more in Go language and its concepts, terminology and explain the methodology we followed for exploring the safety guaranties of the language. In more detail, we explain more terminologies and problems

and at the same time explain more about what is a safe and an unsafe system (Chapter 2). This chapter does not necessarily need to be read due to the fact that contains a lot of information for technological issues and someone who has the knowledge can skip this chapter. Moving on to the next chapter, Chapter 3, we discuss more about Go's practices on improving spatial safety and also explain our approach and methodology on the exploration. The next chapters, 4 and 5, are more technical and describe our different attempts to overcome Go's safety routines. Finaly, Chapter 6 presents some thoughts on how a tool can help with the validation of a binary and identify if it as violated after compilation or not.

Chapter 2

Background

2.1 Memory Safety, Safe and Unsafe Systems

2.1.1 Memory Safety

Memory Safety is a characteristic of many programming languages that keep the source code of a program as safe as possible. This is accomplished because those languages prevent programmers from introducing many different types of bugs related to the wrong usage of the memory. Some of those bugs are buffer overflows, integer overflows and dangling pointers, which are pointers that point to a memory location that is no longer valid or has been deallocated. In other words, a memory safe system is a system where memory accesses are well defined.

2.1.2 Safe Systems

Safe systems use a good form of memory management. These systems use programming languages with run-time environments like Java, Swift, C# and other languages like Python and JavaScript which are scripting languages. The safety guarantees that those languages have make them very heavy on run-time in order to produce large time overheads. Most of them use a garbage collector to ensure safe memory access and deallocation, which make them slow. Due to this fact it is very difficult for those languages to become the fundamental to a system programming application.

2.1.3 Unsafe Systems

A system is considered unsafe if the programming language used on building it does not have any built-in strategy in order to prevent wrong memory access and assure that memory management will be correct. Some of these languages are C and C++. Those languages usually are used for developing both operating systems like the Linux OS and

games. The reason that they are so commonly used for those purposes is their light run-time due to the fact that a system level application cannot afford a heavy run-time. C++ is used in gaming because it can optimize the usage of the computer resources and give to the gaming applications better graphics and the ability to accept multiple players. So those languages are more focused on performance than memory safety due to their purpose of use.

2.2 Problem with Unsafe Systems

Until this point, we passed through the explanation of what is a safe and unsafe system and also discussed their usage and purpose. In this section we will see some problems that systems may face when no safety measurements are taken by the languages themselves. Thinking about a program in Java or in C# what comes in mind is that when a programmer tries to access a memory location, like the index of an array, if accidentally use as index's number an integer greater than the legal bounds of the array a run-time error will occur. In languages without safety guarantees like C or C++ the same scenario will not be secured by the language's runtime and an over-read or an overwrite bug will be created [4].

An over-read bug is an anomaly where a program, while accessing the memory of a buffer in order to gain data from it, passes through the buffer's boundary and tries to read data from an unknown location. This phenomenon is a memory safety bug because it causes violations to the memory usage of a program. This bug is dangerous due to the fact that it can cause loss of information or give access to people with malicious intentions on gaining crucial information inside a program's data. Other times this bug may result in incorrect results on programs output, or other times results in a crash. Those two errors are the best-case scenario. The most dangerous result that an over-read bug can produce is the breach of system security which we mentioned before, and this is where the cyber security community focuses in order to find solutions for safer programming [5].

On the other hand, an overwrite bug is when a potential hacker tries to manipulate a programming error in order to overwrite data or the existing buffer boundaries on a program. The attacker most of the time alters the executable code of a program and overwrites elements of its memory and with this violation the behavior of the application

changes and becomes malicious. With this the hacker can destroy files, gain crucial information and access user's data, accounts or devices [6].

Those two issues are only some of the problems that a program may have if there are no built-in safety guarantees of the language is written on and are needed to understand the field of exploration or this thesis. In the next chapters, we examine how Go language tries to solve or prevent those errors, which can encourage malicious activity and violation of many applications. In this thesis we focused on Spatial Safety, which ensures that every memory access that takes place in a programming system is manipulated correct and it happens inside the object's boundaries that is getting accessed.

2.3 Basic Motivation of Go

Go is ranked on the higher places on StackOverflow by the programmers on the survey took place in 2023 [7]. This survey happens every year and thousands of programmers' vote, based on their experiences and the languages they love and use the most. Go it seems that has stolen a big part of programmers' hearts despite the fact that it is a "young" language in the industry of programming languages compared to the mainstream ones.

Go is a compiled, concurrent, garbage -collected, statically typed language developed at Google, and it is efficient, scalable, and productive [8]. It is considered as memory safe language due to its built-in run-time routines that prevent buffer overflows and dangling pointers. The garbage collection logic makes the memory usage well managed without the need of the programmers' code for memory management. Due to the garbage collector, it was a challenge to make the language fast because as we know languages with garbage collection strategy are slower with heavy runtime, such as Java language.

In order to be faster but remain safe, Go has a well-designed model. Its speed depends on a lot of reasons. First, Go is a compiled language, not an interpreted one, so its compilation process is faster due to the avoidance of working through a virtual machine. Second, Go has built-in support for concurrency, which gives developers the complete freedom to choose their approach on solving problems for software applications. Go is also particularly well suited for high loads applications or intensive I/O operations that require multitasking due its ability to deploy workloads across multiple CPU cores

simultaneously. Finally, Go has native compilation which fasters code execution and take full benefits of modern CPU features [9].

Go tries to be somewhere in the middle of the fast but memory unsafe languages, like C and C++, and the slower but memory safe languages like Java and C#. While writing Go, a developer could write safe and unsafe code, and this depends on him/her. If the programmer follows the rules of Go written in language's documentation combined with the built-in features that language have then we talk about a safe Go program. In this thesis we will focus on safe programs and the ability to violate them on binary level after the compilation time.

2.4 Advantages and Disadvantages when Using Go

2.4.1 Advantages

Go language has many advantages. Some of them are named and explained at a high level in this section. The fact that is a compiled language reduces the run-time of programs and it is able to compile to a single executable binary. The language is minimalist with no complexity, and it has a built-in formatting engine, which both of them help the developers to have an easier time while coding. Another benefit of the language is the automatic garbage collection, as we mentioned several times before. It has automatic memory management, which helps the programmers to focus on more important aspects of their code. There is no need for unit-testing libraries due to Go's built-in testing and benchmarking. Go language has also advanced concurrency techniques and requires very little boilerplate code to create substantial applications. The last but not least advantages mentioned in this section are the dedicated Networking API that the language features, the fast speed that has when is used for back-end operations and its youngness because it has learnt from the giants (mature languages in industry like Java and C/C++) that came before Go [10].

