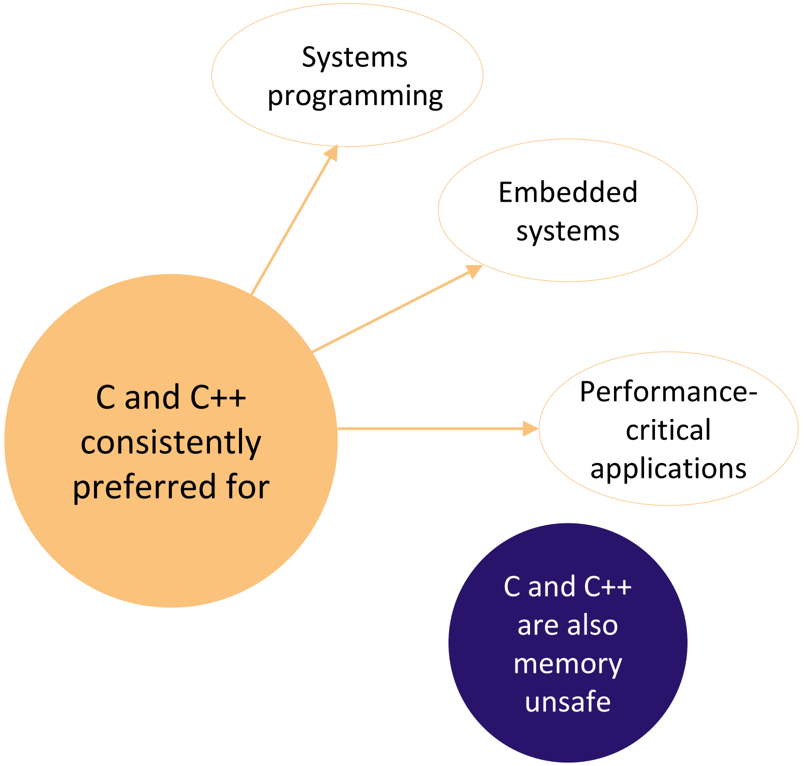
Compiler Options Hardening Guide

for C and C++

Thomas Nyman, Ericsson

The C and C++ Hardening Challenge


C and C++ consistently preferred for
Systems programming
Embedded systems
Performance-critical applications
C and C++ are also memory unsafe

Addressing vulnerabilities in C and C++ on a large scale presents several significant challenges:

- Rewriting existing C and C++ code to memory-safe languages unbearably expensive
- Unsafe dependencies will slow down migration to memory-safe languages, such as Rust*

*) Recent data indicates that over 70% of Rust crates, have dependencies on C or C++

Recent Regulatory Attention

- May 2021: US Presidential Executive Order on Improving the Nation’s Cybersecurity
- September 2022: EU Cyber Resilience Act (Proposal)
- November 2022: NSA Guidance on Software Memory-safety Issues
- December 2022: UK Telecommunications Security Code of Practice
- April 2023: CISA joint publication with 10 national cybersecurity authorities on Shifting the Balance of Cybersecurity Risk: Security-by-Design and -Default
- October 2023: Updated CISA publication with 18 national cybersecurity authorities: Principles and Approaches for Secure by Design
- December 2023: CISA joint publication with 8 national cybersecurity authorities on Memory Safe Roadmaps
- December 2023: Political agreement on EU Cyber Resilience Act
- February 2024: US White House ONCD report Back to the building blocks
- May 2024: US National Cybersecurity Strategy Implementation Plan

Compiler Options Hardening for C and C++

Guide in ***configuring programming tools*** during development to ***reduce attack surface of produced software***

C.f. Product Hardening

Provides guidance in configuring a products *operational parameters* to secure defaults to *reduce attack surface of deployed software*

C and C++ Compilers

Provide optional features that must be enabled to add protection against various security flaws to compiled binaries, e.g., applications and shared libraries

