Compiler Options Hardening Guide for C and C++

by the Open Source Security Foundation (OpenSSF) Best Practices Working Group, 2024-01-31

This document is a guide for compiler and linker options that contribute to delivering reliable and secure code using native (or cross) toolchains for C and C++. The objective of compiler options hardening is to produce application binaries (executables) with security mechanisms against potential attacks and/or misbehavior.

Hardened compiler options should also produce applications that integrate well with existing platform security features in modern operating systems (OSs). Effectively configuring the compiler options also has several benefits during development such as enhanced compiler warnings, static analysis, and debug instrumentation.

This document is intended for:

- Those who write C or C++ code, to help them ensure that resulting code will work with hardened options, including for embedded devices, Internet of Things devices, smartphones, and personal computers.
- Those who build C or C++ code for use in production environments, including Linux distributions, device makers, and those who compile C or C++ for their local environment.

This document focuses on recommended options for the GNU Compiler Collection (GCC) and Clang/LLVM, and we expect the recommendations to be applicable to other compilers based on GCC and Clang technology[1]. In the future, we aim to expand to guide to also cover other compilers, such as Microsoft MSVC.

1. TL;DR: What compiler options should I use?

When compiling C or C++ code on compilers such as GCC and clang, turn on these flags for detecting vulnerabilities at compile time and enable run-time protection mechanisms:

```
-O2 -Wall -Wformat=2 -Wconversion -Wimplicit-fallthrough \
-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
```

Note that support for some options may differ between different compilers, e.g. support for `-D_FORTIFY_SOURCE` varies depending on the compiler[2] and C standard library implementations. See the discussion below for background and for detailed discussion of each option.

When compiling code in any of the situations in the below table, add the corresponding additional options:

When	Additional options flags
using GCC	`-Wtrampolines`
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`

We recommend developers to additionally use a blanket `-Werror` to treat all warnings as errors during development. However, `-Werror` should not be used in this blanket form when distributing source code, as this use of `-Werror` creates a dependency on specific toolchain vendors and versions. The selective form `-Werror=` *`<warning-flag>`* that promotes specific warnings as error in cases that should never occur in the code can be used both during development and when distributing sources.

In this guide, we use the term *production code* for executable code intended for use in the real world with real effects; it should be maximally reliable and performant. We use the term *instrumented test code* for executable code that is instrumented to improve defect detection and debuggability, and as such, often crashes more and is slower. Test processes should use both instrumented test code and production code.

Developers should ensure that both their production code and their instrumented test code pass their automated test suite with all their relevant options. We encourage developers to consider it a bug if the program cannot be compiled with these options. Those who build production code may choose to omit some hardening options that hurt performance if the program only processes trusted data, but remember that it's not helpful to deploy programs that that are insecure and rapidly do the wrong thing. Existing programs may need to be modified over time to work with some of these options.

2. Background

2.1. Why do we need compiler options hardening?

Sadly, attackers today attack the software we use every day. Many programming languages' compilers have options to detect potential vulnerabilities while compiling and/or insert runtime protections against potential attacks. These can be important in any language, but these options are *especially* important in C and C++.

Applications written in the C and C++ programming languages are prone to exhibit a class of software defects known as memory safety errors, a.k.a. memory errors. This class of defects include bugs such as buffer overflows, dereferencing a null pointer, and use-after-free errors. Memory errors can occur because the low-level memory management in C and C++ offers no language-level provisions for ensuring the memory safety of operations such as pointer arithmetic or direct memory accesses. Instead, they require software developers to write correct code when performing common operations, and this has proven to be difficult at scale. Memory errors have the potential to cause memory vulnerabilities, which can be exploited by threat actors to gain unauthorized access to computer systems through run-time attacks. Microsoft has found that 70% of all its security defects in 2006-2018 were memory safety failures[3], and the Chrome team similarly found 70% of all its vulnerabilities are memory safety issues.[4]

Most programming languages prevent such defects by default. A few languages allow programs to temporarily suspend these protections in special circumstances, but they are intended for use in a few lines, not the whole program. There have been calls to rewrite C and C++ programs in other languages, but this is expensive and time-consuming, has its own risks, is sometimes impractical today (especially for less-common CPUs). Even with universal agreement, it would take decades to rewrite all such code. Consequently, it's important to take other steps to reduce the likelihood of defects becoming vulnerabilities. Aggressive use of compiler options can sometimes detect vulnerabilities or help counter their run-time effects.

Run-time attacks differ from conventional malware, which carries out its malicious program actions through a dedicated program executable, in that run-time attacks influence benign programs to behave maliciously. A run-time attack that exploits unmitigated memory vulnerabilities can be leveraged by threat actors as the initial attack vectors that allow them to gain a presence on a system, e.g., by injecting malicious code into running programs.

Modern, security-aware C and C++ software development practices, e.g., secure coding standards such as SEI CERT C[5] and C++[6], and program analysis aim to proactively avoid introducing memory errors (and other software defects) to applications. However, in practice completely eradicating memory errors in production C and C++ software has turned out to be near-impossible.

Consequently, modern operating systems deploy various run-time mechanisms to protect against potential security flaws. The principal purpose of such mechanisms is to mitigate potentially exploitable memory vulnerabilities in a way that prevents a threat actor from exploiting them to gain code execution capabilities. With mitigations in place the affected application may still crash if a

memory error is triggered. However, such an outcome is still preferable if the alternative is the compromise of the system's run-time environment.

To benefit from the protection mechanism provided by the OS the application binaries must be prepared at build time to be compatible with the mitigations. Typically, this means enabling specific option flags for the compiler or linker when the software is built.

Some mechanisms may require additional configuration and fine tuning, for example due to potential compilation issues for certain unlikely edge cases, or performance overhead the mitigation adds for certain program constructs. Some compiler security features depend on data flow analysis of programs and heuristics, results of which may vary depending on program source code details. As a result, the protection mechanisms implemented by these features may not always provide full coverage.

These problems are exacerbated in projects that rely on an outdated version of an open source software (OSS) compiler. In general, security mitigations are more likely to be enabled by default in modern versions of compilers included with Linux distributions. Note that the defaults used by the upstream GCC project do not enable some of these mitigations.

If compiler options hardening is overlooked or neglected during build time it can become impossible to add hardening to already distributed executables. It is therefore good practice to evaluate which mitigations an application should support, and make conscious, informed decisions whenever not enabling a mitigation weakens the application's defensive posture. Ensure that the software is *tested* with as many options as practical, to ensure it can be operated that way.

Some organizations require selecting hardening rules. For example, the US government's NIST SP 800-218 practice PW.6 requires configuring "the compilation, interpreter, and build processes to improve executable security" [7]. Carnegie Mellon University (CMU)'s "top 10 secure coding practices" recommends compiling "code using the highest warning level available for your compiler and eliminate warnings by modifying the code."[8] This guide can help you do that.

2.2. How should this guide be applied?

How you apply this guide depends on your circumstances:

- New or nearly-new project ("Green field"): If you're starting a new project, enable everything as soon as you can, preferably before any code is written for it. That way, you'll be immediately notified of any problematic constructs and avoid it in the future.
- Existing non-trivial project ("Brown field"): It's usually impractical to enable all options at once. First, the number of warnings will probably be overwhelming. Second, while the run-time protection mechanisms will usually not cause correctly-working programs to fail, it's still possible for them to cause problems (e.g., due to increased binary size). Instead, enable one or a few options at a time, assess their impact, resolve any problems, and repeat over time. Some flags (like `-Wall` are groups of other flags; consider breaking them down and enabling a few of those specific flags at a time.

Applications should work towards compiling warning-free. This takes time, but warnings indicate a potential problem. Once done, any new warning indicates a potential problem.

2.3. What should you do when compiling compilers?

If you are compiling a C/C++ compiler, where practical make the generated compiler's default options the *secure* options. For example, when compiling GCC, use `--enable-default-pie` (which enables the flags `-fPIE` and `-pie` by default when using the generated compiler executable) and `--enable-default-ssp` (which enables `-fstack-protector-strong` by default). Similarly, when compiling clang on Linux systems, set `CLANG_DEFAULT_PIE_ON_LINUX` (which has a similar effect as the option `--enable-default-pie` when compiling GCC).

2.4. What does compiler options hardening not do?

Compiler options hardening is not a silver bullet; it is not sufficient to rely solely on security features and functions to achieve secure software. Security is an emergent property of the entire system that relies on building and integrating all parts properly. However, if properly used, secure compiler options will complement existing processes, such as static and dynamic analysis, secure coding practices, negative test suites, profiling tools, and most importantly: security hygiene as a part of a solid design and architecture.

2.5. What is our threat model, goal, and objective?

Our threat model is that all software developers make mistakes, and sometimes those mistakes lead to vulnerabilities. In addition, some malicious developers may intentionally create code that *appears* to be an unintentional vulnerability, or *appears* correct but is intentionally deceiving to reviewers (aka underhanded code[9]).

Our primary goal is to counter vulnerabilities that *appear* to be unintentional (whether or not they're intentional). Our secondary goal is to counter malicious code where its source code's appearance is designed to deceive reviewers.

Many vulnerabilities are caused by common mistakes. Therefore, when implementing these goals, much of our focus is on detecting and countering *common* mistakes, whether or not they are vulnerabilities in a particular circumstance. We especially (but not exclusively) focus on countering memory safety issues, since as discussed above, memory safety issues cause most of the vulnerabilities in C and C++ code.

We are *not* trying to counter software whose source code is clearly written to be malicious. Compilers generally can't counter that, and other countermeasures (such as source code peer review) are more effective countermeasures.

Given these goals, this guidance has the following objectives:

1. *Minimize* the likelihood and/or impact of vulnerabilities that are released in production code.

2. *Maximize* the detection of vulnerabilities during compilation or test (especially when using instrumented test code), so they can be repaired before release.
3. Detect underhanded code[9] (especially Trojan source[10]), where practical, to make peer review more effective.

This guidance cannot guarantee these results. However, when combined with other measures, they can significantly help.

3. Recommended Compiler Options

This section describes recommendations for compiler and linker option flags that 1) enable compile-time checks that warn developers of potential defects in the source code (Table 1), and 2) enable run-time protection mechanisms, such as checks that are designed to detect when memory vulnerabilities in the application are exploited (Table 2).

