Low-Level Software Security for Compiler Developers

Version 0-74-g85fd1a2

Contents

Chapter 1

Introduction

Compilers, assemblers and similar tools generate all the binary code that processors execute. It is no surprise then that for security analysis and hardening relevant for binary code, these tools have a major role to play. Often the only practical way to protect all binaries with a particular security hardening method is to let the compiler adapt its automatic code generation.

With software security becoming even more important in recent years, it is no surprise to see an ever increasing variety of security hardening features and mitigations against vulnerabilities implemented in compilers.

Indeed, compared to a few decades ago, today's compiler developer is much more likely to work on security features, at least some of their time.

Furthermore, with the ever-expanding range of techniques implemented, it has become very hard to gain a basic understanding of all security features implemented in typical compilers.

This poses a practical problem: compiler developers must be able to work on security hardening features, yet it is hard to gain a good basic understanding of such compiler features.

This book aims to help developers of code generation tools such as JITs, compilers, linkers and assemblers to overcome this.

There is a lot of material that can be found explaining individual vulnerabilities or attack vectors. There are also lots of presentations explaining specific exploits. But there seems to be a limited set of material that gives a structured overview of all vulnerabilities and exploits for which a code generator could play a role in protecting against them.

This book aims to provide such a structured, broad overview. It does not necessarily go into full details. Instead it aims to give a thorough description of all relevant high-level aspects of attacks, vulnerabilities, mitigations and hardening techniques. For further details, this book provides pointers to material with more details on specific techniques.

The purpose of this book is to serve as a guide to every compiler developer that

needs to learn about software security relevant to compilers. Even though the focus is on compiler developers, we expect that this book will also be useful to other people working on low-level software.

1.1 Why an open source book?

The idea for this book emerged out of a frustration of not finding a good overview on this topic. Kristof Beyls and Georgia Kouveli, both compiler engineers working on security features, wished a book like this would exist. After not finding such a book, they decided to try and write one themselves. They immediately realized that they do not have all necessary expertise themselves to complete such a daunting task. So they decided to try and create this book in an open source style, seeking contributions from many experts.

As you read this, the book remains unfinished. This book may well never be finished, as new vulnerabilities continue to be discovered regularly. Our hope is that developing the book as an open source project will allow for it to continue to evolve and improve. The open source development process of this book increases the likelihood that it remains relevant as new vulnerabilities and mitigations emerge.

Kristof and Georgia, the initial authors, are far from experts on all possible vulnerabilities. So what is the plan to get high quality content to cover all relevant topics? It is two-fold.

First, by studying specific topics, they hope to gain enough knowledge to write up a good summary for this book.

Second, they very much invite and welcome contributions. If you're interested in potentially contributing content, please go to the home location for the open source project at https://github.com/llsoftsec/llsoftsecbook.

As a reader, you can also contribute to making this book better. We highly encourage feedback, both positive and constructive criticisms. We prefer feedback to be received through https://github.com/llsoftsec/llsoftsecbook.

Add section describing the structure of the rest of the book.

Chapter 2

Memory vulnerability based attacks and mitigations

2.1 A bit of background on memory vulnerabilities

Memory access errors describe memory accesses that, although permitted by a program, were not intended by the programmer. These types of errors are usually defined (Hicks 2014) by explicitly listing their types, which include:

- buffer overflow
- null pointer dereference
- use after free
- use of uninitialized memory
- illegal free

Memory vulnerabilities are an important class of vulnerabilities that arise due to these types of errors, and they most commonly occur due to programming mistakes when using languages such as C/C++. These languages do not provide mechanisms to protect against memory access errors by default. An attacker can exploit such vulnerabilities to leak sensitive data or overwrite critical memory locations and gain control of the vulnerable program.

Memory vulnerabilities have a long history. The Morris worm in 1988 was the first widely publicized attack exploiting a buffer overflow. Later, in the mid-90s, a few famous write-ups describing buffer overflows appeared (Aleph One 1996). Stack buffer overflows were mitigated with stack canaries and non-executable stacks. The answer was more ingenious ways to bypass these mitigations: code reuse attacks, starting with attacks like return-into-libc (Solar Designer 1997). Code reuse attacks later evolved to Return-Oriented Programming (ROP) (Shacham 2007) and even more complex techniques.

To defend against code reuse attacks, the Address Space Layout Randomization (ASLR) and Control-Flow Integrity (CFI) measures were introduced. This interaction between offensive and defensive security research has been essential

Refine section links used here and in the previous paragraph.

to improving security, and continues to this day. Each newly deployed mitigation results in attempts, often successful, to bypass it, or in alternative, more complex exploitation techniques, and even tools to automate them.

Memory safe (Hicks 2014) languages are designed with prevention of such vulnerabilities in mind and use techniques such as bounds checking and automatic memory management. If these languages promise to eliminate memory vulnerabilities, why are we still discussing this topic?

