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

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 arbitrary 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

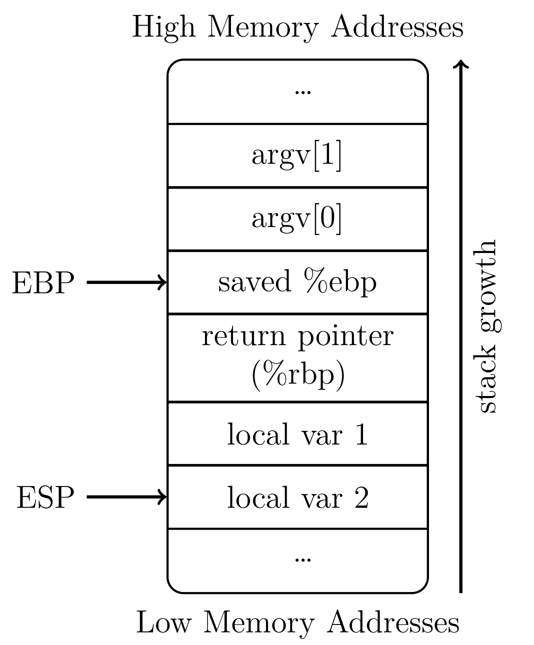
It is almost guaranteed that production grade software will contain flaws. While static and dynamic approaches have been effective in culling these flaws, some will still slip through the gaps (i.e. Heartbleed). Therefore, security hardening is often used alongside these approaches. Security hardening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

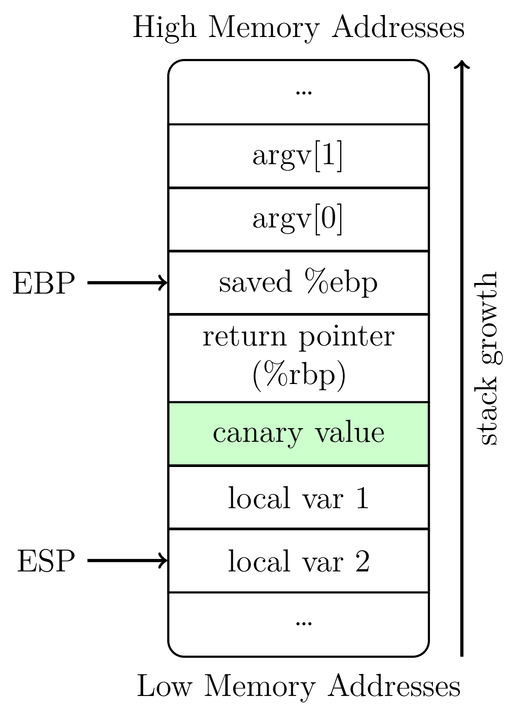
In the previous section, the concept of security hardening was discussed, alongside some brief explanations of options available to a programmer. In this section, we will discuss the chosen flags for analysis, alongside some engineered examples to demonstrate some common use cases that they attempt to 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              ; preserves previous return pointer
mov    %rsp,%rbp         ; save return pointer to %rbp
sub    \$0x40,%rsp       ; makes sure there's space for local variables (changes from 0x32 to 0x40 with and without canaries)
mov    %edi,-0x34(%rbp)
mov    %rsi,-0x40(%rbp)
; inserted code
mov    %fs:0x28,%rax  ; load canary value into %rax
mov    %rax,-0x8(%rbp)  ; store canary value 8 bytes before the return address
xor    %eax,%eax

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

; inserted code
mov    -0x8(%rbp),%rdx  ; load canary value into %rdx
sub    %fs:0x28,%rdx     ; compare 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

THe

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 section we discuss our approach to collecting and interrogating data. We are aiming to investigate the real-life functionality

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.