The recommendations in Table 1 and Table 2 are primarily applicable to compiling user space code in GNU/Linux environments using either the GCC and Binutils toolchain or the Clang / LLVM toolchain and have been included in this document because they are:

- widely deployed and enabled by default for pre-built packages in major Linux distributions, including Debian, Ubuntu, Red Hat and SUSE Linux.
- supported both by the GCC and Clang / LLVM toolchains.
- cross-platform and supported on (at least) Intel and AMD 64-bit x86 architectures as well as the 64-bit version of the ARM architecture (AArch64).

For historical reasons, the GCC compiler and Binutils upstream projects do not enable optimization or security hardening options by default. While some aspects of the default options can be changed when building GCC and Binutils from source, the defaults used in the toolchains shipped with GNU/Linux distributions vary. Distributions may also ship multiple versions of toolchains with different defaults. Consequently, developers need to pay attention to compiler and linker option flags, and manage them according to their need of optimization, level of warning and error detection, and security hardening of the project.

To identify the default flags used by GCC or Clang on your system, you can examine the output of `cc -v <sourcefile.c>` and review the full command line used by the compiler to build the specified source file. This information serves two main purposes: understanding the setup of the compiler on your system and gaining insights into the options chosen by the distribution's maintainers. Additionally, it can be valuable for diagnosing option-related issues or troubleshooting problems that may arise during software compilation. For instance, certain option flags rely on their order of appearance; when a parameter is set more than once, the later occurrence usually takes precedence. By analyzing the complete list of utilized flags, it becomes easier to troubleshoot issues caused by interactions between order-sensitive flags.

Similarly, running `cc -O2 -dM -E - < /dev/null` will produce a comprehensive list of macro-defined constants. This output can be useful for troubleshooting problems related to compiler or library features that are enabled through specific macro definitions.

It's important to note that sourcing GCC from third-party vendor may result in your instance of GCC being preconfigured with certain default flags enabled or disabled. These flags can significantly impact the security of your compiled code. Therefore, it's essential to review the default flags if GCC is sourced through a Package Manager, Linux Distribution, or otherwise. We recommend explicitly enabling desired compiler flags in your build scripts or build system configuration rather than relying on the toolchain defaults. If you are creating packages for Linux distributions the distributions maintainers may have their own recommended ways of incorporating build flags. In such cases refer to the corresponding distribution documentation for, e.g., Debian[11], Gentoo[12], Fedora[13], OpenSUSE[14], or Ubuntu[15].

Typical compiler configurations do not report warnings from system headers, since application developers typically don't control those headers. In GCC this is because `-Wno-system-headers` is on by default, and clang also normally suppresses warnings from system headers [16]. You will probably want to also mark third party include files as system headers so you can strongly increase the warning levels. Directories added with the command line option `-isystem` are treated as system header directories by GCC [17] and Clang [18]. In a Cmake configuration file you can do this with `include_directories` by adding `SYSTEM` before its parameter [19]. There are trade-offs. Silencing warnings from system headers and third party libraries may hide vulnerabilities in them that affect the application. On the other hand, *not* silencing them focuses efforts on issues that the developer typically cannot control, impede progress when using `-Werror` in CI jobs, and often make it difficult to support building with older versions of third party code, making incremental upgrades difficult.

Compile-time checks enabled by options in Table 1 do not have an impact on the binary code generated by the compiler and consequently do not incur any tradeoffs in terms of performance or other run-time characteristics. Rather, they only issue warnings (or errors if the `-Werror` option is enabled) that inform of potential defects found in the source code.

When such additional warnings are enabled, developers should take time to understand the underlying issues that are flagged by the compiler and address them.

The options in Table 2 can be categorized into two types:

1) options that cause the compiler to augment the produced binary with run-time checks aimed to detect memory errors, and 2) options that direct the compiler to adjust the properties of the generated binary or code to ensure the resulting binary is compatible with OS-enforced protection mechanisms.

Testing is essential to validate the impact of enabling any of the options listed in Table 2, as they affect the binary produced by the compiler. Some of the compiler options described below may influence the software's performance. However, this performance impact is usually context-specific, and in most cases, it is either minimal or the benefits outweigh the overhead. For further information

on when significant performance impacts may occur, you can find detailed descriptions of these options later in this document.

When dealing with software for which performance is a critical factor developers should carefully assess the trades-offs between enabling more secure options and observed performance test data, taking into account the specific use cases of their software. Before implementing any changes in production environments, it is essential to conduct thorough benchmarking and testing. This will provide insights into how the compiler options influence both performance and security aspects of the software. Keep in mind that a system that works quickly but is vulnerable to adversaries is likely to be unacceptable to users. Benchmarks should consider any relevant performance characteristics such as average time, worst-case time, and memory use during execution. Additionally, the impact on the size of the produced binaries can be a concern, particularly for embedded systems.

Table 1: Recommended compiler options that enable strictly compile-time checks.

Compiler Flag	Supported since	Description
`-Wall` `-Wextra`	GCC 2.95.3 Clang 4.0	Enable warnings for constructs often associated with defects
`-Wformat=2`	GCC 2.95.3 Clang 4.0	Enable additional format function warnings
`-Wconversion` `-Wsign-conversion`	GCC 2.95.3 Clang 4.0	Enable implicit conversion warnings
`-Wtrampolines`	GCC 4.3	Enable warnings about trampolines that require executable stacks
`-Wimplicit-fallthrough`	GCC 7 Clang 4.0	Warn when a switch case falls through
`-Werror` `-Werror=<warning-flag>`	GCC 2.95.3 Clang 2.6	Treat all or selected compiler warnings as errors. Use the blanket form `-Werror` only during development, not in source distribution.

Table 2: Recommended compiler options that enable run-time protection mechanisms.

Compiler Flag	Supported since	Description
`-D_FORTIFY_SOURCE=3` (requires `-O1` or higher, may require prepending `-U_FORTIFY_SOURCE`)	GCC 12.0 Clang 9.0.0[2]	Fortify sources with compile- and run-time checks for unsafe libc usage and buffer overflows. Some fortification levels can impact performance.
`-D_GLIBCXX_ASSERTIONS` `-D_LIBCPP_ASSERT`	libstdc++ 6.0 libc++ 3.3.0	Precondition checks for C++ standard library calls. Can impact performance.
`-fstrict-flex-arrays=3`	GCC 13 Clang 16.0.0	Consider a trailing array in a struct as a flexible array if declared as `[]`
`-fstack-clash-protection`	GCC 8 Clang 11.0.0	Enable run-time checks for variable-size stack allocation validity. Can impact performance.
`-fstack-protector-strong`	GCC 4.9.0 Clang 5.0.0	Enable run-time checks for stack-based buffer overflows. Can impact performance.
`-fcf-protection=full`	GCC 8 Clang 7.0.0	Enable control flow protection to counter Return Oriented Programming (ROP) and Jump Oriented Programming (JOP) attacks on many x86 architectures
`-mbranch-protection=standard`	GCC 9 Clang 8	Enable branch protection to counter Return Oriented Programming (ROP) and Jump Oriented Programming (JOP) attacks on AArch64
`-Wl,-z,nodlopen`	Binutils 2.10	Restrict `dlopen(3)` calls to shared objects
`-Wl,-z,noexecstack`	Binutils 2.14	Enable data execution prevention by marking stack memory as non-executable
`-Wl,-z,relro` `-Wl,-z,now`	Binutils 2.15	Mark relocation table entries resolved at load-time as read-only. `-Wl,-z,now` can impact startup performance.
`-fPIE -pie`	Binutils 2.16 Clang 5.0.0	Build as position-independent executable. Can impact performance on 32-bit architectures.

Compiler Flag	Supported since	Description
`-fPIC -shared`	< Binutils 2.6 Clang 5.0.0	Build as position-independent code. Can impact performance on 32-bit architectures.
`-fno-delete-null-pointer-checks`	GCC 3.0 Clang 7.0.0	Force retention of null pointer checks
`-fno-strict-overflow`	GCC 4.2	Integer overflow may occur
`-fno-strict-aliasing`	GCC 2.95.3 Clang 18.0.0	Do not assume strict aliasing
`-ftrivial-auto-var-init`	GCC 12 Clang 8.0	Perform trivial auto variable initialization

3.1. Enable warnings for constructs often associated with defects

Compiler Flag	Supported since	Description
`-Wall` `-Wextra`	GCC 2.95.3 Clang 4.0	Enable warnings for constructs often associated with defects

3.1.1. Synopsis

Warnings are compile-time diagnostics messages that indicate programming constructs that, while not inherently erroneous, are risky or suggest a programming error may have been made.

The `-Wall` and `-Wextra` compiler flags enable pre-defined sets of compile-time warnings.

The warnings in the `-Wall` set are generally easy to avoid or can be easily prevented by modifying the offending code.

The `-Wextra` set of warnings are either situational, or indicate problematic constructs that are harder to avoid and in some cases may be necessary.

NOTE: Despite its name the `-Wall` options does NOT enable all possible warning diagnostics, but a pre-defined subset. For a complete list of specific warnings enabled by the `-Wall` and `-Wextra` compiler please consult the GCC[20] and Clang[21] documentation respectively.

3.2. Enable additional format function warnings

Compiler Flag	Supported since	Description
`-Wformat=2`	GCC 2.95.3 Clang 4.0	Enable additional format function warnings

3.2.1. Synopsis

Check calls to the `printf` and `scanf` family of functions to ensure that the arguments supplied have types appropriate to the format string specified, and that the conversions specified in the format string make sense.

The `-Wformat=2` form of the option also enables certain additional checks, including:

- Warning if the format string is not a string literal and so cannot be checked, unless the format function takes its format arguments as a va_list (`-Wformat-nonliteral`).
- Warning about uses of format functions that represent possible security problems (`-Wformat-security`).
- Warning about `strftime` formats that may yield only a two-digit year (`-Wformat-y2k`).

3.3. Enable implicit conversion warnings

Compiler Flag	Supported since	Description
`-Wconversion` `-Wsign-conversion`	GCC 2.95.3 Clang 4.0	Enable implicit conversion warnings

3.3.1. Synopsis

Check for implicit conversions that may alter a value such as:

- conversions between real and integer data types
- conversion between signed and unsigned data types
- conversion between data types of different size
- confusing overload resolution for user-defined conversions in C++
- conversions that never use a type conversion operator in C++: conversions to void, the same type, a base class or a reference.

Conversion between data types that cause the value of the data to be altered can cause information to be omitted or translated in a way that produces unexpected values.