Major Linux distributions

Already package software with such protections enabled by default

Consuming OSS

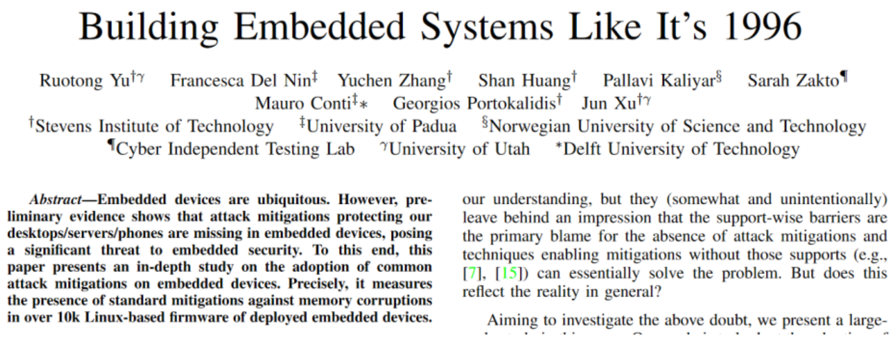
From source means you are responsible for ensuring that these protection features are enabled when building the software

Challenges for deploying hardened compiler options

Possible deployment pitfalls

- Default enabled features depends on compiler, compiler version and where it is sourced from
- OSS projects that do not enable or support protection options in their build system or code
- Protection features that require tradeoffs in performance, memory, or increased binary size
- Protection features that are incompatible with certain language constructs or patterns

"[…] 85.3% of desktop binaries adopt Stack Canaries, but only 29.7% of embedded binaries do"



Building Embedded Systems Like It's 1996

Ruotong Yu[†γ] Francesca Del Nin[‡] Yuchen Zhang[†] Shan Huang[†] Pallavi Kaliyar[§] Sarah Zakto[¶]
Mauro Conti[‡*] Georgios Portokalidis[†] Jun Xu[†γ]
[†]Stevens Institute of Technology [‡]University of Padua [§]Norwegian University of Science and Technology
[¶]Cyber Independent Testing Lab [γ]University of Utah [*]Delft University of Technology

Abstract—**Embedded devices are ubiquitous. However, preliminary evidence shows that attack mitigations protecting our desktops/servers/phones are missing in embedded devices, posing a significant threat to embedded security. To this end, this paper presents an in-depth study on the adoption of common attack mitigations on embedded devices. Precisely, it measures the presence of standard mitigations against memory corruptions in over 10k Linux-based firmware of deployed embedded devices.**

our understanding, but they (somewhat and unintentionally) leave behind an impression that the support-wise barriers are the primary blame for the absence of attack mitigations and techniques enabling mitigations without those supports (e.g., [7], [15]) can essentially solve the problem. But does this reflect the reality in general?

Aiming to investigate the above doubt, we present a large-

[Network and Distributed Systems Security (NDSS) Symposium 2022]

Compiler options hardening is not a silver bullet, but necessary <u>in combination with</u> memory-safe languages, secure coding standards, and security testing

What is covered by the guide?

1 Recommended Compiler Options

- Hardening options widely available in open-source compilers, currently GCC and Clang/LLVM
- Includes both flags that will warn developers about, as well as harden software
- Most of these options are already enabled by the major Linux distributions today

2 Discouraged Compiler Options

- Compiler options that, when used inappropriately, may result in potential defects with significant security implications in produced binaries.