2.4.2 Disadvantages

As always there is no programming language , which only has advantages. The same fact stands also for the Go language. Some of the disadvantages that the language has are

mentioned in the following lines. Go has a lack of libraries and compared to other languages like Python and Java, Golang has a relatively small standard library and due to this situation developers may need third-party libraries for certain tasks. Even though the language is very simple, a problem that a programmer may face is the large time needed to learn and understand some unique features like Golang's concurrency model. In advantages we mentioned the youngness of the language but here we will also mention immaturity as a disadvantage and this happening clearly because there may be fewer tools and resources available for developers, for no other reason [11].

Chapter 3

Methodology

This chapter gets into details about the main approach of Go language's memory safety features and especially the buffer boundaries checking before accessing an index of an array. This technique is developed by the language's creators with the goal of avoiding memory management violation. The language executes these checks on run-time to ensure spatial memory safety and at the same time be as light as possible without using a virtual machine approach for its garbage collection logic. Our goal in this thesis is to explore if we can violate a "safe" binary file, which passed the compilation process without alerting any errors. In more detail we try to modify the binary code in order to overcome or change the run-time checks and insert spatial safety bugs.

3.1 Spatial Safety

As mentioned before in less detail, the goal of spatial safety is to ensure that every memory access occurs within the bounds of a known object [12]. Usually when we talk about enforcing spatial safety in languages like Java or C#, we mean that the language executes run-time checks before accessing a part of an allocated memory object like an array. In that case the language compares the index we need to access with the boundaries of the object in memory.

Go has a similar approach to preventing memory management bugs. Diving in buffer overflow bugs we saw that Go's compiler adds code to prevent unsafe memory access. These checks perform a comparison between the boundaries of the buffer we want to access and the index of the same buffer we want to read or write on it. Using this strategy Golang avoids over-read and overwrite bugs [4]. The code added as a check causes the interruption of the program before overwrite memory data in unknown memory location or before reading data that are not accessed through the object (buffer) we trying to reach.

For integer overflows we observed that the Go's compiler adds a similar check with the difference that this check is inside a function called when an integer value is checked

before its usage. This check compares the register containing the value of the integer we want to use with the max integer value than a Go program can handle. If the comparison failed, then again, the program crashes before the register's overflow takes place.

3.2 Our Methodology

In this section, we describe the methodology we followed in exploring Golang's concepts explained in the sections above. We created examples written in Go in order to explore how the language's compiler manipulates the memory. In our examples we have statically allocated arrays in order to be able to locate in binary level the check added from the compiler in order to prevent the buffer overflow. Integer overflow examples were also created with the goal of locating again the added check and if there is any difference between those two types of checks. After locating the injection from the compile code for checks, we tried to overcome it and produce spatial safety bugs like the way they happened in C or C++. All the above get analyzed in chapters 4 and 5 as different Proofs Of Concept (POC). We followed the quietest and the most cunning approach when modifying the binary because our goal was to prove that by making the minimum of modification, memory safety bugs can be regenerated from nowhere.

3.2.1 Buffer Overflow

For the buffer overflow bug, we followed the following steps:

1. Create a toy example based on each POC.
2. Compile the program, run it, and examine the output results.
3. Explore the program on binary-level, using GDB and Radare2.
4. Locate the boundaries' checks added from the compiler.
5. Modify the binary using Radare2 in order to violate it the way we want.
6. Run the modified program with the right input to exploit the bug and examine the new results.

3.2.2 Integer Overflow

For the integer overflow bug, we followed the following steps, which are very similar to the buffer overflow steps:

1. Create a toy example based on the POC.
2. Compile the program, run it, and examine the output results.
3. Explore the low-level instructions and the assembly of the program, using GDB and Radare2.
4. Locate the check for the integer boundaries added by the compiler.
5. Modify the binary at the point of the check using Radare2.
6. Run the violated program with the correct input in order to exploit the bug and observe the results.

Chapter 4

Buffer Overflow

Before moving to the details of our strategy to reach the exploitation of a program it is important to mention that Go language has two objects that are used as buffers, arrays and slices. An array has a fixed size, its length, which is part of its type and that is why arrays cannot be resized [13][14]. A slice, on the other hand, is a dynamically sized, flexible view into the elements of an array [14]. A slice does not store any data, it just describes a section of an underlying array [15]. It has both a length and a capacity. The length of a slice is the number of elements it contains, when capacity is the number of elements in the underlying array [16]. Most of the times developers use slices because they are more flexible and that is why we also focused on slices.

4.1 Proof Of Concept 1: Locate And Create First Buffer Overflow Bug

4.1.1 Exploring Assembly of a Simple Program

Before doing anything including modification and exploitation of a program we have to explore and understand how the compiler manipulates the boundaries checks. For this reason, we created a simple program that iterates through a buffer and copied its elements to a second smaller one. To do that we wrote the code that is shown below.

```go
package main

import (
    "fmt"
)

func main() {
    buffer1 := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
    buffer2 := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

    var length int = len(buffer1)

    for i := 0; i < length; i++ {
        buffer2[i] = buffer1[i]
    }

    fmt.Println(buffer2)
}
```

This program only contains the main function. In this main function two slices are initialized the way it shown and after that with the use of a for loop we copy elements from one slice to the other, which is a very common way in order to create a buffer overflow bug. The for loop has 15 iterations and tries to copy the same number of elements from *buffer1* to *buffer2*. As we see from the code *buffer2* is only 10 elements long. This creates a buffer overflow bug but the program compiles and runs.

Even though the program runs, it crashes and outputs the message: **"panic: runtime error: index out of range [10] with length 10".**