If the resulting values are used in a context where they control memory accesses or security decisions, then dangerous behaviors may occur, e.g., integer signedness or truncation errors can cause buffer

overflows.

For C++ warnings about conversions between signed and unsigned integers are disabled by default unless `-Wsign-conversion` is explicitly enabled.

3.4. Enable warning about trampolines that require executable stacks

Compiler Flag	Supported since	Description
`-Wtrampolines`	GCC 4.3	Enable warnings about trampolines that require executable stacks

3.4.1. Synopsis

Check whether the compiler generates trampolines for pointers to nested functions which may interfere with stack virtual memory protection (non-executable stack.)

A trampoline is a small piece of data or code that is created at run time on the stack when the address of a nested function is taken and is used to call the nested function indirectly.

For most target architectures, including 64-bit x86, trampolines are made up of code and thus requires the stack to be made executable for the program to work properly. This interferes with the non-executable stack mitigation which is used by all major operating system to prevent code injection attacks (see Section 2.10).

3.5. Warn about implicit fallthrough in switch statements

Compiler Flag	Supported since	Description
`-Wimplicit-fallthrough`	GCC 7 Clang 4.0	Warn when a switch case falls through

3.5.1. Synopsis

Warn when an implicit fallthrough occurs in a switch statement that has not been specifically marked as intended.

The `switch` statement in C and C++ allows a case block to fall through to the following case block (unless preceded by break, return, goto, or similar). This is widely considered a design defect in C, because a common mistake is to have a fallthrough occur when it was *not* intended[22].

This warning flag warns when a fallthrough occurs unless it is specially marked as being *intended*. The Linux kernel project uses this flag; it led to the discovery and fixing of many bugs[23].

This warning flag does not have a performance impact. However, sometimes a fallthrough *is* intentional. This flag requires developers annotate those (rare) cases in the source code where a fallthrough *is* intentional, to suppress the warning. Obviously, this annotation should *only* be used when it is intentional. C++17 (or later) code should simply use the attribute `[[fallthrough]]` as it is standard (remember to add `;` after it).

The C17 standard[24] does not provide a mechanism to mark intentional fallthroughs. Different tools support different mechanisms for marking one, including attributes and comments in various forms[25]. A portable way to mark one, used by the Linux kernel version 5.10 and later, is to define a keyword-like macro named `fallthrough` to mark an intentional fallthrough that adjusts to the relevant tool (e.g., compiler) mechanism:

```
#if __has_attribute(__fallthrough__)
# define fallthrough                    __attribute__((__fallthrough__))
#else
# define fallthrough                    do {} while (0)  /* fallthrough */
#endif
```

3.6. Treat compiler warnings as errors

Compiler Flag	Supported since	Description
`-Werror` `-Werror= <warning-flag>`	GCC 2.95.3 Clang 2.6	Treat compiler warnings as errors

3.6.1. Synopsis

Make the compiler treat all or specific warning diagnostics as errors.

The blanket form: `-Werror` without a selector treats all warnings as errors and can be used to implement a zero-warning policy during development. However, we recommend developers to omit the blanket form when distributing source code as it creates a dependency on specific toolchain vendors and versions [26]. If necessary, such toolchain dependencies, i.e., which compiler version(s) the project is expected to work with, should be clearly noted in the project documentation or the build environment should be completely captured, e.g., via container recipes. However, it's better to ensure that source code is not so dependent on a specific toolchain version.

The selective form: `-Werror= <warning-flag>` can be used for refined warnings-as-error control without introducing a blanket zero-warning policy. This is beneficial to ensure that certain undesirable constructs or defects *never* occur produced builds. The selective form does *not* introduce a dependency on the toolchain or version if the corresponding warning is for a specific construct. For example, developers can decide to promote warnings that indicate interference with OS defense

mechanisms (e.g., `-Werror=trampolines`), undefined behavior (e.g., `-Werror=return-type`), or constructs associated with software weaknesses (e.g., `-Werror=conversion`) to errors.

Zero-warning policies can also be enforced at CI level. CI-based zero- or bounded-warning policies are often preferable, for the reasons explained above, and because they can be expanded beyond compiler warnings. For example, they can also include warnings from static analysis tools or generate warnings when `FIXME` and `TODO` comments are found.

3.7. Fortify sources for unsafe libc usage and buffer overflows

Compiler Flag	Supported since	Description
`-D_FORTIFY_SOURCE=1`	GCC 4.0 Clang 5.0.0	Fortify sources with compile- and run-time checks for unsafe libc usage and buffer overflows
`-D_FORTIFY_SOURCE=2` (requires `-O1` or higher)	GCC 4.0 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
`-D_FORTIFY_SOURCE=3` (requires `-O1` or higher)	GCC 12.0 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

3.7.1. 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[27]:

> 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.

To benefit from `_FORTIFY_SOURCE` checks the following requirements must be met:

- the application must be built with `-O1` optimizations or higher; at least `-O2` is recommended.
- the compiler should be able to estimate sizes of the destination buffers at compile time. This can be facilitated by applications and libraries by using function attribute extensions supported by GCC and Clang[28].
- the application code must use glibc versions of the aforementioned functions (included with standard headers, e.g. `<stdio.h>` and `<string.h>`)

If checks added by `_FORTIFY_SOURCE` detect unsafe behavior at run-time they will print an error message and terminate the application.

A default mode for FORTIFY_SOURCE may be predefined for a given compiler, for instance GCC shipped with Ubuntu 22.04 uses FORTIFY_SOURCE=2 by default. If a mode of FORTIFY_SOURCE is set on the command line which differs from the default, the compiler warns about redefining the FORTIFY_SOURCE macro. To avoid this, the predefined mode can be unset with -U_FORTIFY_SOURCE before setting the desired value.

3.7.2. Performance implications

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

3.7.3. 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[28] 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[29] 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.

3.7.4. Additional Considerations

- Applications that incorrectly use `malloc_usable_size` [30] 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* [30]. 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` [31]. 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`.

3.8. Precondition checks for C++ standard library calls

Compiler Flag	Supported since	Description
`-D_GLIBCXX_ASSERTIONS`	libstdc++ 6.0	(C++ using libcstdc++ only) Precondition checks for libstdc++ calls; can impact performance.

3.8.1. Synopsis

The C++ standard library implementation in GCC (libstdc++) provides run-time precondition checks for C++ standard library calls, such as bounds-checks for C++ strings and containers, and null-pointer checks when dereferencing smart pointers.

These precondition checks can be enabled by defining the `-D_GLIBCXX_ASSERTIONS` macro when compiling C++ code that calls into libstdc++[32].

3.8.2. Performance implications

Most calls into the C++ standard library have preconditions. Some preconditions can be checked in constant-time, others are more expensive. The checks enabled by `-D_GLIBCXX_ASSERTIONS` are intended to be lightweight[33], i.e., constant-time checks but the exact behavior can differ between standard library versions. In some versions of libstdc++ the `-D_GLIBCXX_ASSERTIONS` macro can have a non-trivial impact on performance. Slowdowns of up to 6% have been reported[34].

3.8.3. When not to use?

`-D_GLIBCXX_ASSERTIONS` is recommended for C++ applications that may handle untrusted data, as well as for any C++ application during testing.

This option is unnecessary for security for applications in production that only handle completely trusted data.

3.9. Enable strict flexible arrays

Compiler Flag	Supported since	Description
`-fstrict-flex-arrays=3`	GCC 13 Clang 16.0.0	Consider trailing array (at the end of struct) as flexible array only if declared as `[]`
`-fstrict-flex-arrays=2`	GCC 13 Clang 15.0.0	Consider trailing array as a flexible array only if declared as `[]`, or `[0]`
`-fstrict-flex-arrays=1`	GCC 13 Clang 15.0.0	Consider trailing array as a flexible array only if declared as `[]`, `[0]`, or `[1]`
`-fstrict-flex-arrays=0`	GCC 13 Clang 15.0.0	Consider any trailing array (at the end of a struct) a flexible array (the default)

3.9.1. Synopsis

Modify what the compiler determines is a trailing array. The higher levels make the compiler respect the sizes of trailing arrays more strictly[35] (this affects bounds checking)[36].

By default, GCC and Clang treat all trailing arrays (arrays that are placed as the last member or a structure) as flexible-sized arrays, *regardless* of *declared* size for the purposes of `__builtin_object_size()` calculations used by `_FORTIFY_SOURCE` [37]. This disables various bounds checks that do not always need to be disabled. For example, with the default settings, given:

```
struct trailing_array {
    int a;
    int b;
    int c[4];
};
struct trailing_array *trailing;
```

The value of `__builtin_object_size(trailing->c, 1)` is `-1` ("unknown size"), inhibiting bounds checking. The rationale for this default behavior is to allow for the "struct hack" idiom that allows for trailing arrays to be treated as variable sized (regardless of their declared size)[36].

The `-fstrict-flex-arrays` option makes the compiler respect the sizes of trailing array member more strictly. This allows bounds checks added by instrumentation such as `_FORTIFY_SOURCE` or `-fsanitize=bounds` [38] to be able to correctly determine the size of trailing arrays.

The tradeoff is that code that relies on the "struct hack" for arbitrary sized trailing arrays may break as a result[39]. Such code may need to be modified to clearly state that it does not have a specific bound.

The C99 flexible array notation `[]` is the standards-based approach for notating when an array bound is not specifically stated. However, some codebases use the GCC zero-length array extension `[0]`, and some codebases use a one-sized array `[1]` to indicate a flexible array member. Option values `1` and `2` were created so programs that use `[0]` and `[1]` for such cases can have some bounds-checking without modifying their source code.[40]

In this guide we recommend using the standard C99 flexible array notation `[]` instead of non-standard `[0]` or misleading `[1]`, and then using `-fstrict-flex-arrays=3` to improve bounds checking in such cases. In this case, code that uses `[0]` for a flexible array will need to be modified to use `[]` instead. Code that uses `[1]` for a flexible arrays needs to be modified to use `[]` and also extensively modified to eliminate off-by-one errors. Using `[1]` is not just misleading[41], it's error-prone; beware that *existing* code using `[1]` to indicate a flexible array may *currently* have off-by-one errors[42].

Once in place, bounds-checking can occur in arrays with fixed declared sizes at the end of a struct. In addition, the source code unambiguously indicates, in a standard way, the cases where a flexible array is in use. There is normally no significant performance trade-off for this option (once any necessary changes have been made).