3 Sanitizers

- Compiler-based tools designed to detect and pinpoint memory-safety issues and other defects
- Valuable diagnostics for debugging and testing
- May be prohibitively expensive for release builds due to performance penalties & memory overhead

4 Separating debug data from release builds

- Recommendation for managing debug information that aids in binary analysis and reverse engineering
- However, decompilers can work without debug information, so security of a system must *not* depend on omitting such information

Additional content (in incubation)

5 Compiler attribute annotations

- Separate guide for using GCC and Clang attribute annotations
- Annotations provide additional metadata to compilers
- Enabling better code analysis, benefiting security and performance

TL;DR;

```
-O2 -Wall -Wformat -Wformat=2 -Wconversion -Wimplicit-fallthrough \
-Werror=format-security \
-U_FORTIFY_SOURCE -D_FORTIFY_SOURCE=3 \
-D_GLIBCXX_ASSERTIONS \
-fstrict-flex-arrays=3 \
-fstack-clash-protection -fstack-protector-strong \
-Wl,-z,nodlopen -Wl,-z,noexecstack \
-Wl,-z,relro -Wl,-z,now \
-Wl,--as-needed -Wl,--no-copy-dt-needed-entries
```

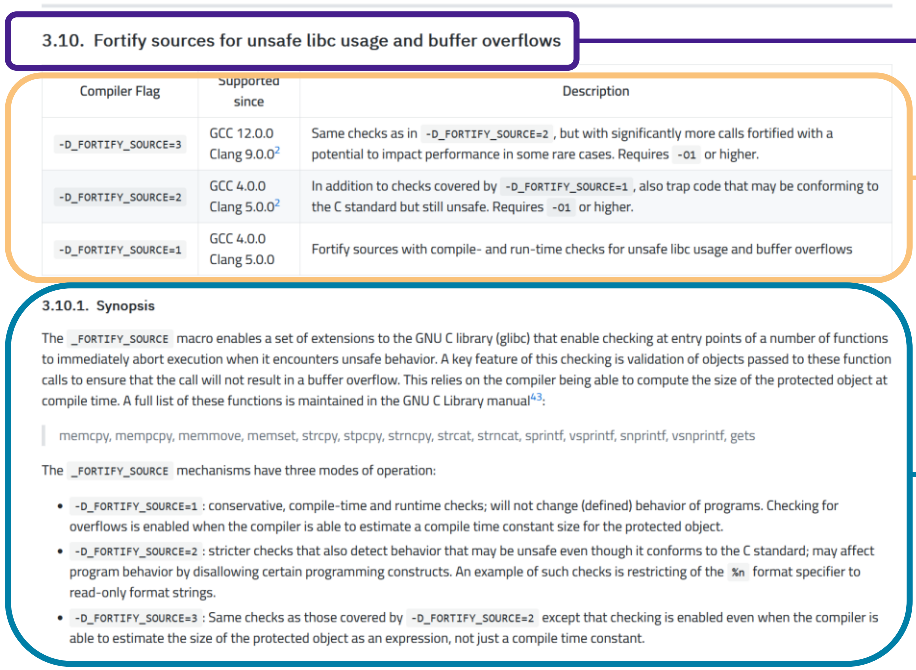
+ conditional options

Conditional options

When	Additional options flags
using GCC	`-Wtrampolines`
using GCC and only left-to-right writing in source code	`-Wbidi-chars=any`
for executables	`-fPIE -pie`
for shared libraries	`-fPIC -shared`
for x86_64	`-fcf-protection=full`
for aarch64	`-mbranch-protection=standard`
for production code	`-fno-delete-null-pointer-checks -fno-strict-overflow -fno-strict-aliasing -ftrivial-auto-var-init=zero`
for treating obsolete C constructs as errors	`-Werror=implicit -Werror=incompatible-pointer-types -Werror=int-conversion`
for multi-threaded C code using GNU C library pthreads	`-fexceptions`

OpenSSF
OPEN SOURCE SECURITY FOUNDATION

Example: -D_FORTIFY_SOURCE



3.10. Fortify sources for unsafe libc usage and buffer overflows

Compiler Flag	Supported since	Description
`-D_FORTIFY_SOURCE=3`	GCC 12.0.0 Clang 9.0.0[2]	Same checks as in `-D_FORTIFY_SOURCE=2` , but with significantly more calls fortified with a potential to impact performance in some rare cases. Requires `-O1` or higher.
`-D_FORTIFY_SOURCE=2`	GCC 4.0.0 Clang 5.0.0[2]	In addition to checks covered by `-D_FORTIFY_SOURCE=1` , also trap code that may be conforming to the C standard but still unsafe. Requires `-O1` or higher.
`-D_FORTIFY_SOURCE=1`	GCC 4.0.0 Clang 5.0.0	Fortify sources with compile- and run-time checks for unsafe libc usage and buffer overflows