With this experiment we are sure that Go language manipulates the buffer overflow bugs by calling the panic function, which terminates the program with an error message. To understand how Go does this we have to inspect and understand the assembly code of the program to find the injected check instructions added from the compiler in order to prevent the overwrite bug. For this operation we use the gdb [17] with the dashboard add-on [18], which makes the exploration of the assembly code easier.

```
1  Dump of assembler code for function main.main:
2  0x00000000004808e0 <+0>:     lea     -0x38(%rsp),%r12
3  0x00000000004808e5 <+5>:     cmp     0x10(%r14),%r12
4  0x00000000004808e9 <+9>:     jbe     0x480a96        <main.main+438>
5  0x00000000004808ef <+15>:    push    %rbp
=== Skip Some Instructions ===
6  0x0000000000480a0e <+302>:   mov     0x28(%rsp,%rcx,8),%rdx
7  0x0000000000480a13 <+307>:   mov     %rdx,(%rax,%rcx,8)
8  0x0000000000480a17 <+311>:   inc     %rcx
9  0x0000000000480a1a <+314>:   nopw    0x0(%rax,%rax,1)
10 0x0000000000480a20 <+320>:   cmp     $0xf,%rcx
11 0x0000000000480a24 <+324>:   jge     0x480a2e        <main.main+334>
12 0x0000000000480a26 <+326>:   cmp     $0xa,%rcx
13 0x0000000000480a2a <+330>:   jb      0x480a0e        <main.main+302>
14 0x0000000000480a2c <+332>:   jmp     0x480a88        <main.main+424>
15 0x0000000000480a2e <+334>:   movups  %xmm15,0xa0(%rsp)
16 0x0000000000480a37 <+343>:   mov     $0xa,%ebx
17 0x0000000000480a3c <+348>:   mov     %rbx,%rcx
18 0x0000000000480a3f <+351>:   nop
```

```
19 0x0000000000480a40 <+352>:  call  0x40aa20                <runtime.convTslice>
20 0x0000000000480a45 <+357>:  lea   0x6e14(%rip),%rdx       # 0x487860
21 0x0000000000480a4c <+364>:  mov   %rdx,0xa0(%rsp)
22 0x0000000000480a54 <+372>:  mov   %rax,0xa8(%rsp)
23 0x0000000000480a5c <+380>:  mov   0xabde5(%rip),%rbx      # 0x52c848
24 0x0000000000480a63 <+387>:  lea   0x3bb7e(%rip),%rax      # 0x4bc5e8
25 0x0000000000480a6a <+394>:  lea   0xa0(%rsp),%rcx
26 0x0000000000480a72 <+402>:  mov   $0x1,%edi
27 0x0000000000480a77 <+407>:  mov   %rdi,%rsi
28 0x0000000000480a7a <+410>:  call  0x47b860                <fmt.Fprintln>
29 0x0000000000480a7f <+415>:  add   $0xb0,%rsp
30 0x0000000000480a86 <+422>:  pop   %rbp
31 0x0000000000480a87 <+423>:  ret
32 0x0000000000480a88 <+424>:  mov   %rcx,%rax
33 0x0000000000480a8b <+427>:  mov   $0xa,%ecx
34 0x0000000000480a90 <+432>:  call  0x462700                <runtime.panicIndex>
35 0x0000000000480a95 <+437>:  nop
36 0x0000000000480a96 <+438>:  call  0x4605e0                <runtime.morestack_noctxt>
37 0x0000000000480a9b <+443>:  nopl  0x0(%rax,%rax,1)
38 0x0000000000480aa0 <+448>:  jmp   0x4808e0                <main.main>
39 End of assembler dump.
```

On the above list of assembly instructions, we can see some interesting observations. With the purple color we marked the basic check that the for-loop does in order to run the correct number of iterations. Two instructions after that we can see the buffer check added by the compiler marked with red color. The instruction and the address marked with the green color is the action that runs the for-loop. Finaly the instruction marked with the blue color leads to the address marked also in blue, from where the panic block begins. In this panic section is where the program crashes.

To get a little bit in more detail, the three instructions in lines 12,13 and 14 are the most important for this example. They are added by the compiler to ensure that every access of the buffer's memory is safe. It checks if the for-loop's iteration counter, which is used as index for the buffer, is below the buffer's boundaries. If this happens the program continues its regular flow, otherwise it leads to the panic function in order to terminate before the overwrite bug exploits.

4.1.2 Binary Modification - Violation

Our goal is to bypass the check by modifying the binary level instructions. For this reason, we used the following code, which can show the results of the violation better in its output.

```go
package main

import (
    "fmt"
)

func copy_table(slice1, slice2 []int) {
    var length = len(slice2)

    for i := 0; i < length; i++ {
        slice1[i] = slice2[i]
    }
}

func main() {
    fmt.Println("Hello")

    slice1 := []int{5, 5, 5, 5, 5, 5, 5, 5, 5, 5}
    overwrite := []int{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    slice2 := []int{10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10}

    fmt.Print("Overwrite Slice Before: ")
    fmt.Println(overwrite)

    copy_table(slice1, slice2)

    fmt.Print("Slice1: ")
    fmt.Println(slice1)

    fmt.Print("Slice2: ")
    fmt.Println(slice2)

    fmt.Print("Overwrite Slice After: ")
    fmt.Println(overwrite)
}
```

The *copy_table* function takes two slices of integer elements and copies one into the other. The source slice of this example is *slice2* and the destination slice is *slice1*. In the *main* function there is also another slice, the *overwrite* slice, which is allocated between the two slices. What we want to show with this example is the overwrite bug. The function *copy_table* tries to copy all the 15 elements of *slice2* inside *slice1*. Although *slice1* can carry only 10 elements, we expect to see the program finishes without crashing and copying all the 15 elements of *slice2* in *slice1*. We expect that the 5 elements that cannot be stored in *slice1* will be stored in the *overwrite* slice, which is allocated after the *slice1*

in *main* function. Since we are exploring spatial safety, we expect that our program has the following arrangement of the slices.

For the modification of the binary, we used the radare2 disassembler [19]. After locating the check injected by compiler to protect the program from the overwrite bug, we patched the ***cmp*** command. In more detail, we changed it from ***cmp %rax, $0xa*** to ***cmp %rax, $0xf***. With this modification, we let the program to overwrite data outside the *slice1* because we made the boundaries of the slice "bigger" that it actually is and more specifically we tricked the program in order to copy more data and write them in unknown for *slice1* memory. After the loop finished, we are sure that all 15 elements of *slice2* copied into *slice1*.

4.1.3 Results After Modification

```
1  Hello
2  Overwrite Slice Before: [0 0 0 0 0 0 0 0 0 0]
3  Slice1: [10 10 10 10 10 10 10 10 10 10]
4  Slice2: [10 10 10 10 10 10 10 10 10 10 10 10 10 10 10]
5  Overwrite Slice After: [10 10 10 10 10 0 0 0 0 0]
```

The results above confirm our thoughts for the overwritten data. *Slice1* at the beginning it was full of 5s as it is shown in the code screenshot above. After the run of the violated binary, we can see that now is full of 10s, which are the *slice2*'s elements. We can also see that the five 10s are written after the *slice1*, in the *overwrite* slice. The fifteen 10s are marked with red in the results shown above in order to understand better how the program assigned the values of *slice2* into the *slice1* and *overwrite* slice.

Before moving to the next example, we have to mention that binary modification is successful only if the changes we make do not shift the binary. If the bytes we add shift the binary, then the program crashes and it means that the modification we are trying to make is not possible to be done.