3.10. Enable run-time checks for variable-size stack allocation validity

Compiler Flag	Supported since	Description
`-fstack-clash-protection`	GCC 8 Clang 11.0.0	Enable run-time checks for variable-size stack allocation validity
`-param stack-clash-protection-guard-size=` *<gap size>*	GCC 8 Clang 11.0.0	Set the stack guard gap size used to determine the probe granularity of the instrumented code

3.10.1. Synopsis

Stack clash protection mitigates attacks that aim to bypass the operating system's *stack guard gap*. The stack guard gap is a security feature in the Linux kernel that protects processes against sequential stack overflows that overflow the stack in order to corrupt adjacent memory regions.

To avoid the stack guard gap from being bypassed each fresh allocation on the stack needs to probe the freshly allocated memory for the stack guard gap if it is present. Stack clash protection ensures a single allocation may not be larger than the stack guard gap size and the compiler translates larger allocations into a series of smaller sub-allocations. In addition, it ensures that any series of sub-allocations cannot exceed the stack guard gap size without an intervening probe.

Probe instructions can either be implicit or explicit. Implicit probes occur naturally as part of the application's code, such as when x86 and x86_64 call instructions push the return address onto the stack. Implicit probes do not incur any additional performance cost. Explicit probes, on the other hand, consists of additional probe instructions emitted by the compiler.

3.10.2. Performance implications

Applications for which functions allocate at most the size of the stack guard gap of stack space memory at a time do not exhibit adverse performance impact from stack clash protection.

However, stack clash protection may cause performance degradation for applications that perform large allocations that exceed the stack guard gap size. Performance impact scales with the size of large allocations and number of explicit probes required. The performance degradation can be mitigated by increasing the Linux stack guard gap size controlled via the `vm.heap-stack-gap` sysctl parameter) and compiling the application with the corresponding `-param stack-clash-protection-guard-size` . Higher values reduce the number of explicit probes, but a value larger than the kernel guard gap will leave code vulnerable to stack clash style attacks.

Note that `vm.heap-stack-gap` expresses the gap as multiple of page size whereas `stack-clash-protection-guard-size` is expressed as a power of two in bytes. Hence for `vm.heap-stack-gap=256` on x86 (256 * 4KiB = 1MiB gap) the corresponding `stack-clash-protection-guard-size` is 20 (2^20 = 1MiB gap).

3.11. Enable run-time checks for stack-based buffer overflows

Compiler Flag	Supported since	Description
`-fstack-protector-strong`	GCC 4.9.0 Clang 5.0.0	Enable run-time checks for stack-based buffer overflows using strong heuristic
`-fstack-protector-all`	GCC Clang	Enable run-time checks for stack-based buffer overflows for all functions
`-fstack-protector --param=ssp-buffer-size= <n>`	GCC Clang	Enable run-time checks for stack-based buffer overflows for functions with character arrays if *n* or more bytes

3.11.1. Synopsis

Stack protector instruments code produced by the compiler to detect overflows in buffers allocated on the program stack at run-time (colloquially referred to as *"stack smashing"*).

The detection is based on inserting a *canary* value into the stack frame in the function prologue. The canary is verified against a reference value in the function epilogue. If they differ the runtime calls `__stack_chk_fail()`, which will terminate the offending application.

This mitigates potential control-flow hijacking attacks that may lead to arbitrary code execution by corrupting return addresses stored on the stack.

Out-of-date versions of GCC may not detect or defend against overflows of dynamically-sized local variables such as variable-length arrays or buffers allocated using `alloca()` when compiling for 64-bit Arm (Aarch64) processors[43]. Users of GCC-based toolchains for Aarch64 should ensure they use up-to-date versions of GCC 7 or later that support an alternative stack frame layout that places all local variables below saved registers, with the stack-protector canary between them[44].

Some versions of GCC[45] may not detect or defend against overflows of dynamically-sized local variables such as variable-length arrays or buffers allocated using alloca() when compiling for 64-bit Arm (Aarch64) processors[43]. Users of GCC-based toolchains for Aarch64 should, before depending on it, determine if they support an alternative stack frame layout that places all local variables below saved registers, with the stack-protector canary between them[44].

3.11.2. Performance implications

Stack protector supports three different heuristics that are used to determine which functions are instrumented with run-time checks during compilation:

- `-fstack-protector-strong` [46]: instrument any function that
  - takes the address of any of its local variables on the right-hand-side of an assignment or as part of a function argument
  - allocates a local array, regardless of type or length
  - allocates a local struct or union which contains an array, regardless of the type of length of the array
  - has explicit local register variables
- `-fstack-protector` : instrument functions that call alloca() or allocate character arrays of n bytes or more in size . The threshold for instrumentation is adjustable via the `--param=ssp-buffer-size=` *n* option (default: 8 bytes).
- `-fstack-protector-all` : instrument all functions.

The performance overhead is dependent on the number of function's instrumented and the frequency at which instrumented functions are activated at run-time. Enabling `-fstack-protector-strong` is recommended as it provides a good balance between function coverage and performance. Projects

using older compiler versions can consider `-fstack-protector-all` or `-fstack-protector` with a stricter threshold, e.g. `--param=ssp-buffer-size=4` .

3.11.3. When not to use?

`-fstack-protector-strong` is recommended for all applications with conventional stack behavior. Applications with hand-written assembler optimization that make assumptions about the layout of the stack may be incompatible with stack-protector functionality.

3.12. Implement control flow integrity checks

Compiler Flag	Supported since	Description
`-fcf-protection=full`	GCC 8 Clang 7.0.0	Enable control flow protection to counter Return Oriented Programming (ROP) and Jump Oriented Programming (JOP) attacks on many x86 architectures
`-mbranch-protection=standard`	GCC 9 Clang 8	Enable branch protection to counter Return Oriented Programming (ROP) and Jump Oriented Programming (JOP) attacks on AArch64

3.12.1. Synopsis

Return-oriented programming (ROP) uses an initial subversion (such as a buffer overflow) to perform an indirect jump that executes a different sequence of instructions. This is often existing code being misused, so these are often called “code reuse attacks”. A countermeasure is to ensure that jump addresses and return addresses are correct. This is not a complete solution, but it makes attacks harder to perform.

3.12.2. Performance implications

There are performance implications but they are typically mild due to hardware assistance. The `-fcf-protection=full` flag enables Intel’s Control-Flow Enforcement Technology (CET) [47], which introduces shadow stack (SHSTK) and indirect branch tracking (IBT). The `-mbranch-protection=standard` flag invokes similar protections in the AArch64. In clang `-mbranch-protection=standard` is equivalent to `-mbranch-protection=bti+pac-ret` and invokes the AArch64 Branch Target Identification (BTI) and Pointer Authentication using key A (pac-ret) [48].

3.13. Restrict dlopen calls to shared objects

Compiler Flag	Supported since	Description
`-Wl,-z,nodlopen`	Binutils 2.10	Restrict `dlopen(3)` calls to shared objects

3.13.1. Synopsis

The `nodlopen` option passed to the linker when building shared objects will mark the resulting object as not available to `dlopen(3)` calls. This can help in reducing an attacker's ability to load and manipulate shared objects. Loading new objects or duplicating an already existing shared object in a process can constitute a part of the attack chain in runtime exploitation.

The `nodlopen` restrictions are based on setting the `DF_1_NOOPEN` flags in the object's `.dynamic` section tags. Since the enforcement of restricted calls is done inside libc when `dlopen(3)` are called it is possible for attackers to bypass the check by 1) manipulating the tag embedded in the object if they have the ability to modify the object file on disk, or 2) bypassing `dlopen(3)` and loading shared objects through attacker controlled code, e.g., pieces of shellcode or return-oriented-programming gadgets. However, restrictions on `dlopen(3)` put in place at link time can still be useful in restricting the attacker before they have obtained arbitrary code execution capabilities.

3.13.2. Performance implications

None, marking shared objects as restricted to `dlopen(3)` does not have an impact on performance at run time.

3.13.3. When not to use?

In some cases it is desirable for applications to manage the loading of libraries directly via `dlopen(3)` without relying on the conventional dynamic linking. Such situations include:

- Selecting application plugins to load
- Selecting a version of a library optimized for particular CPUs. Leveraged by, e.g., math libraries that provide different implementations of mathematical operations for different environments.
- Selecting an implementation of an API by different vendors
- Delay loading of shared libraries to decrease application start times. (See also lazy binding in Section 2.11)

Since `nodlopen` interferes with applications that rely on to `dlopen(3)` to manipulate shared objects they cannot be used with applications that rely on such functionality.

3.14. Enable data execution prevention

Compiler Flag	Supported since	Description
`-Wl,-z,noexecstack`	Binutils 2.14	Enable data execution prevention by marking stack memory as non-executable

3.14.1. Synopsis

All major modern processor architectures incorporate memory management primitives that give the OS the ability to mark certain memory areas, such as the stack and heap, as non-executable, e.g., the AMD *"non-execute"* (NX) bit and the Intel *"execute disable"* (XD) bit. This mechanism prevents the stack or heap from being used to inject malicious code during a run-time attack.

The `-Wl,-z,noexecstack` option tells the linker to mark the corresponding program segment as non-executable which enables the OS to configure memory access rights correctly when the program executable is loaded into memory.

3.14.2. Performance implications

None, marking the stack and/or heap as non-executable does not have an impact on performance at run time.

3.14.3. When not to use?

Some language-level programming constructs, such as taking the address of a nested function (a GNU C extension to ISO standard C) requires special compiler handling which may prevent the linker from marking stack segments correctly as non-executable[49].

Consequently the `-Wl,-z,noexecstack` option works best when combined with appropriate warning flags (`-Wtrampolines` where available) that indicate whether language constructs interfere with stack virtual memory protection.

3.15. Mark relocation table entries resolved at load-time as read-only

Compiler Flag	Supported since	Description
`-Wl,-z,relro -Wl,-z,now`	Binutils 2.15	Mark relocation table entries resolved at load- time as read-only

"Read-only relocation" (RELRO) marks relocation table entries as read-only after they have been resolved by the dynamic linker/loader (`ld.so`). Relocation is the process performed by `ld.so` that connects unresolved symbolic references to proper addresses of corresponding in-memory objects.

