Is more always better? Evaluating security hardening options in GCC

Jacob Inwald

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

2025

Abstract

NOT STARTED, need to make methodology concrete first

Research Ethics Approval

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

Declaration

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

(*Jacob Inwald*)

Acknowledgements

This project would not have been possible without all the people who have helped, in one way or another. I owe a lot to my parents, who have supported my educational journey from the very beginning and have pushed me in all my endeavours. Thank you to my supervisor, Prof. David Aspinall, for providing guidance and feedback throughout this project, as well as a great deal of books. My appreciation also goes to the faculty and staff in the University of Edinburgh, who assisted with access to the resources used in the development of this project. Benjamin Smith, for setting up a DICE machine I could use for testing and Prof. Bjoern Franke, for granting me access to SPEC2017.

To my partner, Emma, your unwavering support and advice has been both pivotal in the completion of this project and my degree. I am eternally grateful of the support you have given in the more difficult stretches. I am also grateful for my friends, who provided distractions when needed, and support in the stressful times. The experiences I have had at this university has been made that much more special, knowing I have had them around.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

C and C++ are vulnerable to memory safety errors, which can lead to extensive damage to systems or property e.g. HeartBleed[1]. These errors are pervasive, Microsoft and Google have both reported that $\sim 70\%$ of known vulnerabilities are memory safety errors [11, 19]. Despite this, C/C++ usage is still widespread, with Orlowska et al. [27] finding that C and C++ were among the most discussed languages on StackOverflow[2]. Further to this, CISA et al. [6] has found that 52% of open source security critical projects[3] are written in memory unsafe languages. While attempts have been made to add compile-time memory safety to C (SafeC [3], SoftBound [22]), these solutions remain under-utilised due to the performance overhead they incur. Therefore, other solutions must be sought. This paper discusses security hardening measures, which are transformations applied at compile-time that aim to prevent the exploitation of memory errors. These act as fail-safe measures, making exploits more difficult to carry out or thwarting exploits entirely.

Security hardening measures date back to 1998 [8], with the introduction of StackGuard, a technique of detecting memory tampering on the program stack. Since then, many different hardening measures have been proposed and merged into mainstream compilers, such as GCC or Clang. These use a variety of approaches, such as randomization defences (ASLR [10], PIE [36]), memory tampering detection (Shadow Stack [14], StackGuard [8]), and control-flow enforcement (IBT [13], CFI [1]). Each approach has been proven to be effective in its own regard [16], but they were all developed independently and merged into mainstream compilers individually. The current assumption is that more is better; stacking on hardening measures always improves security.

This may not necessarily be the case, compilers can work in unintended ways. Wang et al. [38] showed that optimization transformations interfere with security tests, in

[1]This vulnerability is an out-of-bounds read in the OpenSSL library, and allowed attackers to leak sensitive information from servers [5].

[2]A website for providing help and support for programmers (`https://stackoverflow.com`)

[3]As specified by Open Source Security Foundation Securing Critical Projects Working Group's List of Critical Projects [12]

some cases removed checks entirely. Xu et al. [40] found that particular optimization transformations interfered with the security of CFI schemes, making programs *more* vulnerable. Xu et al. [41] found that compiler introduced security bugs (CISBs) are more common that expected, and the rate of found CISBs is increasing. Currently, no papers have tested whether hardening measures interfere with each other at compile time, instead focusing on optimization transformations.

As the hardening measures available in mainstream compilers were added independently of each other, they may interact in unforeseeable ways. If these interactions were to introduce or cause a failure of one of the hardening measures, code may be falsely trusted. The impact of a CISB here could be vast; any interaction introduced at compile time will impacts thousands of programs. The potential impact of this was demonstrated recently, when Ye and Hu [42] found that some well known CFI transformations enabled an executable stack. This made *every* program compiled with those transformations more vulnerable. Therefore, investigating the potential for combinations of hardening measures to introduce CISBs is both relevant and important.

1.2 Project Aims and Contributions

This project investigates whether combinations of hardening measures create new unintended side effects, such as one causing failure of another, or introducing new vulnerabilities. Namely, this project accepts or rejects the following hypothesis:

H_1: Combinations of stack canaries, fortify source, shadow stack and IBT do not interfere with each other, leading to loss of security.