4.2 Proof Of Concept 2: Overwriting Bug When Accessing a Slice

In this proof of concept, we explore two specific concepts that referred to the buffer overflow bug. Those two concepts are, firstly, if the compiler also injects a kind of the same check when we try to access an index of a slice and, secondly, if this is happening, we want to explore the possibility of modifying this check, bypass and create an overwrite or an over-read bug. For this example, we created a different toy program, which is explained in the following sections.

4.2.1 Creation of the Toy Program

```go
package main

import (
    "fmt"
)

func change_element(arr []int, in int) {
    arr[in] = 27
}

func main() {
    var slice = []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19}
    fmt.Print("Initial Buffer: ")
    fmt.Println(slice)

    fmt.Print("Please give the index of the element you want to change: ")

    var index int

    fmt.Scanln(&index)

    var diagnostic int = 0xdeadbeef
    change_element(slice, index)

    fmt.Print("Buffer After: ")
    fmt.Println(slice)
    fmt.Printf("diagnostic = %X\n", diagnostic)
}
```

This toy program is very simple and interacts with the user. The *change_element* function takes as parameters the *slice* and the *index*, which is the input that the user gave when the

program started. The only thing that *change_element* function does is replacing the element in the given index to 27. There is also an integer variable with the name *diagnostic*, which helps us better locate the boundaries check. Its value is a hexadecimal value, 0xdeadbeef, which is easy to find in the binary instructions. That is why it is used just before the call of the *change_element* function.

4.2.2 Locating the Check

When running the example with an input greater than 18 the program crashes. This is the correct behavior that the program must have due to the fact that the slice can only hold 19 items, so indexes are between 0 and 18. The program terminates by giving the handle to the panic function as the output says. This generates the first thoughts about the existence of similar boundaries check as the one examined in the previous proof of concept.

```
1  Initial Buffer: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19]
2  Please give the index of the element you want to change: 20
3  panic: runtime error: index out of range [20] with length 19
4
5  goroutine 1 [running]:
6  main.change_element(...)
           /home/nektarios/Documents/Diplomatiki/TestingGo/table_access/toy2.go:8
7  main.main()
           /home/nektarios/Documents/Diplomatiki/TestingGo/table_access/toy2.go:23 +0xa73
```

When we dived into more detail of the assembly code in the binary file, we located a check similar to the one we mentioned in the previous proof of concept. We are sure that we found a correct and valid check because the hexadecimal value 0x13 is the number 19 in decimal.

```
1  Dump of assembler code for function main.main:
2  0x000000000048e800 <+0>:     lea     -0x40(%rsp),%r12
3  0x000000000048e805 <+5>:     cmp     0x10(%r14),%r12
4  0x000000000048e809 <+9>:     jbe     0x48eb10              <main.main+784>
```

```
5   0x000000000048e80f <+15>:   push    %rbp
6   === Skip Some Instructions ===
7   0x000000000048ea05 <+517>:  mov     0xb0(%rsp),%rcx
8   0x000000000048ea0d <+525>:  mov     (%rcx),%rax
9   0x000000000048ea10 <+528>:  cmp     $0x13,%rax
10  0x000000000048ea14 <+532>:  jae     0x48eb05            <main.main+773>
11  0x000000000048ea1a <+538>:  mov     0x38(%rsp),%rdx
12  0x000000000048ea1f <+543>:  movq    $0x1b,(%rdx,%rax,8)
13  0x000000000048ea27 <+551>:  movups  %xmm15,0x60(%rsp)
14  0x000000000048ea2d <+557>:  lea     0x7d2c(%rip),%r8    # 0x496760
15  0x000000000048ea34 <+564>:  mov     %r8,0x60(%rsp)
16  0x000000000048ea39 <+569>:  lea     0x3ecf0(%rip),%r8   # 0x4cd730
17  0x000000000048ea40 <+576>:  mov     %r8,0x68(%rsp)
18  0x000000000048ea45 <+581>:  mov     0xb815c(%rip),%rbx  # 0x546ba8
19  0x000000000048ea4c <+588>:  lea     0x3f195(%rip),%rax  # 0x4cdbe8
20  0x000000000048ea53 <+595>:  lea     0x60(%rsp),%rcx
21  0x000000000048ea58 <+600>:  mov     $0x1,%edi
22  0x000000000048ea5d <+605>:  mov     %rdi,%rsi
23  0x000000000048ea60 <+608>:  call    0x4832c0            <fmt.Fprint>
24  === Skip Some Instructions ===
25  0x000000000048eafc <+764>:  add     $0xb8,%rsp
26  0x000000000048eb03 <+771>:  pop     %rbp
27  0x000000000048eb04 <+772>:  ret
28  0x000000000048eb05 <+773>:  mov     $0x13,%ecx
29  0x000000000048eb0a <+778>:  call    0x462ca0            <runtime.panicIndex>
30  0x000000000048eb0f <+783>:  nop
31  0x000000000048eb10 <+784>:  call    0x460b80            <runtime.morestack_noctxt>
32  0x000000000048eb15 <+789>:  jmp     0x48e800            <main.main>
33  End of assembler dump.
```

The lines marked with red is the check of the boundaries that the compiler injected in order to prevent the bug. If the index given is equal or greater than 19 the program jumps to the panic block of instructions, else the program continues its usual flow.