Marking relocations read-only will mitigate run-time attacks that corrupt Global Offset Table (GOT) entries to hijack program execution or to cause unintended data accesses. Collectively such attacks are referred to as *GOT overwrite attacks* or *GOT hijacking*.

RELRO can be instantiated in one of two modes: partial RELRO or full RELRO. Full RELRO is necessary for effective mitigation for GOT overwrite attacks; partial RELRO is not sufficient.

Partial RELRO (`-Wl,-z,relro`) will mark certain ELF sections as read-only after initialization by the runtime loader. These include `.init_array` , `.fini_array` , `.dynamic` , and the non-PLT portion of `.got` . However, in partial RELRO the auxiliary procedure linkage portion of the GOT (`.got.plt`) is still left writable to facilitate late binding.

Full RELRO (`-Wl,-z,relro -Wl,-z,now`) disables lazy binding. This allows `ld.so` to resolve the entire GOT at application startup and mark also the PLT portion of the GOT as read-only.

3.15.1. Performance implications

Since lazy binding is primarily intended to speed up application startup times by spreading out the symbol resolution operations throughout the lifetime of the application, enabling full RELRO can increase the startup time for applications with large numbers of dynamic dependencies. The performance impact scales with the number of dynamically linked functions.

3.15.2. When not to use?

Applications that are sensitive to the performance impact on startup time should consider whether the increase in startup time caused by full RELRO impacts the user experience. As an alternative, developers can consider statically linking large library dependencies to the application executable.

Static linking avoids the need for dynamic symbol resolution altogether but can make it more difficult to deploy patches to dependencies compared to upgrading shared libraries. Developers need to consider whether static linking is discouraged in their deployment scenarios, e.g., major Linux distributions generally forbid static linking of shared application dependencies.

3.16. Build as position-independent code

Compiler Flag	Supported since	Description
`-fPIE -pie`	Binutils 2.16 Clang 5.0.0	Build as position-independent executable.

Compiler Flag	Supported since	Description
`-fPIC -shared`	< Binutils 2.6 Clang 5.0.0	Build as position-independent code.

3.16.1. Synopsis

Position-independent code (PIC) and executables (PIE) are machine code objects that execute properly regardless of the exact address at which they are loaded at in process memory.

GNU/Linux requires program executable to be built as PIE in order to benefit from address-space layout randomization (ASLR). ASLR is the primary means of mitigating code-reuse exploits, e.g., *return-to-libc* and *return-oriented programming* in modern GNU/Linux distributions. In code-reuse exploits the adversary corrupts vulnerable code pointers, such as return addresses stored on the program stack and makes them refer to pre-existing executable code in program memory. ASLR randomizes the location of shared libraries and the program executable every time the object is loaded into memory to make memory addresses useful in exploits harder to predict.

3.16.2. Performance implications

Negligible on 64-bit architectures.

On 32-bit x86 PIC exhibits moderate performance penalties (5-10%)[15]. This is due to data accesses using mov instructions on 32-bit x86 only support absolute addresses. To make the code position-independent memory references are transformed to lookup memory addresses from a global offset table (GOT) populated at load-time with the correct addresses to program data. Consequently, data references require an additional memory load compared to non-PIC code on 32-bit x86. However, the main reason for the performance penalty is the increased register pressure resulting from keeping the lookup address to the GOT available in a register[50].

The x86_64 architecture supports mov instructions that address memory using offsets relative to the instruction pointer (i.e., the address of the currently executing instruction). This is referred to as RIP addressing. PIC on x86_64 uses RIP addressing for accessing the GOT which relieves the register pressure associated with PIC on 32-bit x86 and results in a smaller impact on performance. Shared libraries are created PIC on x86_64 by default[51].

3.16.3. When not to use?

Resource-constrained embedded systems may save memory by *prelinking* executables at compile time. Prelinking performs some relocation decisions, normally made by the dynamic linker, ahead of time. As a result, fewer relocations need to be performed by the dynamic linker, reducing startup time and memory consumption for applications. PIE does not prevent prelinking but enabling ASLR on prelinked binaries overrides the compile-time decisions, thus nullifying the run-time memory savings gained by prelinking. If the memory savings gained by prelinking are important for a system PIE can be enabled for a subset of executables that are at higher risk, e.g., applications that process untrusted external input.

3.17. Do not delete null pointer checks

Compiler Flag	Supported since	Description
`-fno-delete-null-pointer-checks`	GCC 3.0 Clang 7.0.0	Force retention of null pointer checks

3.17.1. Synopsis

If a code defect references a potentially-null pointer, compilers are allowed to remove the null pointer check, under the theory that since developers never make mistakes, the pointer check is unnecessary.

Since developers *do* make mistakes, without this option, the result is that the source code may *appear* to request a check, one that is necessary for security, but the check will *not* be done by the compiled executable. This option is one of several that are recommended and used by various sources to address real-world errors [52].

An example of this defect occurred in the Linux kernel and led to a serious vulnerability. In the following simplified Linux kernel code, the construct `dev->priv` presumes `dev` is non-null. That means that if `dev` is null, we have undefined behavior. In this case, the C compiler presumed that `dev` is not null, and threw away the code `if (!dev) return` ([53] for more):

```
static void __devexit agnx_pci_remove (struct pci_dev *pdev)
{
  struct ieee80211_hw *dev = pci_get_drvdata(pdev);
  struct agnx_priv *priv = dev->priv;

  if (!dev) return;

  ... do stuff using dev ...
}
```

The Linux kernel now enables `-fno-delete-null-pointer-checks`; as explained later by Linux Torvalds [54], "we had buggy code that accessed a pointer before the NULL pointer check, but the bug was "benign" as long as the compiler didn't actually remove the check. ... Removing the NULL pointer check turned a benign bug into a trivially exploitable one by just mapping user space data at NULL ... Removing the NULL pointer check turned a benign bug into a trivially exploitable one by just mapping user space data at NULL (which avoided the kernel oops, and then made the kernel use the user value!)... the kernel generally really doesn't want optimizations that are perhaps allowed by the standard, but that result in code generation that doesn't match the source code."

The option `-fno-delete-null-pointer-checks` forces the retention of such checks even when in theory they are unnecessary, and is in use in the Linux kernel.

3.17.2. Performance implications

There are normally no significant performance implications. Null pointer checks are extremely quick and can often be performed in parallel by the CPU.

3.18. Integer overflow may occur

Compiler Flag	Supported since	Description
`-fno-strict-overflow`	GCC 4.2	Integer overflow may occur

3.18.1. Synopsis

In C and C++ unsigned integers have long been defined as "wrapping around". However, for many years C and C++ have assumed that overflows do not occur in many other circumstances. Overflow when doing arithmetic with signed numbers is considered undefined by many versions of the official specifications, This approach also allows the compiler to assume strict pointer semantics: if adding an offset to a pointer does not produce a pointer to the same object. In practice, this means that important security checks written in the source code may be silently ignored when generating executable code.

For example, here is some code from `fs/open.c` of the Linux kernel [52]:

```
int do_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if ((offset + len > inode->i_sb->s_maxbytes)
        || (offset + len < 0))
        return -EFBIG;
    ...
}
```

A developer *might* expect that the computation `offset + len` would produce a useful value for comparison. However, if the compiler is in `strict-overflow` mode, the compiler is free to determine that `offset + len` is always more than 0, and thus it can omit an important security check.

The Linux kernel enables `-no-strict-overflow` to reduce the likelihood that important security checks in the source code will be silently ignored by the compiler.

An alternative option is to use the `-fwrapv` option. With `-fwrapv`, integer signed overflow wraps (and is thus defined).

Note that GCC and Clang interpret this option slightly differently. On clang, this option is considered a synonym for `-fwrapv`. On GCC, this option does not fully enforce two's complement on signed integers, allowing for additional optimizations. [52]

3.19. Do not assume strict aliasing

Compiler Flag	Supported since	Description
`-fno-strict-aliasing`	GCC 2.95.3 Clang 18.0.0	Do not assume strict aliasing

3.19.1. Synopsis

Pointers can be cast from one type to another. Standards have strict rules for aliasing, requiring that pointers of different types do *not* alias in most cases. However, in practice, many constructs depend on such aliasing even though it is undefined. By default, undefined code can do *anything* and this is undesirable. [52]

This option eliminates this problem. It's used by the Linux kernel.

3.20. Perform trivial auto variable initialization

Compiler Flag	Supported since	Description
`-ftrivial-auto-var-init`	GCC 12 Clang 8.0	Perform trivial auto variable initialization

3.20.1. Synopsis

This option controls if (and how) automatic variables are initialized. Even with the option, the compiler will consider an automatic variable as uninitialized unless it is explicitly initialized.

This option has three choices:

- `uninitialized` - automatic variables are not initialized. This is the default.
- `pattern` - automatic variables are initialized with a value likely to cause a crash if there is a logic bug.
- `zero` - automatic variables are initialized with zeros, to reduce the risk of a logic bug leading to a security vulnerability or other problems.

We recommend using `zero` for production code, to reduce the risk of a logic bug leading to a security vulnerability.

This setting can sometimes interfere with other tools that are being used to monitor executable code, since it is expressly setting a value that was not set by the source code.

4. Discouraged Compiler Options

This section describes discouraged compiler and linker option flags that may lead to potential defects with security implications in produced binaries.

Table 3: List of discouraged compiler and linker options.

Compiler Flag	Supported since	Description
`-Wl,-rpath, path_to_so`	Binutils 2.11	Hard-code run-time search paths in executable files or libraries

4.1. Hard-code run-time search paths in executable files or libraries

Compiler Flag	Supported since	Description
`-Wl,-rpath, path_to_so`	Binutils 2.11	Hard-code run-time search paths in executable files or libraries

4.1.1. Synopsis

The `-rpath` option records the specified path to a shared object files to the `DT_RPATH` or `DT_RUNPATH` header value in the produced ELF binary. The recorded rpath may override or supplement the system default search path used by the dynamic linker to find the specified library dependency.

The rpath provided by the original (and default) `DT_RPATH` entry takes precedence over environmental overrides such as `LD_LIBRARY_PATH`, and an object's `DT_RPATH` can be used for resolving dependencies of another object. These design errors were rectified with `DT_RUNPATH`, which has a lower precedence with respect to `LD_LIBRARY_PATH` and only affects the search path of an object's own, immediate dependencies[55].

