Low-Level Software Security for Compiler Developers

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 How this book is created

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 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 discussing 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

Discuss exploitation primitives

2.3 Stack overflows

Describe stack overflows and mitigations

2.4 Code reuse attacks

Discuss ROP, JOP, COOP and mitigations (ASLR, CFI etc)

2.5 Non-control data exploits


Discuss data-oriented programming and other attacks

2.6 Hardware support for protection against memory vulnerabilities


Describe architectural features for mitigating memory vulnerabilities and for CFI

2.7 Other issues


Mention other issues, e.g. sigreturn-oriented programming

2.8 JIT compiler vulnerabilities


Write section on JIT compiler vulnerabilities

Chapter 3

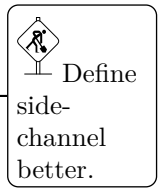
Physical access side-channel attacks

Write chapter on physical access side-channel attacks.

Chapter 4

Remote access side-channel attacks

This chapter covers side-channel attacks for which the attacker does not need physical access to the hardware.



4.1 Timing attacks

An implementation of a cryptographic algorithm can leak information about the data it processes if its run time is influenced by the value of the processed data. Attacks making use of this are called timing attacks.

The main mitigation against such attacks consists of carefully implementing the algorithm such that the execution time remains independent of the processed data. This can be done by making sure that both:

a) The control flow, i.e. the trace of instructions executed, does not change depending on the processed data. This guarantees that every time the algorithm runs, exactly the same sequence of instructions is executed, independent of the processed data.

b) The instructions used to implement the algorithm are from the subset of instructions for which the execution time is known to not depend on the data values it processes.

   For example, in the Arm architecture, the Armv8.4-A DIT extension guarantees that execution time is data-independent for a subset of the AArch64 instructions.

   By ensuring that the extension is enabled and only instructions in the subset are used, data-independent execution time is guaranteed.

At the moment, we do not know of a compiler implementation that actively helps to guarantee both (a) and (b). A great reference giving practical advice on how to achieve (a), (b) and more security hardening properties specific for cryptographic kernels is found in (Pornin 2018).

As discussed in (Pornin 2018), when implementing cryptographic algorithms, you also need to keep cache side-channel attacks in mind, which are discussed in the section on cache side-channel attacks.

4.2 Cache side-channel attacks

Write section on cache side-channel attacks. See the first comment on PR24 for suggestions of what this should contain.

Chapter 5

Supply chain attacks

A software *supply chain attack* occurs when an attacker interferes with the software development or distribution processes with the intention to impact users of that software.

Supply chain attacks and their possible mitigations are not specific to compilers. However, compilers are an attractive target for attack because they are widely deployed to developers, in continuous integration systems and as JITs. Also, an infected compiler has the possibilty to make a much larger impact if it can silently spread the infection to other software created with or run using it.

This chapter explores the history of supply chain attacks that involve compilers and what can be done to prevent them.

5.1 History of supply chain attacks

As far back as 1974 Karger & Schell theorized about an attack on the Multics operating system via the PL/I compiler (Paul A. and Roger R. 1974). In this attack, a trap door is inserted into the compiler, which then injects malicious code into generated object code. Furthermore, the trap door could be designed to reinsert itself into the compiler binary so that future compilers are silently infected without needing changes to their source code. This attack method was subsequently popularised by Ken Thompson in his 1984 ACM Turing Award acceptance speech *Reflections on Trusting Trust* (Thompson 1984).

If these cases seem far-fetched then consider that there have been several real examples of supply chain attacks on development tools.

Induc is a family of viruses that infects a pre-compiled library in the Delphi toolchain with malicious code (Gostev 2009). When Delphi compiles a project the malicious library is included into the resulting executable, thus enabling the virus to spread. The virus was first detected in 2009 and was circulating undetected for at least a year beforehand. Several popular applications are known to have been infected, including a chat client and a media player. Overall, in excess of a hundred thousand infected computers were detected world-wide by anti-virus solutions.

XcodeGhost is the name given to malware first detected in 2015 that infected thousands of iOS applications (Cox 2015). The source of the infection was tracked down to a trojanized version of Xcode tools. The malware exists in an extra object file within the Xcode tools and is silently linked into each application as it is built. File sharing sites were used to spread the trojanized Xcode tools to unwitting developers.

A trojanized linker was found to be involved in a supply chain attack discovered in 2017 named ShadowPad (Greenberg 2019). Some instances of the attack were perpetrated using a trojanized Visual Studio linker that silently incorporates a malicious library into applications as they are built. Related attacks named CCleaner and ShadowHammer used the same approach of a trojanized linker to infect built applications. Infected applications from these attacks were distributed to millions of users world-wide.

These cases highlight that attacks on compilers, and especially linkers and libraries, are a viable route to silently infect many other applications, and there is no doubt that there will be more such attacks in the future. Let us now explore what we can do about these.

Explain how these vulnerabilities arise and how to mitigate them.

Chapter 6

Other security topics relevant for compiler developers

Write chapter with other security topics.

Write section on securely clearing memory in C/C++ and undefined behaviour.

Appendix: contribution guidelines

Write chapter on contribution guidelines. These should include at least: project locaton on github; how to create pull requests/issues. Where do we discuss - mailing list? Grammar and writing style guidelines. How to use todos and index.

Index

Todo list

References

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

Cox, Joseph. 2015. "Hack Brief: Malware Sneaks into the Chinese iOS App Store." *WIRED*. https://www.wired.com/2015/09/hack-brief-malware-sneaks-chinese-ios-app-store/.

Gostev, Alexander. 2009. "A Short History of Induc." 2009. https://securelist.com/a-short-history-of-induc/30555/.

Greenberg, Andy. 2019. "Supply Chain Hackers Snuck Malware into Videogames." *WIRED*. https://www.wired.com/story/supply-chain-hackers-videogames-asus-ccleaner/.

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

Paul A., Karger, and Schell Roger R. 1974. "MULTICS Security Evaluation: VULNERABILITY Analysis," 52. https://csrc.nist.gov/csrc/media/publications/conference-paper/1998/10/08/proceedings-of-the-21st-nissc-1998/documents/early-cs-papers/karg74.pdf.

Pornin, Thomas. 2018. "Why Constant-Time Crypto?" 2018. https://www.bearssl.org/constanttime.html.

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.

Thompson, Ken. 1984. "Reflections on Trusting Trust." https://www.cs.cmu.edu/~rdriley/487/papers/Thompson_1984_ReflectionsonTrustingTrust.pdf.