3.10.1. Synopsis

The `_FORTIFY_SOURCE` macro enables a set of extensions to the GNU C library (glibc) that enable checking at entry points of a number of functions to immediately abort execution when it encounters unsafe behavior. A key feature of this checking is validation of objects passed to these function calls to ensure that the call will not result in a buffer overflow. This relies on the compiler being able to compute the size of the protected object at compile time. A full list of these functions is maintained in the GNU C Library manual[43]:

> memcpy, mempcpy, memmove, memset, strcpy, stpcpy, strncpy, strcat, strncat, sprintf, vsprintf, snprintf, vsnprintf, gets

The `_FORTIFY_SOURCE` mechanisms have three modes of operation:

- `-D_FORTIFY_SOURCE=1` : conservative, compile-time and runtime checks; will not change (defined) behavior of programs. Checking for overflows is enabled when the compiler is able to estimate a compile time constant size for the protected object.
- `-D_FORTIFY_SOURCE=2` : stricter checks that also detect behavior that may be unsafe even though it conforms to the C standard; may affect program behavior by disallowing certain programming constructs. An example of such checks is restricting of the `%n` format specifier to read-only format strings.
- `-D_FORTIFY_SOURCE=3` : Same checks as those covered by `-D_FORTIFY_SOURCE=2` except that checking is enabled even when the compiler is able to estimate the size of the protected object as an expression, not just a compile time constant.

Descriptive title

Option flags (with notable variants)
Compatible compilers and versions
High-level description

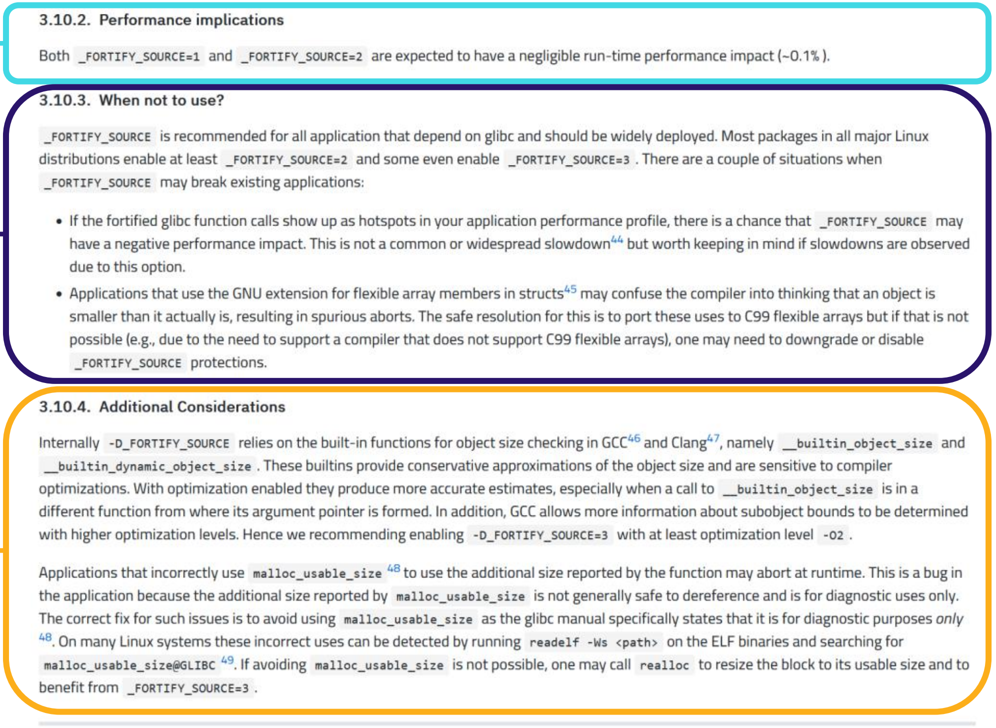
Synopsis; with explanation of the objective of the feature and basic usage

Example: -D_FORTIFY_SOURCE (cont.)