4.2.3 Performing the Attack

In order to perform the attack and modify the binary we used again the radare2 disassembler [19]. We changed the value of the boundaries from 19 to 30 which means that when we patched the binary, we changed the ***cmp*** command from ***cmp $0x13, %rax*** to ***cmp $0x1e, %rax***. The results of the modified binary are shown below.

```
1  Dump of assembler code for function main.main:
2  0x000000000048e800 <+0>:     lea     -0x40(%rsp),%r12
3  0x000000000048e805 <+5>:     cmp     0x10(%r14),%r12
4  0x000000000048e809 <+9>:     jbe     0x48eb10        <main.main+784>
5  0x000000000048e80f <+15>:    push    %rbp
6  === Skip Some Instructions ===
7  0x000000000048ea05 <+517>:   mov     0xb0(%rsp),%rcx
8  0x000000000048ea0d <+525>:   mov     (%rcx),%rax
9  0x000000000048ea10 <+528>:   cmp     $0x1e,%rax
10 0x000000000048ea14 <+532>:   jae     0x48eb05        <main.main+773>
11 0x000000000048ea1a <+538>:   mov     0x38(%rsp),%rdx
12 0x000000000048ea1f <+543>:   movq    $0x1b,(%rdx,%rax,8)
13 0x000000000048ea27 <+551>:   movups  %xmm15,0x60(%rsp)
14 0x000000000048ea2d <+557>:   lea     0x7d2c(%rip),%r8        # 0x496760
15 0x000000000048ea34 <+564>:   mov     %r8,0x60(%rsp)
16 0x000000000048ea39 <+569>:   lea     0x3ecf0(%rip),%r8       # 0x4cd730
17 0x000000000048ea40 <+576>:   mov     %r8,0x68(%rsp)
18 0x000000000048ea45 <+581>:   mov     0xb815c(%rip),%rbx      # 0x546ba8
19 0x000000000048ea4c <+588>:   lea     0x3f195(%rip),%rax      # 0x4cdbe8
20 0x000000000048ea53 <+595>:   lea     0x60(%rsp),%rcx
21 0x000000000048ea58 <+600>:   mov     $0x1,%edi
22 0x000000000048ea5d <+605>:   mov     %rdi,%rsi
23 0x000000000048ea60 <+608>:   call    0x4832c0        <fmt.Fprint>
24 === Skip Some Instructions ===
25 0x000000000048eafc <+764>:   add     $0xb8,%rsp
26 0x000000000048eb03 <+771>:   pop     %rbp
27 0x000000000048eb04 <+772>:   ret
28 0x000000000048eb05 <+773>:   mov     $0x13,%ecx
29 0x000000000048eb0a <+778>:   call    0x462ca0        <runtime.panicIndex>
30 0x000000000048eb0f <+783>:   nop
31 0x000000000048eb10 <+784>:   call    0x460b80        <runtime.morestack_noctxt>
```

```
32  0x000000000048eb15 <+789>:   jmp     0x48e800          <main.main>
33  End of assembler dump.
```

As we can see from the above listing nothing more changed than the ***cmp*** command. The binary did not shift and that is why modification is possible in this scenario.

4.2.4 Final Results

For this toy program we provided as input the number 25 after the violation of the binary. As we can see from the results below the program finished without crashing or displaying any error.

```
1  Initial Buffer: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19]
2  Please give the index of the element you want to change: 25
3  Buffer After: [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19]
4  diagnostic = DEADBEEF
```

We also can see that there is no difference for the elements included in buffer after the run of the patched binary and this is totally normal because the buffer can only hold 19 elements and when we give as input the number 25 the value 27 (the values that our program puts in the given index) will be written in an unknown for our program memory after the end of our buffer. If we inspect the stack of our program before and after accessing the memory at index 25, we can observe the following pictures of memory.

In the first one we can see that everything looks fine and correct, at the level we can say this for assembly code of a binary file. Numbers marked in blue are the initial values of the slice we stored when we initialized it at the beginning of the program. Those are the values 1 to 19 and below we can see them in the stack with their hexadecimal values.

- offset -	0 1	2 3	4 5	6 7	8 9	A B	C D	E F
0xc0000b4000	**01**00	0000	0000	0000	**02**00	0000	0000	0000
0xc0000b4010	**03**00	0000	0000	0000	**04**00	0000	0000	0000
0xc0000b4020	**05**00	0000	0000	0000	**06**00	0000	0000	0000
0xc0000b4030	**07**00	0000	0000	0000	**08**00	0000	0000	0000

0xc0000b4040	0900	0000	0000	0000	0a00	0000	0000	0000
0xc0000b4050	0b00	0000	0000	0000	0c00	0000	0000	0000
0xc0000b4060	0d00	0000	0000	0000	0e00	0000	0000	0000
0xc0000b4070	0f00	0000	0000	0000	1000	0000	0000	0000
0xc0000b4080	1100	0000	0000	0000	1200	0000	0000	0000
0xc0000b4090	1300	0000	0000	0000	0000	0000	0000	0000
0xc0000b40a0	0000	0000	0000	0000	0000	0000	0000	0000
0xc0000b40b0	0000	0000	0000	0000	0000	0000	0000	0000
0xc0000b40c0	0000	0000	0000	0000	0000	0000	0000	0000
0xc0000b40d0	0000	0000	0000	0000	0000	0000	0000	0000
0xc0000b40e0	0000	0000	0000	0000	0000	0000	0000	0000
0xc0000b40f0	0000	0000	0000	0000	0000	0000	0000	0000

After accessing the memory if we observe the stack, we will see that the value 0x1b will be shown, which is the value 27 in decimal. This is the value our program adds at the given index.

- offset -	0 1	2 3	4 5	6 7	8 9	A B	C D	E F
0xc0000b4000	0100	0000	0000	0000	0200	0000	0000	0000
0xc0000b4010	0300	0000	0000	0000	0400	0000	0000	0000
0xc0000b4020	0500	0000	0000	0000	0600	0000	0000	0000
0xc0000b4030	0700	0000	0000	0000	0800	0000	0000	0000
0xc0000b4040	0900	0000	0000	0000	0a00	0000	0000	0000
0xc0000b4050	0b00	0000	0000	0000	0c00	0000	0000	0000
0xc0000b4060	0d00	0000	0000	0000	0e00	0000	0000	0000
0xc0000b4070	0f00	0000	0000	0000	1000	0000	0000	0000
0xc0000b4080	1100	0000	0000	0000	1200	0000	0000	0000
0xc0000b4090	1300	0000	0000	0000	0000	0000	0000	0000
0xc0000b40a0	0000	0000	0000	0000	0000	0000	0000	0000
0xc0000b40b0	0000	0000	0000	0000	0000	0000	0000	0000
0xc0000b40c0	0000	0000	0000	0000	1b00	0000	0000	0000
0xc0000b40d0	0000	0000	0000	0000	0000	0000	0000	0000
0xc0000b40e0	0000	0000	0000	0000	0000	0000	0000	0000
0xc0000b40f0	0000	0000	0000	0000	0000	0000	0000	0000

As we can see above, the value 0x1b is marked red and it is the new entry in our stack. If we count carefully, we can also see that is at the 25th index of the memory started at 0xc0000b4000 address which is our buffer's starting point. So, we accomplished the attack with the minimum of modification on binary level, without crashing the program while it runs. We successfully modified the stack out of the legal boundaries of the slice and that means that we created an overwrite bug.

Chapter 5

Integer Overflow

In this section we will dive into integers and the possibility of creating integer overflow bugs. Our goal is to explore compiled binaries of programs that manipulate integers and find out if Go language manipulates the integer overflow with the same way that manipulates buffer overflows. More specifically we will search for a similar block of instructions injected from the compiler in order to prevent the integer overflows or find out if Go has a different approach for this kind of bug.

5.1 Proof Of Concept: Searching for Integer Overflow Check

For this proof of concept, we will focus on integer manipulation from Go language and explore the language's behavior through different examples.