The effectiveness of the combinations on modern x86-64 systems are tested on the RIPE and RecIPE benchmarks [16, 39], alongside a self-curated dataset of case studies. This projects key contributions and findings include:

1. **A repeatable framework for reproducing crowd-sourced exploits**. To curate the dataset of exploits, I formulate a framework to ensure that any results are repeatable, and usable for research outside the scope of this project.

2. **A curated dataset of exploit-code pairings.** I use the framework to create a small dataset of exploit-code pairings, that can be used to test the hypothesis of this project. Moreover, these exploits are robust, meaning that they can be run on a variety of x86-64 machines to achieve similar results.

3. **Systemic evaluation of combinations of hardening flags.** Two separate approaches are used to evaluate the effectiveness of combinations. Namely, these approaches are benchmarking and case study analysis. Overall, more is better, excepting one possible interaction.

4. **Correction of misinformation regarding certain exploits.** This project finds inconsistencies in a variety of exploits tested, including errors in the National Vulnerability Database (NVD)[4]. Additionally, chapter **??** demonstrates the im-

[4]A standardized database of vulnerability information [24].

portance of correctly versioning the compilation stack when reporting exploits, such as GCC.

1.3 Overview

The following chapters will present the background required, methodology used, results gained and finally discuss in the wider context. Namely:

Chapter 2 provides essential background on memory safety vulnerabilities in C/C++ programs, focusing on common exploitation methods such as stack and heap-based overflows. It then introduces and describes the four security hardening techniques studied in this project: stack canaries, fortify source, shadow stack, and indirect branch tracking (IBT).

Chapter 3 outlines the methodology used to evaluate the effectiveness and potential interactions of these hardening measures. This includes the definition of compiler flag combinations, the benchmarking process using RIPE and RecIPE, and the construction of a framework for reproducing real-world CVE exploits. The chapter also discusses the filtering and preparation of case studies drawn from ExploitDB, and the process of building consistent testing environments using Docker.

Chapter 4 presents the results from both benchmark suites and the curated set of reproduced exploits. It analyzes the impact of individual and combined hardening flags on preventing successful exploitation, and highlights key observations, such as unexpected interactions, challenges in reproduction, and discrepancies introduced by compiler and platform differences.

Chapter 5 reflects on the findings, discussing the results with respect to the original hypothesis. It examines the limitations of the project, particularly in terms of hardware support, emulator fidelity, and exploit variability. Additionally, it proposes avenues for future work, and finishes with any concluding remarks.

Chapter 2

Background

2.1 Memory Errors

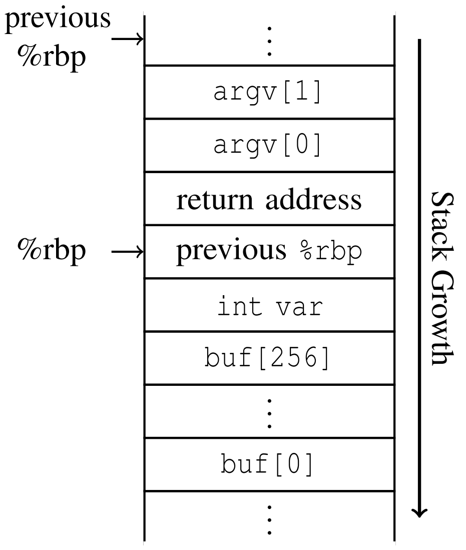
Memory safety errors refer to situations when programs read/write in unintended ways to arbitrary regions of memory. An example would be a buffer variable overflowing onto the stack, or a heap allocated structure overwriting previously allocated data on the same segment. These errors have been exploited for decades, from the first exploits in 1989 (Morris Worm [28]) to the modern day (Dirty Pipe [18]). They remain as prevalent today as they did 35 years ago. In 2019, Microsoft reported that $\sim 70\%$ of CVEs they have assigned from 2006 to 2018 were memory safety errors [19]. Similarly, an analysis of 0-day vulnerabilities used in 2021 found that 67% were memory safety errors [11]. Understanding the design of security hardening measures requires an understanding of how memory safety errors are exploited. The following subsections will introduce some common exploitation methods for memory corruption on the stack and on the heap. This should provide context for the hardening measures discussed in Section 2.2.

2.1.1 Stack Exploitation

The program stack is a data structure that maintains information about the active subroutines of a program. It is split in stack frames, each representing an active subroutine, shown in Figure 2.1. Each stack frame stores the subroutines arguments, local variables, return address, and a pointer to the previous stack frame. The return address tells the program where to pick up execution after the subroutine is finished, and the previous stack frame pointer tells the program where to find the stack frame of the previous subroutine. This information allows the program to maintain the current state of the program.