Setting either `DT_RPATH` or `DT_RUNPATH` in release binaries may lead to security vulnerabilities under certain conditions. An attacker may be able to supply their own shared files in the target directories and override the operating system's libraries, resulting in arbitrary code execution.

Relative paths (e.g. `.` or `./lib`) are resolved relative to the working directory, which may be set by an attacker to a directory with a malicious dependency.

The keyword `$ORIGIN` in rpath is expanded by the dynamic loader to the path of the directory where the object is found, which may be set by an attacker (e.g., via hard links) to a directory with a malicious dependency. On Linux, the `fs.protected_hardlinks` sysctl can help prevent this attack.

Setting rpath in setuid/setgid programs can lead to privilege escalation under conditions where untrusted libraries loaded via a set rpath are executed as part of the privileged program. While setuid/setgid binaries ignore environmental overrides to search path (such as `LD_PRELOAD`, `LD_LIBRARY_PATH` etc.), rpath within such binaries can provide an attacker with equivalent capabilities to manipulate the dependency search paths.

5. Sanitizers

Sanitizers are a suite of compiler-based tools designed to detect and pinpoint memory-safety issues and other defects in applications written in C and C++. They provide similar capabilities as dynamic analysis tools built on frameworks such as Valgrind. However, unlike Valgrind, sanitizers leverage compile-time instrumentation to intercept and monitor memory accesses. This allows sanitizers to be more efficient and accurate compared to dynamic analyzers. On average, Sanitizers impose a 2× to 4× slowdown in instrumented binaries, whereas dynamic instrumentation can exhibit slowdowns as large as 20× to 50×[56]. As a tradeoff, sanitizers must be enabled at compile time whereas Valgrind can be used with unmodified binaries. Table 4 lists sanitizer options supported by GCC and Clang.

While more efficient compared to dynamic analysis, sanitizers are still prohibitively expensive in terms of performance penalty and memory overhead to be used with Release builds, but excel at providing memory diagnostics in Debug, and in certain cases Test builds. For example, fuzz testing (or "fuzzing") is a common security assurance activity designed to identify conditions that trigger memory-related bugs. Fuzzing is primarily useful for identifying memory errors that lead to application crashes. However, if fuzz testing is performed in binaries equipped with sanitizer functionality it is possible to also identify bugs which do not crash the application. Another benefit is the enhanced diagnostics information produced by sanitizers.

As with all testing practices, sanitizers cannot absolutely prove the absence of bugs. However, when used appropriately and regularly they can help in identifying latent memory, concurrency, and undefined behavior-related bugs which may be difficult to pinpoint.

Table 4: Sanitizer options in GCC and Clang.

Compiler Flag	Supported since	Description
`-fsanitize=address`	GCC 4.8	Enables AddressSanitizer to detect memory errors

Compiler Flag	Supported since	Description
	Clang 3.1	at run-time
`-fsanitize=thread`	GCC 4.8 Clang 3.2	Enables ThreadSanitizer to detect data race bugs at run time
`-fsanitize=leak`	GCC 4.8 Clang 3.1	Enables LeakSanitizer to detect memory leaks at run time
`-fsanitize=undefined`	GCC 4.9 Clang 3.3	Enables UndefinedBehaviorSanitizer to detect undefined behavior at run time

5.1. AddressSanitizer

Compiler Flag	Supported since	Description				
`-fsanitize=address`	GCC 4.8 Clang 3.1	Enables AddressSanitizer to detect memory errors at run-time		`-fsanitize=thread`	GCC 4.8 Clang 3.2	E T t r r

AddressSanitizer (ASan) is a memory error detector that can identify memory defects that involve:

- Buffer overflows in the stack, on the heap, and in global variables
- Use-after-free conditions (dereference of dangling pointers)
- Use-after-return (use of stack memory reserved for locals after return from function)
- Use-after-scope conditions (use of stack address outside the lexical scope of variable)
- Initialization order bugs
- Memory leaks (see also LeakSanitizer in Section 0)

To enable ASan add `-fsanitize=address` to the compiler flags (`CFLAGS` for C, `CXXFLAGS` for C++) and linker flags (`LDFLAGS`). Consider combining ASan with the following compiler flags:

- `-O1` (disables inlining and improves stack traces, higher levels improve performance)
- `-g` (to display source file names and line numbers in the produced error messages)
- `-fno-omit-frame-pointer` (to further improve stack traces)
- `-fno-optimize-sibling-calls` (disable tail call optimizations)
- `-fno-common` (disable common symbols to improve tracking of globals)

The run-time behavior of ASan can be influenced using the `ASAN_OPTIONS` environment variable. The run-time options can be used enable additional memory error checks and to tweak ASan performance. An up-to-date list of supported options are available on the AddressSanitizerFlags article on the project's GitHub Wiki[57]. If set to `ASAN_OPTIONS=help=1` the available options are shown at startup of the instrumented program. This is particularly useful for determining which options are supported by the specific version ASan integrated to the compiler being used. A useful pre-set to enable more aggressive diagnostics compared to the default behavior is given below:

```
ASAN_OPTIONS=strict_string_checks=1:detect_stack_use_after_return=1:
check_initialization_order=1:strict_init_order=1 ./instrumented-executable
```

When ASan encounters a memory error it (by default) terminates the application and prints an error message and stack trace describing the nature and location of the detected error. A systematic description of the different error types and the corresponding root causes reported by ASan can be found in the AddressSanitizer article on the project's GitHub Wiki[58].

ASan cannot be used simultaneously with ThreadSanitizer or LeakSanitizer. It is not possible to mix ASan-instrumented code produced by GCC with ASan-instrumented code produced Clang as the ASan implementations in GCC and Clang are mutually incompatible.

5.2. ThreadSanitizer

Compiler Flag	Supported since	Description
`-fsanitize=thread`	GCC 4.8 Clang 3.2	Enables ThreadSanitizer to detect data race bugs at run time

ThreadSanitizer (TSan) is a data race detector for C/C++. Data races occur when two (or more) threads of the same process access the same memory location concurrently and without synchronization. If at least one of the accesses is a write the application risks entering an inconsistent internal state. If two or more threads attempt to write to the memory location simultaneously a data race may cause memory corruption. Data races are notoriously difficult to debug since the order of accesses is typically non-deterministic and dependent on the precise timing of events in the offending threads.

To enable TSan add `-fsanitize=thread` to the compiler flags (`CFLAGS` for C, `CXXFLAGS` for C++) and linker flags (`LDFLAGS`). Consider combining TSan with the following compiler flags:

- `-O2` (or higher for reasonable performance)
- `-g` (to display source file names and line numbers in the produced warning messages)

The run-time behavior of TSan can be influenced using the `TSAN_OPTIONS` environment variable. An up-to-date list of supported options are available on the ThreadSanitizerFlags article on the project's

GitHub Wiki[59]. If set to `TSAN_OPTIONS=help=1` the available options are shown at startup of the instrumented program.

When TSan encounters a potential data race it (by default) reports the race by printing a warning message with a description of the program state that led to the data race. A detailed description of the report format can be found in the ThreadSanitizerReportFormat article on the project's GitHub Wiki[60].

TSan cannot be used simultaneously with AddressSanitizer (ASan) or LeakSanitizer (LSan). It is not possible to mix TSan-instrumented code produced by GCC with TSan-instrumented code produced Clang as the TSan implementations in GCC and Clang are mutually incompatible. TSan generally requires all code to be compiled with `-fsanitize=thread` to operate correctly.

5.3. LeakSanitizer

Compiler Flag	Supported since	Description
`-fsanitize=leak`	GCC 4.8 Clang 3.1	Enables LeakSanitizer to detect memory leaks at run time

LeakSanitizer (LSan) is a stand-alone version of the memory leak detection built into ASan. It allows analysis of memory leaks without the associated slowdown introduced by ASan. Unlike ASan, LSan does not require compile-time instrumentation, but consists only of a runtime library. The `-fsanitize=leak` option instructs the linker to link the application executable against the LSan library which overrides `malloc()` and other allocator functions.

The run-time behavior of LSan can be influenced using the `LSAN_OPTIONS` environment variable. If set to `LSAN_OPTIONS=help=1` the available options are shown at startup of the program.

LSan cannot be used simultaneously with AddressSanitizer (ASan) or ThreadSanitizer (TSan). If either ASan or TSan is enabled during the build the `-fsanitize=leak` option is ignored by the linker.

5.4. UndefinedBehaviorSanitizer

Compiler Flag	Supported since	Description
`-fsanitize=undefined` (requires `-O1` or higher)	GCC 4.9 Clang 3.3	Enables UndefinedBehaviorSanitizer to detect undefined behavior at run time

UndefinedBehaviorSanitizer (UBSan) is a detector of non-portable or erroneous program constructs which cause behavior which is not clearly defined in the ISO C standard. UBSan provides a large number of sub-options to enable / disable individual checks for different classes of undefined behavior. Consult the GCC[61] and Clang[62] documentation respectively for up-to-date information on supported sub-options.

To enable UBSan add `-fsanitize=undefined` to the compiler flags (`CFLAGS` for C, `CXXFLAGS` for C++) and linker flags (`LDFLAGS`) together with any desired sub-options. Consider combining TSan with the following compiler flags:

- `-O1` (required or higher for reasonable performance)
- `-g` (to display source file names and line numbers in the produced warning messages)

The run-time behavior of UBSan can be influenced using the `UBSAN_OPTIONS` environment variable. If set to `UBSAN_OPTIONS=help=1` the available options are shown at startup of the instrumented program.

6. Maintaining debug information in separate files

An application's debugging information can be placed in a debug info file separate from the application's executable. This allows the executable to be shipped stripped of debug information while still allowing a debugger to obtain the debugging information from the debug info files when problems in the release executable are being diagnosed. Both the GNU Debugger (GDB) and LLVM Debugger (LLDB) allows debug information for stripped binaries to be loaded from separate debug info files.

There are several reasons why developers may wish to separate the debug information from the executable:

- Debug information can be very large – in some cases even larger than the executable code itself! If separate, it can be omitted where it is not needed. For this reason, most Linux distributions distribute debug information for application packages in separate debug info files.
- It avoids inadvertently revealing some sensitive implementation details about the application if its source code is not available. The availability of symbol information makes binary analysis and reverse engineering of the application's executable easier. However, tools like decompilers can work without debug information, so the security of a system must *not* depend on omitting such information.