Performance implications; what aspects contribute to performance or memory overhead.

Contraindicators; conditions which may suggest against leveraging particular feature

Additional considerations; in-depth information of internals, common pitfalls, possibilities for additional tuning, etc.



3.10.2. Performance implications

Both `_FORTIFY_SOURCE=1` and `_FORTIFY_SOURCE=2` are expected to have a negligible run-time performance impact (~0.1%).

3.10.3. When not to use?

`_FORTIFY_SOURCE` is recommended for all application that depend on glibc and should be widely deployed. Most packages in all major Linux distributions enable at least `_FORTIFY_SOURCE=2` and some even enable `_FORTIFY_SOURCE=3`. There are a couple of situations when `_FORTIFY_SOURCE` may break existing applications:

- If the fortified glibc function calls show up as hotspots in your application performance profile, there is a chance that `_FORTIFY_SOURCE` may have a negative performance impact. This is not a common or widespread slowdown[44] but worth keeping in mind if slowdowns are observed due to this option.
- Applications that use the GNU extension for flexible array members in structs[45] may confuse the compiler into thinking that an object is smaller than it actually is, resulting in spurious aborts. The safe resolution for this is to port these uses to C99 flexible arrays but if that is not possible (e.g., due to the need to support a compiler that does not support C99 flexible arrays), one may need to downgrade or disable `_FORTIFY_SOURCE` protections.

3.10.4. Additional Considerations

Internally `-D_FORTIFY_SOURCE` relies on the built-in functions for object size checking in GCC[46] and Clang[47], namely `__builtin_object_size` and `__builtin_dynamic_object_size`. These builtins provide conservative approximations of the object size and are sensitive to compiler optimizations. With optimization enabled they produce more accurate estimates, especially when a call to `__builtin_object_size` is in a different function from where its argument pointer is formed. In addition, GCC allows more information about subobject bounds to be determined with higher optimization levels. Hence we recommending enabling `-D_FORTIFY_SOURCE=3` with at least optimization level `-O2`.

Applications that incorrectly use `malloc_usable_size` [48] to use the additional size reported by the function may abort at runtime. This is a bug in the application because the additional size reported by `malloc_usable_size` is not generally safe to dereference and is for diagnostic uses only. The correct fix for such issues is to avoid using `malloc_usable_size` as the glibc manual specifically states that it is for diagnostic purposes *only* [48]. On many Linux systems these incorrect uses can be detected by running `readelf -Ws <path>` on the ELF binaries and searching for `malloc_usable_size@GLIBC` [49]. If avoiding `malloc_usable_size` is not possible, one may call `realloc` to resize the block to its usable size and to benefit from `_FORTIFY_SOURCE=3`.

Roadmap and how to contribute

- New features, new compilers
- Contributions that improve readability, presentation and accessibility also welcome
- Development happens in the Best Practices WG community on GitHub and on OpenSSF Slack.
- The Compiler Hardening sub-initiative has regular Zoom calls, see Public Calendar

Compiler Options Hardening Guide for C and C++

https://best.openssf.org/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C++

Ways to Participate

- Join a Working Group/Project
- Come to a Meeting (see Public Calendar)
- Collaborate on Slack
- Contribute on GitHub
- Become an Organizational Member
- Keep up to date by subscribing to the OpenSSF Mailing List

OpenSSF
OPEN SOURCE SECURITY FOUNDATION

Legal Notice