5.1.1 Creation of the Toy Program

For this experiment we created a small toy example, which calculates the addition of two integers inserted from the user at the beginning of the program.

```go
package main

import (
    "fmt"
)

func addition(number1 int, number2 int) {

    var result int = number1 + number2

    fmt.Printf("The result of the addition: %d + %d = %d\n", number1, number2, result)
}

func main() {
    var input1 int
    var input2 int

    fmt.Println("Hello! Welcome to the addition program!")

    fmt.Print("Please give the first integer: ")
    fmt.Scanln(&input1)

    fmt.Print("Please give the second integer: ")
    fmt.Scanln(&input2)

    addition(input1, input2)
}
```

As we can see in the above screenshot of the source code, the function *addition* takes two integer parameters, calculates their sum, and prints the results on user's display. In function *main*, as shown above, the only thing that happens is the "reading" of user's inputs stored in integer variables *input1* and *input2*.

First Simple Run Output

```
1  Hello! Welcome to the addition program!
2  Please give the first integer: 10
3  Please give the second integer: 15
4  The result of the addition: 10 + 15 = 25
```

In the output shown above the two red numbers are *input1* and *input2* that the user gave at the beginning of the program. The number marked green is the result that *addition* function calculates and prints.

5.1.2 Exploring Binary

In order to find out if the compiler of Go language injects code in binary level in order to prevent the integer overflow, we have to observe the assembly instructions of our toy program's binary. A part of this binary, which is part of the *addition* function is shown below.

```
1   Dump of assembler code for function main.addition:
2   0x000000000048e800 <+0>:    cmp     0x10(%r14),%rsp
3   0x000000000048e804 <+4>:    jbe     0x48e8b7        <main.addition+183>
4   0x000000000048e80a <+10>:   push    %rbp
5   0x000000000048e80b <+11>:   mov     %rsp,%rbp
6   0x000000000048e80e <+14>:   sub     $0x70,%rsp
7   0x000000000048e812 <+18>:   mov     %rbx,0x88(%rsp)
8   0x000000000048e81a <+26>:   lea     (%rax,%rbx,1),%rcx
9   0x000000000048e81e <+30>:   mov     %rcx,0x38(%rsp)
10  0x000000000048e823 <+35>:   movups  %xmm15,0x40(%rsp)
11  0x000000000048e829 <+41>:   movups  %xmm15,0x50(%rsp)
12  0x000000000048e82f <+47>:   movups  %xmm15,0x60(%rsp)
13  0x000000000048e835 <+53>:   call    0x40a900        <runtime.convT64>
14  0x000000000048e83a <+58>:   lea     0x813f(%rip),%rcx  # 0x496980
```

```
15    === Skip Some Instructions ===
16    0x000000000048e8ac <+172>:    call    0x4831a0                <fmt.Fprintf>
17    0x000000000048e8b1 <+177>:    add     $0x70,%rsp
18    0x000000000048e8b5 <+181>:    pop     %rbp
19    0x000000000048e8b6 <+182>:    ret
20    0x000000000048e8b7 <+183>:    mov     %rax,0x8(%rsp)
21    0x000000000048e8bc <+188>:    mov     %rbx,0x10(%rsp)
22    0x000000000048e8c1 <+193>:    call    0x460b80                <runtime.morestack_noctxt>
23    0x000000000048e8c6 <+198>:    mov     0x8(%rsp),%rax
24    0x000000000048e8cb <+203>:    mov     0x10(%rsp),%rbx
25    0x000000000048e8d0 <+208>:    jmp     0x48e800                <main.addition>
26    End of assembler dump.
```

In the above listing is marked red the instruction that performs the addition of the two integers. There is not any check for testing the boundaries of the integer maximum value neither before nor after the addition. It seems that the compiler did not inject any kind of boundaries check. To explore the binary in more depth we also observed the main function to see if there is at least a check while “reading” the integer values inserted by the user as the program’s inputs. A snapshot of the binary is shown in the following listing.

```
1     Dump of assembler code for function main.main:
2     0x000000000048e8e0 <+0>:      lea     -0x10(%rsp),%r12
3     0x000000000048e8e5 <+5>:      cmp     0x10(%r14),%r12
4     0x000000000048e8e9 <+9>:      jbe     0x48ea75                <main.main+405>
5     0x000000000048e8ef <+15>:     push    %rbp
6     0x000000000048e8f0 <+16>:     mov     %rsp,%rbp
7     === Skip Some Instuction ===
8     0x000000000048ea2f <+335>:    mov     %rcx,0x30(%rsp)
9     0x000000000048ea34 <+340>:    mov     0xb8165(%rip),%rbx      #0x546ba0
10    0x000000000048ea3b <+347>:    lea     0x3f146(%rip),%rax      #0x4cdb88
11    0x000000000048ea42 <+354>:    mov     $0x1,%edi
12    0x000000000048ea47 <+359>:    mov     %rdi,%rsi
13    0x000000000048ea4a <+362>:    lea     0x28(%rsp),%rcx
14    0x000000000048ea4f <+367>:    call    0x489e20                <fmt.Fscanln>
15    0x000000000048ea54 <+372>:    mov     0x78(%rsp),%rcx
16    0x000000000048ea59 <+377>:    mov     (%rcx),%rbx
17    0x000000000048ea5c <+380>:    mov     0x80(%rsp),%rcx
18    0x000000000048ea64 <+388>:    mov     (%rcx),%rax
19    0x000000000048ea67 <+391>:    call    0x48e800                <main.addition>
20    0x000000000048ea6c <+396>:    add     $0x88,%rsp
21    0x000000000048ea73 <+403>:    pop     %rbp
```

```
22  0x000000000048ea74 <+404>:  ret
23  0x000000000048ea75 <+405>:  call  0x460b80          <runtime.morestack_noctxt>
24  0x000000000048ea7a <+410>:  jmp   0x48e8e0          <main.main>
25  End of assembler dump.
```