The following series of commands generate the debug info file, strip the debugging information from the main executable, and add the debug link section.

```
objcopy --only-keep-debug executable_file executable_file.debug
objcopy --strip-unneeded executable_file
objcopy --add-gnu-debuglink=executable_file.debug executable_file
```

6.1. Debug information in the ELF binary format

In ELF binaries debug and symbol information are stored in discrete ELF sections unless separate debug info files are created. Table 5 shows the ELF sections which normally contain debug, symbol or other auxiliary information.

Elf Section	Description
`.debug`	Symbolic debug information for debuggers (typically in DWARF format[63])
`.comment`	GCC version information
`.dynstr`	Strings needed for dynamic symbol name lookup via .dynsym
`.dynsym`	Dynamic symbol lookup table used for run-time relocations
`.note`	Auxiliary metadata, e.g, ABI tags[64] and Build ID[65]
`.strtab`	Strings representing names in `.symtab`
`.symtab`	Global symbol table used for symbol name lookup by debuggers

Whether a particular section is present or absent in an ELF binary indicates what type of information is available. The availability of symbol information makes binary analysis easier as debuggers, disassemblers and binary code analysis tools, such as Ghidra[66] and IDA Pro[67], can use available symbol information to automatically annotate decompiled machine code. Similarly, the availability of debug information makes dynamic analysis of the application in a debugger easier. Stripping unnecessary debug and symbol information from the binary does not make it impervious against reverse engineering, however it does considerably increase the cost and manual effort required for successful exploitation.

6.2. Creating debug info files

The debug info files are ordinary executables with an identical section layout as the application's original executable, but without the executable's data. The debug info file is created by compiling the application executable with the desired debug information included, then processing the executable with the `objcopy` utility to produce the stripped executable (without debugging information) and the debug info file (without executable data). Both GNU binutils `objcopy` [68] and LLVM `llvm-objcopy` [69] support the same options for stripping debug information and creating the debug info file. The shell snippet below shows the `objcopy` invocation for creating a debug info file from an executable with debug information.

```
objcopy --only-keep-debug executable_file executable_file.debug
```

There are no particular requirements for the debug link filename, although a common convention is to name debug info for an executable, e.g., "executable.debug". While the debug info file can have the

same name as the executable it is preferred to use an extension such as ".debug" as it means that the debug info file can be placed in the same directory as the executable.

Debug info files allows the binary to be analyzed in the same way as the original binary with debug and symbol information intact. They should be handled with care and not exposed in computing environments where they may be obtained by adversaries.

6.3. Strip debug and symbol information

Once the debug info file has been created the debug and symbol information can be stripped from the original binary using either the `objcopy` or `strip` [70] utilities provided by Binutils, or the `llvm-objcopy` or `llvm-strip` [71] equivalents provided by LLVM. The shell snippets below show how the debug and unneeded symbol information can be removed from an executable using `objcopy` and `strip` respectively. If code signing is enforced on the application binaries the debug and symbol information must be stripped away before the binaries are signed.

```
strip --strip-unneeded executable_file
```

```
objcopy --strip-unneeded executable_file
```

The `--strip-unneeded` option in `objcopy` and will remove all symbol information (ELF `.symtab` and `.strtab` sections) from the binary that is not needed for processing relocations. In addition, it will trigger the removal of any symbolic debug information from the binary (ELF `.debug` sections and all sections with the `.debug` prefix).

Removing symbol information used for relocations is discouraged as it may interfere with resolving dynamically linked symbols (ELF `.dynsym` and `.dynstr` sections) and Address Space Layout Randomization (ASLR) at run-time. As a result, it should be expected that debuggers and binary analysis will be able to resolve calls to dynamically linked functions to the correct symbol information. Static linking can be considered as an alternative where applicable to avoid dynamically linked symbols to remain visible in resulting binaries.

Stripping additional sections

Note that `--strip-unneeded` only discards standard ELF sections as unneeded. Since an ELF binary can have any number of additional sections which are unknown to `objcopy` and `strip` they cannot determine whether such unrecognized sections are safe to remove. This includes for example the `.comment` section added by GCC. The shell snippets below show how non-standard sections, such as `.comment` can be removed in addition to the unneeded sections identified by `--strip-unneeded`. If the application includes custom, application-specific ELF sections with possible sensitive diagnostics information or metadata which is not required at run-time during normal operations developers may wish to strip such additional sections from release binaries.

```
objcopy --strip-unneeded --remove-section=.comment executable_file

strip --strip-unneeded --remove-section=.comment executable_file
```

6.4. Add a debug link to the binary

To allow the debugger to identify the correct debug information the executable must be associated with its corresponding debug info file. This can be done in two ways:

- Include a "debug link" within the executable that specifies the name of the corresponding debug info file.
- Include a "build ID", a unique bit string, within the executable from which the debug info file's name can be derived.

In most cases the debug link is preferrable as it allows the developers to name the debug info file and verifies a checksum over the debug information files content before the symbol information is sourced from the file during debugging.

Debug link

A debug link is a special section (`.gnu_debuglink`) in the executable file that contains the name of the corresponding debug info file and a 32-bit cyclic redundancy checksum (CRC) computed over the debug info file's full contents. Any executable file format can carry debug link information as long is can contain a section named `.gnu_debuglink` . The shell snippet below shows how a debug link can be added to an executable using `objcopy` (or `llvm-objcopy`).

```
objcopy --add-gnu-debuglink=executable_file.debug executable_file
```

If the debug information file is built in one location but is going to be later installed at a different location the `--add-gnu-debuglink` option should be used with the path to the built debug information file. The debug info file must exist at the specified path as it is required for the CRC calculation which allows the debugger to validate that the debug info file it loads matches that of the executable.

Note that `.gnu_debuglink` does not contain the full pathname to the debug info; only a filename with the leading directory components removed. GDB looks for the debug info file with the specified filename in a series of search directories starting from the directory where the executable is placed. For a complete list of search paths refer to the GDB documentation[72].

Build ID

A build ID is a unique bit string stored in `.note.gnu.build-id` of the ELF `.note` section that is (statistically) unique to the binary file. A debugger can use the build ID to identify the corresponding debug info file if the same build ID is also present in the debug info file.

If the build ID method is used the debug info file's name is computed from the build ID. GDB searches the global debug directories (typically `/usr/lib/debug`) for a `.build-id/xx/yyyy.debug` file, where `xx` are the first two hex characters of the build ID and `yyyy` are the rest of the build ID bit string in hex (actual build ID strings are 32 or more hex characters).

Note that the build ID does not act as a checksum for the executable or debug info file. For more information on the build ID feature please refer to the GDB[68] and GNU linker[65] documentation.

7. Contributors

The OpenSSF Developer BEST Practices Working group thanks Ericsson for their generous initial donation of content to start collaboration on this guide.

- Thomas Nyman, Ericsson
- Robert Byrne, Ericsson
- Jussi Auvinen, Ericsson
- Christopher "CRob" Robinson, Intel
- David A. Wheeler, Linux Foundation
- David Edelsohn, IBM
- Gabriel Dos Reis, Microsoft
- Georg Kunz, Ericsson
- George Wilson, IBM
- Jack Kelly, ControlPlane
- Kees Cook, Google
- Mark Esler, Canonical
- Randall T. Vasquez, Linux Foundation
- Robert C. Seacord, Woven by Toyota
- Siddharth Sharma, Red Hat
- Siddhesh Poyarekar, Red Hat
- William Huhn, Intel

8. License

9. Appendix: List of Considered Compiler Options

Many more security-relevant compiler options exist than are recommended in this guide. Some of these have been considered for inclusion, but for various reasons have, in-the-end been excluded from the set of recommended options. The following table lists options that have been reviewed and the

rationale for their exclusion. While they are not in the recommended list, you may find them useful for your purposes.

Compiler Flag	Supported since	Rationale
`-Wl,-z,nodump`	Binutils 2.10	Single-purpose feature for Solaris compatibility[73].
`-Wl,-z,noexecheap`	Binutils 2.15 (Hardened Gentoo / PaX only)	Hardened Gentoo / PaX specific Binutils extension[74], not present in upstream toolchains.
`-D_LIBCPP_ASSERT`	libc++ 3.3.0	Deprecated in favor of `_LIBCPP_ENABLE_HARDENED_MODE` [75]
`-D_LIBCPP_ENABLE_ASSERTIONS`	libc++ 3.3.0	Deprecated in favor of `_LIBCPP_ENABLE_HARDENED_MODE` [75]
`-mshstk`	GCC 8 Clang 6.0	Enables discouraged shadow stack built-in functions[76], which are only needed for programs with an unconventional management of the program stack. CET instrumentation is controlled by `-fcf-protection` .
`-fsanitize=safe-stack`	Clang 4.0	Known compatibility limitations with garbage collection, signal handling, and shared libraries[77].

10. References

1. Such as the Intel C/C++ Compiler, the Arm Compiler for Embedded, the Apple Clang compiler included in Xcode, IBM Open XL C/C++ Compiler, the Red Hat Developer Toolset, the Siemens Sourcery Toolchain, and AdaCore GNAT Pro Enterprise. ↩

2. The implementation of `-D_FORTIFY_SOURCE={1,2,3}` in the GNU libc (glibc) relies heavily on implementation details within GCC. Clang implements its own style of fortified function calls (originally introduced for Android's bionic libc) but as of Clang / LLVM 14.0.6 incorrectly produces non-fortified calls to some glibc functions with `_FORTIFY_SOURCE` . Code set to be fortified with Clang will still compile, but may not always benefit from the fortified function variants in glibc. For more information see: Guelton, Serge, Toward _FORTIFY_SOURCE parity between Clang and GCC. Red Hat Developer, Red Hat Developer, 2020-02-11 and Poyarekar, Siddhesh, D91677 Avoid

simplification of library functions when callee has an implementation, LLVM Phabricator, 2020-11-17. ↩ ↩2 ↩3 ↩4

3. Cimpanu, Catalin, Microsoft: 70 percent of all security bugs are memory safety issues, ZDNet, 2019-02-11 ↩

4. Cimpanu, Catalin, Chrome: 70% of all security bugs are memory safety issues, ZDNet, 2020-05-22 ↩

5. Carnegie Mellon University (CMU), SEI CERT C Coding Standard Rules for Developing Safe, Reliable, and Secure Systems, 2016 edition, June 2016. ↩

6. Carnegie Mellon University (CMU), SEI CERT C++ Coding Standard Rules for Developing Safe, Reliable, and Secure Systems, 2016 edition, March 2017. ↩