On the one hand, C and C++ remain very popular languages, particular in the implementation of low-level software. On the other hand, programs written in memory safe languages can themselves be vulnerable to memory errors as a result of bugs in how they are implemented, e.g. a bug in their compiler. Can we fix the problem by also using memory safe languages for the compiler and runtime implementation? Even if that were as simple as it sounds, unfortunately there are types of programming errors that these languages cannot protect against. For example, a logical error in the implementation of a compiler or runtime for a memory safe language can lead to a memory access error not being detected. We will see examples of such logic errors in compiler optimizations in a later section.

Given the rich history of memory vulnerabilities and mitigations and the active developments in this area, compiler developers are likely to encounter some of these issues over the course of their careers. This chapter aims to serve as an introduction to this area. We start with a discussion of exploitation primitives, which can be useful when analyzing threat models . We then continue with a more detailed discussion of the various types of vulnerabilities, along with their mitigations, presented in a rough chronological order of their appearance, and, therefore, complexity.

Discuss threat models elsewhere in book and refer to that section here

2.2 Exploitation primitives

Newcomers to the area of software security may find themselves lost in many blog posts and other publications describing specific memory vulnerabilities and how to exploit them. Two very common, yet unfamiliar to a newcomer, terms that appear in such publications are *read primitive* and *write primitive*. In order to understand memory vulnerabilities and be able to design effective mitigations, it's important to understand what these terms mean, how these primitives could be obtained by an attacker, and how they can be used.

An *exploit primitive* is a mechanism that allows an attacker to perform a specific operation in the memory space of the victim program. This is done by providing specially crafted input to the victim program.

A *write primitive* gives the attacker some level of write access to the victim's memory space. The value written and the address written to may be controlled by the attacker to various degrees. The primitive, for example, may allow:

- writing a fixed value to an attacker-controlled address, or
- writing to an address consisting of a fixed base and an attacker-controlled offset limited to a specific range (e.g. a 32-bit offset) , or
- writing to an attacker-controlled base address with a fixed offset.

Consider describing in more detail why the range limitation matters

Primitives can be further classified according to more detailed properties. See slide 11 of (Miller, n.d.) for an example.

The most powerful version of a write primitive is an *arbitrary write* primitive, where both the address and the value are fully controlled by the attacker.

A *read primitive*, respectively, gives the attacker read access to the victim's memory space. The address of the memory location accessed will be controlled by the attacker to some degree, as for the write primitive. A particularly useful primitive is an *arbitrary read* primitive, in which the address is fully controlled by the attacker.

The effects of a write primitive are perhaps easier to understand, as it has obvious side-effects: a value is written to the victim program's memory. But how can an attacker observe the result of a read primitive?

This depends on whether the attack is interactive or non-interactive (Hu et al. 2016).

- In an *interactive attack*, the attacker gives malicious input to the victim program. The malicious input causes the victim program to perform the read the attacker instructed it to, and to output the results of that read. This output could be any kind of output, for example a network packet that the victim transmits. The attacker can observe the result of the read primitive by looking at this output, for example parsing this network packet. This process then repeats: the attacker sends more malicious input to the victim, observes the output and prepares the next input. You can see an example of this type of attack in (Beer 2020), which describes a zero-click radio proximity exploit.
- In a *non-interactive (one-shot) attack*, the attacker provides all malicious input to the victim program at once. The malicious input triggers multiple primitives one after the other, and the primitives are able to observe the effects of the preceding operations through the victim program's state. The input could be, for example, in the form of a JavaScript program (Groß 2020), or a PDF file pretending to be a GIF (Beer and Groß 2021).

How does an attacker obtain these kinds of primitives in the first place? The details vary, and in some cases it takes a combination of many techniques, some of which are out of scope for this book. But we will be describing a few of them in this chapter. For example a stack buffer overflow results in a (restricted) write primitive when the input size exceeds what the program expected.

As part of an attack, the attacker will want to execute each primitive more than once, since a single read or write operation will rarely be enough to achieve their end goal (more on this later). How can primitives be combined to perform multiple reads/writes?

In the case of an interactive attack, preparing and sending input to the victim program and parsing the output of the victim program are usually done in an external program that drives the exploit. The attacker is free to use a programming language of their choice, as long as they can interact with the victim program in it. Let's assume, for example, an exploit program in C, communicating with the victim program over TCP. In this case, the primitives are abstracted into C functions, which prepare and send packets to the victim,

The references in this section describe complicated modern exploits. Consider linking to simpler exploits, as well as some tutorial-level material.

and parse the victim's responses. Using the primitives is then as simple as calling these functions. These calls can be easily combined with arbitrary computations, all written in C, to form the exploit.

For this cycle of repeated input/output interactions to work, the state of the victim program must not be lost between the different iterations of providing input and observing output. In other words, the victim process must not be restarted.

It's interesting to note that while the read/write primitives consist of carefully constructed inputs to the victim program, the attacker can view these inputs as *instructions* to the victim program. The victim program effectively implements an interpreter unintentionally, and the attacker can send instructions to this interpreter. This is explored further in (Dullien 2020).