The listing shown above cannot help us find if there is any check at the time of the "reading". As we can see there are not any ***cmp*** instruction. The only hint that these assembly instructions gave us is the call to the *Fscanln* function, which performs the reading. The only option we have is to dive into this function and find our answers. On the listing below we can see the assembly instructions of the *Fsanln* function which helps us read the inputs that user gives to us.

```
1   Dump of assembler code for function fmt.Fscanln:
2   0x0000000000489e20 <+0>:    lea   -0x10(%rsp),%r12
3   0x0000000000489e25 <+5>:    cmp   0x10(%r14),%r12
4   0x0000000000489e29 <+9>:    jbe   0x489f27          <fmt.Fscanln+263>
5   0x0000000000489e2f <+15>:   push  %rbp
6   0x0000000000489e30 <+16>:   mov   %rsp,%rbp
7   0x0000000000489e33 <+19>:   sub   $0x88,%rsp
8   0x0000000000489e3a <+26>:   mov   %rax,0x98(%rsp)
9   0x0000000000489e42 <+34>:   mov   %rbx,0xa0(%rsp)
10  0x0000000000489e4a <+42>:   mov   %rsi,0xb8(%rsp)
11  0x0000000000489e52 <+50>:   mov   %rdi,0xb0(%rsp)
12  0x0000000000489e5a <+58>:   mov   %rcx,0xa8(%rsp)
13  0x0000000000489e62 <+66>:   xor   %ecx,%ecx
14  0x0000000000489e64 <+68>:   mov   $0x1,%edi
15  0x0000000000489e69 <+73>:   call  0x48a6a0          <fmt.newScanState>
16  0x0000000000489e6e <+78>:   mov   %rax,0x78(%rsp)
17  0x0000000000489e73 <+83>:   mov   %bl,0x58(%rsp)
18  0x0000000000489e77 <+87>:   mov   %cl,0x59(%rsp)
19  0x0000000000489e7b <+91>:   mov   %dil,0x5a(%rsp)
20  0x0000000000489e80 <+96>:   mov   %rsi,0x60(%rsp)
21  0x0000000000489e85 <+101>:  mov   %r8,0x68(%rsp)
22  0x0000000000489e8a <+106>:  mov   %r9,0x70(%rsp)
23  0x0000000000489e8f <+111>:  movups 0x58(%rsp),%xmm0
24  0x0000000000489e94 <+116>:  movups %xmm0,0x28(%rsp)
25  0x0000000000489e99 <+121>:  movups 0x68(%rsp),%xmm0
26  0x0000000000489e9e <+126>:  movups %xmm0,0x38(%rsp)
27  === Skip Some instuctions ===
28  0x0000000000489ea3 <+131>:  mov   0xa8(%rsp),%rbx
29  0x0000000000489eab <+139>:  mov   0xb0(%rsp),%rcx
```

```
30  0x0000000000489eb3 <+147>:  mov   0xb8(%rsp),%rdi
31  0x0000000000489ebb <+155>:  nopl  0x0(%rax,%rax,1)
32  0x0000000000489ec0 <+160>:  call  0x48e400        <fmt.(*ss).doScan>
33  0x0000000000489ec5 <+165>:  mov   %rax,0x50(%rsp)
34  0x0000000000489f59 <+313>:  mov   0x28(%rsp),%rsi
35  0x0000000000489f5e <+318>:  xchg  %ax,%ax
36  0x0000000000489f60 <+320>:  jmp   0x489e20        <fmt.Fscanln>
37  End of assembler dump.
```

After exploring this set of assembly instructions, we can see that there is no check in the function. There are not any checks in the whole program.

5.1.3 Final Results

Combining all the above results we stated believe very strongly that Go language does not implement any check for the boundaries of the integers. This fact made us dive deeper into Golang's documentation in order to find answers. The findings ensured our thoughts of not checking integers for overflow. Go language follows the same approach as C, C++ and Java languages. This means that Go language implements the 2's complement every time an integer overflow happens. If the overflow is because the value is greater than the maximum integer value, then the result starts from the minimum value and also the opposite. An example is provided below after using our toy program shown in this chapter.

```
1  Hello! Welcome to the addition program!
2  Please give the first integer: 9223372036854775807
3  Please give the second integer: 1
4  The result of the addition: 9223372036854775807 + 1 = -9223372036854775808
```

As we can see in the example above, we used as the first input of the program the maximum integer value [22] and as the second input the number 1. The final result is the minimum integer value that an integer can have in Go [22]. This is happening, as we already said, because Golang's architecture implements the 2's complement for the integer overflows.

Chapter 6

Future Work

After we finished all the previous work explained in both chapters 4 and 5, we thought a lot of future work could be done in order to both explore more aspects of spatial safety and try to find some solutions to identify if a binary is modified or not. In this chapter we will discuss in some detail what else we would like to explore and also give ideas for some kind of a validator that can separate the modified from the original compiled binaries of Go language's programs.

6.1 More Exploration

There is an aspect of spatial safety that needs exploration and maybe some of the aspects we already discussed need to dive into more depth to find out more about them.

6.1.1 More Exploration Depth

The concept it may need more in-depth exploration is the integer overflow. It may be a way to create an integer overflow bug if we find a way to "disable" the implementation of 2's complement. It is difficult due to the fact that this is the way the Go language works. With more research maybe something will come on top in order to use it.

6.1.2 New Area to Search

Something new that we have not discussed so far and can be get under investigation is the scenario of using the buffer overflow to redirect the flow of a program to a different address than the correct one. This may be a redirection from jumps to the destination function to a function we exploited in binary level and created there another bug that will makes the program gain malicious behavior. This concept contains both assembly exploration and stack observation in order to understand exactly how a program works and implement the redirection. That is the reason we believe that the investigation of this

concept will give us much more information about how language manipulates its memory every time a program is run.

6.2 Ideas for a Validator

The main idea is to create a program which can identify if a binary has modified boundaries checks like those we implemented in the previous chapters. This validator can be added as a plugin to the Google Play store or to the Apple's App Store in order to prevent users from downloading malicious applications. It can also be used by both, Google and Apple, in order to inspect easily the applications that want to be in their online Stores to identify from the beginning if they have any malicious behavior on the aspect of memory management and usage.

Chapter 7

Related Work

Since the day programing started evolving so fast until today, developers face memory safe bugs. Those bugs are found mostly in unsafe languages and produce big problems for computing systems. The attackers take advantage of those bugs and exploit many programming systems with the goal of taking advantage of the victim or gaining information and data without authorization. This fact makes computer society search and discover new ways to harden the systems' safety.

Unsafe languages like C and C++ have hardening techniques to enforce the memory safety they provide. Such an example is stack canaries which enforce the safety of the stack while a program is running [23].

Another tool that can be used in order to offer spatial safety is SafeStack, which is an instrumentation pass that protects programs against attacks based on buffer overflows [25]. It is a part of the Code-Pointer Integrity project [26].

The HardBound project can also provide spatial safety to the C language, but not in the software side because it is a new hardware design. It maintains memory layout compatibility by encoding the bounds information in a disjoint shadow space, support implicitly checks and propagates the bounds information as the bounded pointer is dereferenced, incremented, and copied to and from memory, and reduces storage and runtime overheads by caching compressed pointer encodings, thereby allowing many bounded pointers to be efficiently represented using just a few additional bits of state [24].