7. US NIST, Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities, NIST SP 800-218, February 2018. ↩

8. Carnegie Mellon University (CMU), Top 10 Secure Coding Practices, SEI CERT Coding Standards Wiki, 2018-05-02. ↩

9. Wheeler, David, Initial Analysis of Underhanded Source Code, Institute for Defense Analysis, April 2020. ↩ ↩2

10. Wikipedia contributors, Trojan Source, Wikipedia, 2023-11-30. ↩

11. Software in the Public Interest, Hardening in Debian, Debian Wiki, 2022-01-07. ↩

12. Gentoo Foundation, Hardening in Gentoo, Gentoo Wiki, 2023-03-08. ↩

13. Red Hat, Using RPM build flags in Fedora, Fedora Package Sources (`redhat-rpm-config`), 2023-08-04. ↩

14. SUSE, openSUSE Security Features, openSUSE Wiki, 2022-12-8. ↩

15. Ubuntu, Ubuntu Security Features, Ubuntu Wiki, 2023-08-07. ↩ ↩2

16. LLVM team, Controlling Diagnostics in System Headers, Clang Compiler User's Manual, 2017-03-08. ↩

17. GCC team, Options for Directory Search, GCC Manual, 2023-07-27. ↩

18. LLVM team, Clang command line argument reference¶: -isystem<directory>, Clang documentation, 2017-09-05. ↩

19. Kitware, include_directories¶, Cmake Documentation, 2023-10-23. ↩

20. GCC team, Using the GNU Compiler Collection (GCC): Warning Options., GCC Manual, 2023-07-27. ↩

21. LLVM Team, Clang documentation: Diagnostics flags in Clang, Clang documentation, 2023-03-17. ↩

22. Polacek, Marek, "-Wimplicit-fallthrough in GCC 7", Red Hat Developer, 2017-03-10 ↩

23. Corbet, Jonathan. "An end to implicit fall-throughs in the kernel", LWN, 2019-08-01. ↩

24. ISO/IEC, Programming languages — C ("C17"), ISO/IEC 9899:2018, 2017. Note: The official ISO/IEC specification is paywalled and therefore not publicly available. The final specification draft is publicly available. ↩

25. Shafik, Yaghmour, "GCC 7, -Wimplicit-fallthrough warnings, and portable way to clear them?", StackOverflow, 2015-01-15. ↩

26. Johnston, Philip. -Werror is Not Your Friend. Embedded Artistry Blog, 2017-05-22. ↩

27. GNU C Library team, Source Fortification in the GNU C Library, GNU C Library (glibc) manual, 2023-02-01. ↩

28. Poyarekar, Siddhesh, How to improve application security using _FORTIFY_SOURCE=3, Red Hat Developer, 2023-02-06. ↩ ↩2

29. GCC team, Arrays of Length Zero, GCC Manual (experimental 20221114 documentation), 2022-11-14. ↩

30. Linux Man Pages team, malloc_usable_size(3), Linux manual page, 2023-03-30. ↩ ↩2

31. kpcyrd, Task Todo List Prepare packages for -D_FORTIFY_SOURCE=3, Arch Linux Task Todo List, 2023-09-05. ↩

32. GCC team, Using Macros in the GNU C++ Library, The GNU C++ Library Manual, 2023-07-27. ↩

33. Wakely, Jonathan, Enable lightweight checks with _GLIBCXX_ASSERTIONS, GCC Mailing List, 2015-09-07 ↩

34. Metzger-Kraus, Christof. Don't use GLIBCXX_ASSERTIONS in production, Object Oriented Particle Accelerator Library (OPAL) Issue Tracker, 2021-01-16. ↩

35. GCC team, Using the GNU Compiler Collection (GCC): Warning Options: `-Wstrict-flex-arrays`, GCC Manual, 2023-07-27. ↩

36. Guelton, Serge, The benefits and limitations of flexible array members, Red Hat Developer, 2022-09-29. ↩ ↩2

37. Cook, Kees, GCC Bug 101836 - `__builtin_object_size(P->M, 1) where M is an array and the last member of a struct fails`, GCC Bugzilla, 2021-08-09. ↩

38. GCC team, Program Instrumentation Options: `-Wsanitize=bounds`, GCC Manual, 2023-07-27. ↩

39. Corbet, Jonathan, "GCC features to help harden the kernel", LWN, 2023-09-05. ↩

40. Zhao, Qing, "[GCC13][Patch][V2][1/2]Add a new option -fstrict-flex-array[=n] and attribute strict_flex_array(n) and use it in PR101836", GCC Mailing List, 2022-08-01. ↩

41. Edge, Jake, Safer flexible arrays for the kernel, LWN, 2022-09-22. ↩

42. Cook, Kees, and Gustavo A.R. Silva, "Progress on Bounds Checking in C and the Linux Kernel", Linux Security Summit North America 2023, 2023-05-12. ↩

43. Hebb, Tom, GCC's -fstack-protector fails to guard dynamic stack allocations on ARM64, GitHub metaredteam/external-disclosures Advisories, 2023-09-12. ↩ ↩2

44. Arm, GCC Stack Protector Vulnerability AArch64, Arm Security Center, 2023-09-12. ↩ ↩2

45. Common Vulnerability Enumeration Database, CVE-2023-4039, 2023-09-13. ↩

46. Shen, Han, New stack protector option for gcc, Google Docs, 2011-11-30. ↩

47. Intel, "A Technical Look at Intel's Control-flow Enforcement Technology", 2020-06-13. ↩

48. ARM Developer, Arm Compiler armclang Reference Guide Version 6.12 -mbranch-protection. ↩

49. GCC team, Support for Nested Functions., GCC Internals, 2023-07-27. ↩

50. Bendersky, Eli, Position Independent Code (PIC) in shared libraries, Eli Bendersky's website, 2011-11-03. ↩

51. Bendersky, Eli. Position Independent Code (PIC) in shared libraries on x64, Eli Bendersky's website, 2011-11-11. ↩

52. Wang, Xi, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek, 2012, "Undefined Behavior:What Happened to My Code?", APSys '12, ACM, https://pdos.csail.mit.edu/papers/ub:apsys12.pdf ↩ ↩2 ↩3 ↩4

53. Zdrnja, Bojan Zdrnja, 2009-07-17, A new fascinating Linux kernel vulnerability, https://isc.sans.edu/diary/A+new+fascinating+Linux+kernel+vulnerability/6820 ↩

54. Torvalds, Linus, 2018-04-04, https://lkml.org/lkml/2018/4/4/601 ↩

55. Kerrisk, Michael, Building and Using Shared Libraries on Linux, Shared Libraries: The Dynamic Linker, man7.org, February 2023. ↩

56. Kratochvil, Jan, Memory error checking in C and C++: Comparing Sanitizers and Valgrind, Red hat Developers, 2021-05-05. ↩

57. LLVM Sanitizers team, AddressSanitizerFlags, GitHub google/sanitizers Wiki, 2019-05-15. ↩

58. LLVM Sanitizers team, AddressSanitizer, GitHub google/sanitizers Wiki, 2019-05-15. ↩

59. LLVM Sanitizers team, ThreadSanitizerFlags, GitHub google/sanitizers Wiki, 2015-08-31. ↩

60. LLVM Sanitizers team, ThreadSanitizerReportFormat, GitHub google/sanitizers Wiki, 2015-08-31. ↩

61. GCC team, Program Instrumentation Options, GCC Manual, 2023-07-27. ↩

62. LLVM team, UndefinedBehaviorSanitizer, Clang documentation, 2023-03-17. ↩

63. DWARF Debugging Information Format Committee, DWARF Version 5 Debugging Format Standard, DWARF Debugging Standard Website, 2017-02-13. ↩

64. Linux Foundation, Linux Standard Base Core Specification, Generic Part, Chapter 10.8. ABI note tag., Linux Foundation Referenced Specifications, 2015-05-27. ↩

65. Binutils team, LD Options, Documentation for binutils, 2023-07-30. ↩ ↩[2]

66. US NSA, Ghidra homepage ↩

67. Hex-Rays, IDA Pro homepage ↩

68. Binutils team, objcopy, Documentation for binutils, 2023-07-30. ↩ ↩[2]

69. LLVM team, llvm-objcopy, LLVM Command Guide, 2023-03-17. ↩

70. Binutils team, strip, Documentation for binutils, 2023-07-30. ↩

71. LLVM team, llvm-strip, LLVM Command Guide, 2023-03-17. ↩

72. GDB team, Debugging Information in Separate Files, Debugging with GDB, 2023-08-16. ↩

73. The `-Wl,-z,nodump` option sets `DF_1_NODUMP` flag in the object's `.dynamic` section tags. On Solaris this restricts calls to `dldump(3)` for the object. However, other operating systems ignore the `DF_1_NODUMP` flag. While Binutils implements `-Wl,-z,nodump` for Solaris compatibility a choice was made to not support it in `lld` (D52096 lld: add -z nodump support). ↩

74. The `-Wl,-z,noexecheap` option is a Hardened Gentoo extension to Binutils ported from PaX. PaX is a patch to the Linux kernel and Binutils that adds a `PT_PAX_FLAGS` program header to ELF objects that stores memory protection information the PaX kernel can enforce. The protection information stored in `PT_PAX_FLAGS` will not benefit software running on systems without a PaX kernel. The Gentoo patch (`63_all_binutils-` *<version>* `-pt-pax-flags-` *<date>* `.patch`) for various versions of Binutils since 2.15 can be found at https://dev.gentoo.org/~vapier/dist/. ↩

75. The LLVM libc++ has gone through a number of design iterations with its "safe" mode of operation. Starting with libc++ release 17.0.0 the "safe" mode has been deprecated in favor a new hardened mode of operation that provides a narrower set of checks (security-critical checks that are performant enough to be used in production). For more information see: LLVM team, Libc++ 17.0.0 Release Notes, Libc++ documentation, 2023-07-27; LLVM Team, Hardened Mode, Libc++

documentation, 2023-07-27 and Varlamov, Konstatin, Deprecate `_LIBCPP_ENABLE_ASSERTIONS` , LLVM Phabricator, 2022-07-11. ↩ ↩[2]

76. GCC team, x86 Built-in Functions, GCC Manual, 2023-07-27. ↩

77. LLVM team, SafeStack, Clang documentation, 2023-11-14. ↩
