Evaluating Security Hardening Options in GCC

Jacob Inwald

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

NOT STARTED, need to make methodology concrete first

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jacob Inwald*)

Acknowledgements

Any acknowledgements go here.

Table of Contents

1 Background **1**
1.1 Static Analysis 1
1.2 Fuzz Testing 2
1.3 Security Hardening 2

Bibliography **4**

Chapter 1

Background

C and C++ are both prone to a class of bugs known as memory safety errors (memory errors). Memory errors occur when memory is accessed in an undefined manner, allowing the program to write/read in unintended ways to arbritary regions of memory. This can lead to critical vulnerabilities that attackers can exploit. These errors have long been known about and long been exploited, from the Morris Worm in 1989 (Steeley [1989]) to Heartbleed in 2012 (Sass [2015]). Indeed, Microsoft has found that 70% of all its security defects in 2006-2018 were memory safety failures (Cimpanu [2019]), and the Chrome team similarly found 70% of all its vulnerabilities are memory safety issues (Cimpanu [2020]). Programming languages have since been developed that are memory safe and prevent such errors, but C and C++ remain widespread due to their speed and precision. It is infeasible to rewrite the language to prevent memory errors, and migrating away from C and C++ is not always possible for some hardware. Therefore, other approaches are required to aid in the preventation of memory errors. The key ones to discuss sit at development level (static analysis and fuzz testing) and production level (compiler hardening options).

1.1 Static Analysis

Static analysis for security assurance has been around since ITS4's release in early 2000; a simplistic syntactic matcher to rules that indicated vulnerabilities i.e. use of `strcpy()`**?**. Since then, static analysis tools have become significantly more sophisticated and complex, but the aim has always been the same - to catch security problems without executing the code. Static analysis tools will examine the program source code for flaws, marking sections of code that have potential bugs for a human programmer to then go over and resolve. These tools are incredibly effective at finding known bugs, but often rely on bugs being known. One of the more famous examples of this was Heartbleed, a memory error which was overlooked by static analysis tools until after it's discovery and subsequent use in exploits (Sass [2015]). Another key challenge with static anlysis is the reduction of false positives and false negatives. Static analysis tools walk the line between false positives (reporting a bug where there is none) and false negatives (not reporting a bug where there is one). Both situations lead to different

outcomes, with false positives leading to overhead for developers and false negatives leading to a false sense of security. Static analysis is now a strongly recommended part of the development lifecyce (SEI CERT C and C++), but is known to be fallible and is not a "cure-all" to security bugs.

1.2 Fuzz Testing

A fuzz tester (fuzzers) is a tool that iteratively mutates random input in an attempt to find security vulnerabilities for a piece of software. Fuzzers have been proven to be incredibly effective; as of 2023, a prominent fuzzer, OSS-Fuzz, has located over 10,000 vulnerabilities and 36,000 bugs across 1,000 projects (Google [2023]). Fuzzers differ to static analysis in that they *execute* the code while testing. Fuzzers rely on instrumentation, where extra code is added at compile-time to allow the fuzzer more access to the internals of program. This instrumentation can vary from sanitisers, which throw warnings or errors when dangerous behaviour is detected from the program i.e. out of bounds access, to tracing the execution path of the software or control flow (LLVM [2024]). The instrumentation will guide the mutations made by the fuzzer to iteratively improve vulnerability discovery. The upshot of this is that fuzzers can discover novel vulnerabilties not specified by the creator of the tool. These tools provide powerful security testing abilities, however, like static analysis, they rely on the developers using them effectively during development and the production lifecycle.

1.3 Security Hardening

It is almost guaranteed that production grade software will contain flaws. While static and dynamic approaches have been effective in culling these flaws, some will still slip through the gaps (i.e. Heartbleed). Therefore, security hardening is often used alongside these approaches. Security hardening attempts to prevent the memory unsafe behaviour of a language by introducing checks or transformations to workaround the potential flaws in the program. In GCC, different hardening features are enabled via the use of different command-line options such as `-fstack-protector-strong` or `-fPIe`. There are numerous hardening options to pick from, with each option aiming to solve a different problem. For example, the option `-fPIe` enables Address Space Layout Randomization, making `return2libcc` attacks difficult on 32-bit systems and near impossible on 64-bit systems **?**. Shown in Figure 1.1, is a list of security compiler options for GCC (and Clang and Binutils TODO: Remove those ones). One of those options, `-fhardened`, enables a subset of other compiler options to provide a default hardening configuration for a "naive" developer.

Compiler Flag	**Description**
`-D_FORTIFY_SOURCE`	Fortify sources with compile- and run-time checks for unsafe libc usage and buffer overflows. Some fortification levels can impact performance. Requires `-O1` or higher, may require prepending `-U_FORTIFY_SOURCE`.
`-D_GLIBCXX_ASSERTIONS`	Precondition checks for C++ standard library calls. Can impact performance.
`-fstrict-flex-arrays=3`	Consider a trailing array in a struct as a flexible array if declared as []
`-fstack-clash-protection`	Enable run-time checks for variable-size stack allocation validity. Can impact performance.
`-fstack-protector-strong`	Enable run-time checks for stack-based buffer overflows. Can impact performance.
`-fcf-protection=full`	Enable control-flow protection against return-oriented programming (ROP) and jump-oriented programming (JOP) attacks on x86_64
`-mbranch-protection=standard`	Enable branch protection against ROP and JOP attacks on AArch64
`-Wl,-z,nodlopen`	Restrict dlopen(3) calls to shared objects
`-Wl,-z,noexecstack`	Enable data execution prevention by marking stack memory as non-executable
`-Wl,-z,relro` *or* `-Wl,-z,now`	Mark relocation table entries resolved at load-time as read-only. `-Wl,-z,now` can impact startup performance.
`-fPIE -pie`	Build as position-independent executable. Can impact performance on 32-bit architectures.
`-fPIC -shared`	Build as position-independent code. Can impact performance on 32-bit architectures.
`-fno-delete-null-pointer-checks`	Force retention of null pointer checks
`-fno-strict-overflow`	Integer overflow may occur
`-fno-strict-aliasing`	Do not assume strict aliasing
`-ftrivial-auto-var-init`	Perform trivial auto variable initialization
`-fexceptions`	Enable exception propagation to harden multi-threaded C code
`-fhardened`	Enable pre-determined set of hardening options in GCC

Figure 1.1: From the Compiler Options Hardening Guide for C and C++

Bibliography

Catalin Cimpanu. Microsoft: 70 percent of all security bugs are memory safety issues, 2019. URL https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-m

Catalin Cimpanu. Chrome: 70 percent of all security bugs are memory safety issues, 2020. URL https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safet

Google. Oss-fuzz, 2023. URL https://google.github.io/oss-fuzz/.

LLVM. Sanitizer coverage, 2024. URL https://clang.llvm.org/docs/SanitizerCoverage.html#tracing-control-flow.

Jeff Sass. The role of static analysis in heartbleed. 2015. URL https://www.giac.org/paper/gsec/36189/role-static-analysis-heartbleed/143117.

Don Steeley. A tour of the worm. 1989. URL https://collections.lib.utah.edu/details?id=702918.