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

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

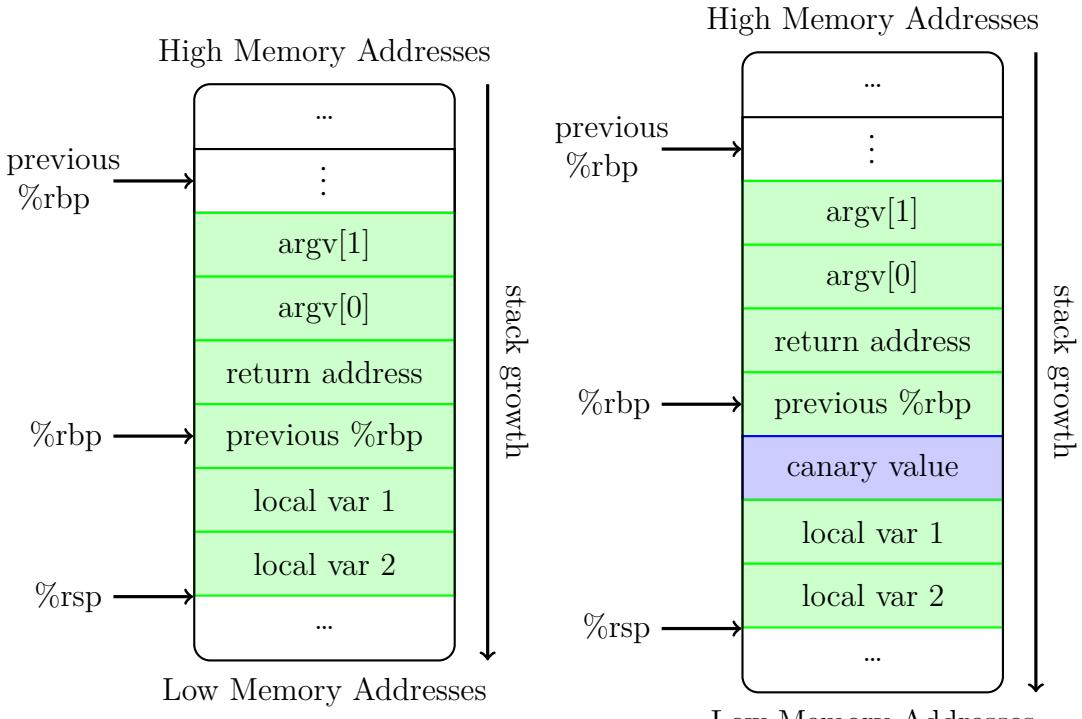
It is almost guaranteed that production grade software will contain flaws. While static and dynamic approaches have been effective in culling these flaws, some will still slip through the gaps (i.e. Heartbleed). Therefore, security hardening is often used alongside these approaches. Security hardening 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.

1.4 Leading Examples

In the previous section, the concept of security hardening was discussed, alongside some brief explanations of options available to a programmer. In this section, we will discuss the chosen flags for analysis, alongside some engineered examples to demonstrate some common use cases that they attempt to solve. We will start with the simpler flags, and then continue from there.

1.4.1 Stack Canaries

Stack canaries [Cowan et al., 1998] are a security hardening option that aim to prevent buffer overflows on the stack. The concept behind their implementation is very simple: some known word value is set before the return pointer on the stack frame and after the local variables, shown in Figure 1.2. This value is then checked against the known value before the program jumps to the return address, and aborts the program if the canary has changed. In order for the local variables to overwrite the return address in the stack frame, the canary will need to be overwritten. This means that an attacker needs to know the value of the canary in order to bypass the protection, which can prove challenging.



(a) A normal stack frame without a canary

(b) A stack frame with a canary inserted

Figure 1.2: Stack frame with and without canaries

In Figure 1.3, we see some sample assembly code after stack canaries have been added in. As shown, the canary value is loaded in the word between the return address and then checked before the function returns to the return address given.

If the canary is different to what it was set to, the program throws a stack check fail exception and then exits.

```
; start of main()
push   %rbp            ; save previous %rbp
mov    %rsp,%rbp       ;  new frame pointer for main()
sub    \$0x40,%rsp     ; makes sure there's space for local variables (adds an extra
  ↪  word of space when canaries are active)
mov    %edi,-0x34(%rbp)
mov    %rsi,-0x40(%rbp)
; inserted code
mov    %fs:0x28,%rax  ; %rax = canary value
mov    %rax,-0x8(%rbp)  ; %rbp-8 = canary value
xor    %eax,%eax

; start code block for main, left unchanged
;                    ...
; end code block for main

; inserted code
mov    -0x8(%rbp),%rdx  ; %rdx = canary value
sub    %fs:0x28,%rdx    ; compare %rdx with saved canary value
je     0x4011c9 <main+147>  ; skip error if not broken
call   0x401030 <__stack_chk_fail@plt> ; throw stack smashing error
; unchanged
leave
ret    ; exit
```

Figure 1.3: x86 assembly code after compilation with the flag

Stack canaries are effective in preventing overflow attacks on the return address, where the buffer is overflowed sequentially from a local buffer. However, this doesn't prevent corruption of local variables. For example, in Figure 1.2, if local var 1 contains a boolean value used in a check and local var 2 can be arbitrarily written, then an attacker can change the value of local var 1 with impunity. This could lead to them passing checks they shouldn't be able to. Regardless, canaries are often a simple and relatively effective method of preventing sequential buffer overflows affecting the return address of a stack frame.