In the case of a non-interactive attack, all computation happens within the victim program. The duality of input data and code is even more obvious in this case, as the malicious input to the victim can be viewed as the exploit code. There are cases for which the input is obviously interpreted as code by the victim application as well, as in the case of a JavaScript program given as input to a JavaScript engine. In this case, the read/write primitives would be written as JavaScript functions, which when called have the unintended side-effect of accessing arbitrary memory that a JavaScript program is not supposed to have access to. The primitives can be chained together with arbitrary computations, also expressed in JavaScript.

There are, however, cases where the correspondence between data and code isn't as obvious. For example, in (Beer and Groß 2021), the malicious input consists of a PDF file, masquerading as a GIF. Due to an integer overflow bug in the PDF decoder, the malicious input leads to an unbounded buffer access, therefore to an arbitrary read/write primitive. In the case of JavaScript engine exploitation, the attacker would normally be able to use JavaScript operations and perform arbitrary computations, making exploitation more straightforward. In this case, there are no scripting capabilities officially supported. The attackers, however, take advantage of the compression format intricacies to implement a small computer architecture, in thousands of simple commands to the decoder. In this way, they effectively *introduce* scripting capabilities and are able to express their exploit as a program to this architecture.

So far, we have described read/write primitives. We have also discussed how an attacker might perform arbitrary computations:

- in an external program in the case of interactive attacks, or
- by using scripting capabilities (whether originally supported or introduced by the attacker) in non-interactive attacks.

Assuming an attacker has gained these capabilities, how can they use them to achieve their goals?

The ultimate goal of an attacker may vary: it may be, among other things, getting access to a system, leaking sensitive information or bringing down a service. Frequently, a first step towards these wider goals is arbitrary code execution within the victim process. We have already mentioned that the attacker will typically have arbitrary computation capabilities at this point, but arbitrary

code execution also involves things like calling arbitrary library functions and performing system calls.

Some examples of how the attacker may use the obtained primitives:

- Leak information, such as pointers to specific data structures or code, or the stack pointer.
- Overwrite the stack contents, e.g. to perform a ROP attack.
- Overwrite non-control data, e.g. authorization state. Sometimes this step is sufficient to achieve the attacker's goal, bypassing the need for arbitrary code execution.

Once arbitrary code execution is achieved, the attacker may need to exploit additional vulnerabilities in order to escape a process sandbox, escalate privilege, etc. Such vulnerability chaining is common, but for the purposes of this chapter we will focus on:

- Preventing memory vulnerabilities in the first place, thus stopping the attacker from obtaining powerful read/write primitives.
- Mitigating the effects of read/write primitives, e.g. with mechanisms to maintain Control-Flow Integrity (CFI).

2.3 Stack buffer overflows

A buffer overflow occurs when a read from or write to a data bufferexceeds its boundaries. This typically results in adjacent data structures being accessed, which has the potential of leaking or compromising the integrity of this adjacent data.

Aleph One. 1996. "Smashing the Stack for Fun and Profit." 1996. http://www.phrack.org/issues/49/14.html#article.

Beer, Ian. 2020. "An iOS Zero-Click Radio Proximity Exploit Odyssey." 2020. https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html.

Beer, Ian, and Samuel Groß. 2021. "A Deep Dive into an Nso Zero-Click iMessage Exploit: Remote Code Execution." 2021. https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html.

Dullien, Thomas. 2020. "Weird Machines, Exploitability, and Provable Unexploitability." *IEEE Transactions on Emerging Topics in Computing* 8 (2): 391–403. https://doi.org/10.1109/TETC.2017.2785300.

Groß, Samuel. 2020. "JITSploitation I: A Jit Bug." 2020. https://googleprojectzero.blogspot.com/2020/09/jitsploitation-one.html.

Hicks, Michael. 2014. "What Is Memory Safety?" 2014. http://www.pl-enthusiast.net/2014/07/21/memory-safety/.

Hu, Hong, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks." In *2016 Ieee Symposium on Security and Privacy (Sp)*, 969–86. https://doi.org/10.1109/SP.2016.62.

Miller, Matt. n.d. "Modeling the Exploitation and Mitigation of Memory Safety Vulnerabilities." Breakpoint 2012. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2012_10_Breakpoint/BreakPoint2012_Miller_Modeling_the_exploitation_and_mitigation_of_memory_safety_vulnerabilities.pdf.

Shacham, Hovav. 2007. "The Geometry of Innocent Flesh on the Bone: Return-into-Libc Without Function Calls (on the X86)." In *Proceedings of the 14th Acm Conference on Computer and Communications Security*, 552–61. CCS '07. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1315245.1315313.

Solar Designer. 1997. "Getting Around Non-Executable Stack (and Fix)." 1997. https://seclists.org/bugtraq/1997/Aug/63.