As the stack essentially stores the control flow of the program, if an attacker can write to it they can hijack the execution of the program. This makes stack-based memory errors, such as buffer overflows[1], extremely powerful. An attacker can take advantage

[1]These are specific kinds of memory errors where an allocated buffer on the stack ‘overflows‘ its boundaries

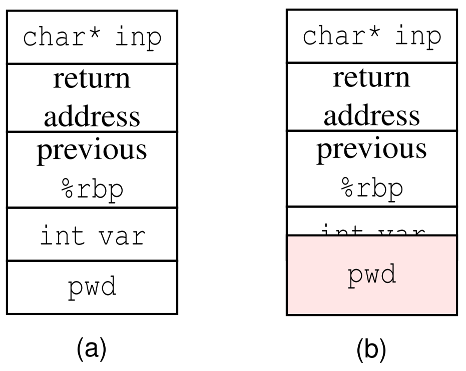


```
void foo(char* argv) {
  int var;
  char[256] buf;
}
```

Figure 2.1: A stack frame, alongside the subroutine it represents.

of an overflow on the stack in a variety of ways, but the most relevant ways are local variable corruption, and return oriented programming (ROP).

Local variable corruption takes advantage of the fact that a subroutines stack-allocated variables are *contiguous* in memory. Due to this, a stack buffer overflow can corrupt local variables for the subroutine. If one of those variables is used for a check later on down the line, this can erroneously trip the check. An example is shown in Figure 2.2. Here the `authenticate` function copies the input into a password buffer, and then checks if the secret is correct, returning true if the user is authenticated or not. If a input larger than the buffers bounds of 16 characters is passed in, the `var` variable will be overwritten on the stack, incorrectly authenticating the user.

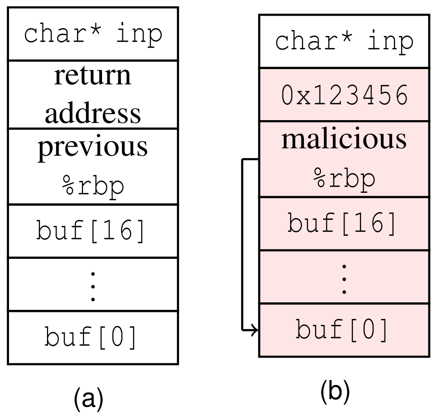


```
bool authenticate(char* inp) {
  int var = 0;
  char[16] pwd;
  strcpy(pwd, inp);
  if (stcmp(pwd, inp))
    var = 1;
  return var == 1;
}
```

(c)

Figure 2.2: An example where a stack buffer can overflow and overwrite a local variable. In (c) the example function is shown, and (a) shows the corresponding stack frame. (b) shows the overflow described in text, where the `pwd` buffer overwrites the `var` variable.

Another method of attack is Return Oriented Programming (ROP), which overwrites the return address and previous stack frame pointer. First, the attacker finds a series of isolated functions (gadgets) that serve a purpose, e.g. setting up a system call. They then construct a chain of phony stack frames that represents this chain of functions. They then overwrite the return address and previous frame pointer to point to the start of the chain. When the program attempts to return, the ROP chain will start, executing some arbitrary code, normally `execve('\bin\sh')`. Figure 2.3 shows a example of what a ROP attack would look like. Here, the gadget is the `sh` function in Figure 2.3c, which the return address is overwritten with in Figure 2.3b. The malicious `%rbp` points to an attacker controlled location on the stack, which simulates a valid stack frame.



```c
void sh() { // address 0x123456
    execve('/bin/sh');
}

void vuln(char* inp) {
  char[16] buf;
  strcpy(buf, inp);
}
```

(c)

Figure 2.3: An example where a stack buffer can overflow and overwrite a local variable. In (c) the vulnerable code is shown, and (a) shows the corresponding stack frame. (b) shows the overflow described in text, where the `buf` buffer overwrites the return address and previous `%rbp`.