1.4.2 Fortify Source

This section refers to the D_FORTIFY_SOURCE flag, which hardens inbuilt C99 library functions []. The intention is to add size checks to functions that don't implement them, such as strcpy or memcpy. This prevents library functions from writing over the size limit of a buffer, stopping buffer overflows. D_FORTIFY_SOURCE has 3 modes with the most hardened being D_FORTIFY_SOURCE=3, and the lower options providing more compatibility with legacy code.

In Figure 1.4, we see the disassembled fortified strcpy function. As shown, there

is an additional check before calling strcpy which ensures that source array is smaller than the destination array

```
endbr64
; setup frame
push   %rbp          ; save previous %rbp
mov    %rsp,%rbp     ; new frame pointer for ___strcpy_chk()
push   %r13
mov    %rdx,%r13 ; %r13 = %rdx | %rdx stores size of dest array
push   %r12
mov    %rdi,%r12
mov    %rsi,%rdi
push   %rbx
mov    %rsi,%rbx
sub    $0x8,%rsp

; perform check
call   0x7ffff7dd7510 <*ABS*+0xafb80@plt> ; calls size on source array
cmp    %r13,%rax     ; size(src) - size(dest)
jae    0x7ffff7ed8470 <___strcpy_chk+64> ; jump to last line if source size is greater
    ↪  or equal to destination size

; remove stack frame
add    $0x8,%rsp
mov    %rbx,%rsi
lea    0x1(%rax),%rdx
mov    %r12,%rdi
pop    %rbx
pop    %r12
pop    %r13
pop    %rbp

jmp    0x7ffff7dd76c0 <*ABS*+0xac430@plt>   ; jump to strcpy
call   0x7ffff7ed6a30 <___chk_fail>  ; throw exception
```

Figure 1.4: x86 assembly code for the fortified strcpy() function (___strcpy_chk) after compilation

This hardening option is a very simple transformation, and is effective in making glibc functions such as strcpy and memcpy safer. However, this option relies on the program being able to correctly find the size of the source array at runtime, something that may not be concrete [?].

1.4.3 Stack Clashing

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++

Chapter 2

Methodology

In this chapter we will discuss our approach to collecting and interrogating data. We aim to investigate whether different hardening options interact (RQ1) and whether those interactions lead to security being lost or gained (RQ2). As measuring the absolute security of a piece of software is a complex and multi-faceted process [?], we have limited the scope of this study to the 5 flags specified in Section 1. In the further sections, we outline the structure of our tests, the data sources used, the approach to curating our dataset and how we will use the curated data.

2.1 Test Structure & Data Sources

Past literature that discusses the security of hardening options will often utilize known exploit-code pairs to test this [?]. Inline with this practice, we use this method to test the security of combinations of different hardening options (RQ1, RQ2). We use 5 exploit-code pairs for each of the chosen options, to provide relevant tests for each specific option, allowing us to investigate the effect of combinations afterwards. To sandbox and ensure reproducibility of the exploit-code pairs, Docker is used [?]. All tests are run on x86_64 Linux, with hardening options that are on by default, such as ASLR, left on to simulate a normal level of security in the system.

We source our exploits from exploit.db [?] and 0day.today [?]; websites that catalog and store thousands of known exploit-code pairs. While these websites share the exploit, they do not necessarily have the related source code, as some exploits are on closed-source software. Therefore, we discard closed-source examples, and focus on open-source software which is available on version control platforms such as GitHub [?] and GitLab [?]. We further discard examples that do not work on our selected test platform (x86_64 Linux).

2.2 Data Curation

At the time of writing, there exists no standard collection of exploit-code pairings which can be easily set up and tested. There exist datasets of insecure code such as SARD [?] and SEI CERT code snippets [?], however the aims of these datasets do not align with the aims of this study. SARD is aimed at trained static analysis tools, so the code snippets provided do not have corresponding exploits, and are often taken out of context of the larger program. SEI CERT is aimed at demonstrating insecure coding practices, so the code snippets also do not have corresponding exploits. Therefore, we have curated a dataset of exploit-code pairings which fit with the aims of the study.

We use the data sources described above to collect data from. First, all exploit-code pairings that are not compatible with our test hardware and are closed-source are discarded. Next, for each option, we consider the relevant CWE/s and CVEs that are addressed by that option. We then choose exploit-code pairings that are linked to the chosen CVE/CWE; prioritizing pairings that have more available exploits and are more compatible with the test hardware. Where possible, we further supplement the data with self-authored exploits. Finally, we create Docker containers for each pairing to ensure reproducibility of results. To assess the security impact of combination options, we can then compare the effectiveness of the exploits between combinations of flags.

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-memory-safety-issues/.

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-safety-issues/.

Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In USENIX security symposium, volume 98, pages 63–78. San Antonio, TX, 1998.

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.