This thesis does not provide new solutions on detecting violations of spatial safety and we do not also try to improve the existing ones. Our goal was to explore the approach of Go language on handling its memory model. We wanted to understand Go's safe code and observe the injected, from the compiler, blocks of code that perform the boundaries checks. We also used the reasoning that a safe binary can modified and generates bugs to the program that was safe before. Finally, we tried to share some ideas for a validator tool

that could identify the modified binaries and prevent that way the usage of malicious applications.

This thesis and our whole approach of this research is based on the fact that every binary produced for a program written in Go language contains only safe code, which produced by the Golang's compiler and after the modifications the safe binary instantly becomes unsafe and malicious if someone use it for his/her personal interests.

Chapter 8

Conclusion

This thesis explored and dived into detail on the safety of binaries written in Go language. Throughout Chapter 3 we explained in detail our methodology and approach in order to accomplish our goal, to find out if we can manually create spatial safety bugs in binaries that passed successfully through their compilation. In Chapters 4 and 5 we explored different scenarios for both buffers over-read or overwrite bugs and integer overflow bugs. We provided all the necessary information to validate our reasoning for patching and inserting bugs at the binary level of a program.

We exploited programs that manipulate buffers by modifying the boundaries checks. We located the injected code that compiler adds to prevent buffer overflows and violated it at the minimum level, just to exploit the binary without cause any error occurred or crash the program. As proven in this thesis, these modifications can produce over-read and overwrite bugs, that can be used for malicious exploitations.

We also explored the possibility of the compiler injecting similar checks to prevent integer overflow bugs. After diving into the binaries and observing them carefully we did not locate any block of instructions producing any kind of check on integers. This fact and our research in depth of the documentation of Go language leads us to the conclusion that Golang does not perform any kind of boundaries checks, due to the fact that it implements the 2's complement's strategy.

Furthermore, in Chapter 6 we provided more aspects of spatial safety that can be explored in order to have more information and gain a better knowledge of how Go language manipulates and manages its memory. Lastly, in the same chapter, we suggested a potential validator, which will identify the modified binaries and can be a great tool in order to prevent the usage of malicious applications from unsuspecting users.

References

[1] J. P. M. Jr, “Memory-safe languages and security by design: Key insights, lessons learned,” ReversingLabs, Mar. 21, 2024. https://www.reversinglabs.com/blog/memory-safe-languages-and-secure-by-design-key-insights-and-lessons-learned

[2] “go.dev,” *go.dev*. https://go.dev/

[3] C. Stieg, “Memory Safe Programming Languages to Learn,” *Codecademy Blog*, Mar. 01, 2024. https://www.codecademy.com/resources/blog/memory-safe-programming-languages/

[4] “Accessing array out of bounds in C/C++,” *GeeksforGeeks*, Jul. 07, 2017. https://www.geeksforgeeks.org/accessing-array-bounds-ccpp/

[5] “About: Buffer over-read,” *dbpedia.org*. https://dbpedia.org/page/Buffer_over-read (accessed May 20, 2024).

[6] Fortinet, “What Is Buffer Overflow? Attacks, Types & Vulnerabilities,” Fortinet, 2023. https://www.fortinet.com/resources/cyberglossary/buffer-overflow

[7] “Stack Overflow Developer Survey 2023,” *Stack Overflow*. https://survey.stackoverflow.co/2023/#technology-top-paying-technologies

[8] R. Pike, “Go at Google: Language Design in the Service of Software Engineering - the Go Programming Language,” *go.dev*, 2012. https://go.dev/talks/2012/splash.article

[9] V. Le, “Golang Performance: Go Programming Language vs. Other Languages,” *Orientsoftware.com*, 2023. https://www.orientsoftware.com/blog/golang-performance/

[10] J. Etienne, “Why Go: The benefits of Golang,” *Medium*, Apr. 21, 2022. https://medium.com/@julienetienne/why-go-the-benefits-of-golang-6c39ea6cff7e

[11] C. D. Bobade, “What are the advantages and disadvantages of Golang?,” *Medium*, Apr. 30, 2023. https://chandrakant22.medium.com/what-are-the-advantages-and-disadvantages-of-golang-4c2b1cb77fbc

[12] M. S. Simpson and R. K. Barua, “MemSafe: ensuring the spatial and temporal memory safety of C at runtime,” *Software: Practice and Experience*, vol. 43, no. 1, pp. 93–128, Feb. 2012, doi: https://doi.org/10.1002/spe.2105.

[13] “A Tour of Go,” *go.dev*. https://go.dev/tour/moretypes/6.

[14] “A Tour of Go,” *go.dev*. https://go.dev/tour/moretypes/7.

[15] “A Tour of Go,” *go.dev*. https://go.dev/tour/moretypes/8.

[16] “A Tour of Go,” *go.dev*. https://go.dev/tour/moretypes/11.

[17] “GDB: The GNU Project Debugger,” *www.sourceware.org*. https://www.sourceware.org/gdb/.

[18] A. Cardaci, “cyrus-and/gdb-dashboard,” *GitHub*. https://github.com/cyrus-and/gdb-dashboard.

[19] “radare,” *rada.re*. https://rada.re/n/radare2.html.

[20] J. Griffin, “Integer Overflow in Golang,” *Medium*, Jul. 18, 2017. https://medium.com/@griffinish/integer-overflow-in-golang-9e13e274c8a5.

[21] “overflow package - github.com/FUSIONFoundation/efsn/common/overflow - go Packages,” *pkg.go.dev*. https://pkg.go.dev/github.com/FUSIONFoundation/efsn/common/overflow#section-documentation.

[22] “The maximum and minimum value of the int types in Go (Golang) | gosamples.dev,” *gosamples.dev*, Apr. 14, 2022. https://gosamples.dev/int-min-max/.

[23] M. Lemmens, “Stack Canaries – Gingerly Sidestepping the Cage | SANS Institute,” *www.sans.org*, Feb. 04, 2021. https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/

[24] J. Devietti, C. Blundell, M. Martin, and S. Zdancewic, “HardBound: Architectural Support for Spatial Safety of the C Programming Language.” Available: https://repository.upenn.edu/server/api/core/bitstreams/6ef98ba8-d1db-4eed-8c1b-65f285c37dfb/content

[25] “SafeStack — Clang 19.0.0git documentation,” *clang.llvm.org*. https://clang.llvm.org/docs/SafeStack.html.

[26] “Code-Pointer Integrity - Dependable Systems Lab,” *dslab.epfl.ch*. https://dslab.epfl.ch/research/cpi/.