2.1.2 Heap Exploitation

The heap is a memory region allocated beneath the stack that stores dynamically allocated variables. A new variable is allocated into a block of memory on the heap using the `malloc` system call, which is then freed when the `free` system call is used. Many important variables are stored on the heap, such as the Global Offset Table (GOT) or C++ virtual method tables (vtables), which both hold contextual references to subroutines. Attacks on the heap can be just as powerful as their stack based counterparts, as they are not limited in write space. A successful heap attack could change application level pointers or alter global function calls. Heap memory errors can come in a variety of flavours, but for this paper only the method of exploiting a Use After Free is relevant.

Use After Free refers to a particular memory error that occurs when a heap allocated pointer is used after it is freed, shown in Listing 2.1. This is dangerous as when a block is freed, any program can write to it. Therefore, the assumption is that an attacker can always write to freed blocks. In this case, the attacker has control over what is stored in `bar`, which may not lead to any further exploitation.

```c
void foo(){
  char* var = malloc(sizeof(char)*16);
  char* bar = malloc(sizeof(char)*16);
  free(var);
  strcpy(var, bar);
6 }
```

Listing 2.1: A example of a use after free in C, here `var` is used in line 5 after it is freed in line 4

Listing 2.2 illustrates what a fully exploitable C++[2] snippet may look like. In this case, an attacker could alter the `vtable` of `obj`, which acts as the function lookup table for a given class. If successful, when `obj->bar` is called in line 10, an arbitrary function may be called instead.

```
[..]
1 class Foo {
2     public virtual void bar() {
3         std::cout << "bar" << std::endl;
4     }
5 };
6
7 int main() {
8     Base* obj = new Base(); // allocate on heap
9     delete obj;             // free heap block
10     obj->bar();           // call heap function
11 }
[..]
```

Listing 2.2: A example of a use after free in C++, here `obj` is used in line 10 after it is freed in line 9

2.2 Hardening Techniques

Hardening measures aim to make exploiting pre-existing vulnerabilities more difficult, or in some cases, infeasible. Rather than eliminating vulnerabilities at the source-code level, these mechanisms operate at compile-time, introducing instrumentation and structural transformations to generate binaries that are more resilient to exploitation attempts. They are particularly valuable in C and C++ applications, where memory safety is not enforced by the language itself and manual memory management is prone to subtle and dangerous bugs.

This project focuses specifically on *runtime-enforced hardening techniques*, as they operate during program execution and are more likely to exhibit complex interactions when enabled simultaneously. These techniques include stack canaries, fortify source, shadow stack, and indirect branch tracking (IBT), each targeting a different attack vector. The following sections describe the internal mechanisms of these hardening measures in detail, as well as the compiler-level transformations that implement them. Understanding how each protection operates is key to later evaluating whether any unintended interference or degradation occurs when multiple measures are combined within a single binary.

[2]In C++, the `new` keyword acts as `malloc` does in C, and the `delete` keyword acts as `free` does in C

2.2.1 Stack Canaries

Stack canaries [8] are a security hardening option that aim to prevent buffer overflows on the stack. The concept behind their implementation is very simple: some known value is set before the return pointer on the stack frame and after the local variables, shown in Figure 2.4. This value is then checked against the known value before the program jumps to the return address, and aborts the program if the canary has changed. As a contiguous overflow from the local variables will overflow the canary value before the return address, an attack will alter the canary value and the program can stop the execution to prevent exploitation. This means that an attacker needs to know the value of the canary in order to bypass the protection, which can prove challenging.

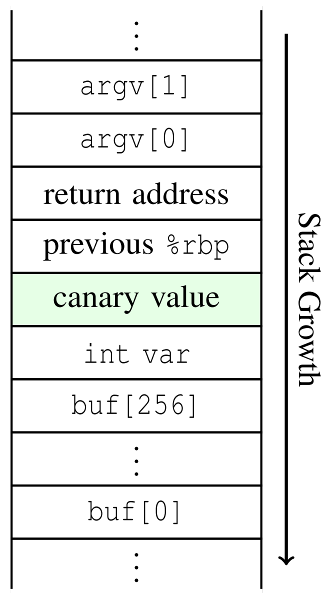


Figure 2.4: The stack frame from Figure 2.1, with stack canaries enabled

Listing 2.3 shows assembly code with stack canaries added in, to illustrate the transformation. As shown, the canary value is loaded in the word between the return address and then checked before the function returns to the return address given. If the canary is different to what it was set to, the program throws a stack check fail exception and then exits.

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

```

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

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

Listing 2.3: x86 assembly code after compilation with stack canaries

Stack canaries are effective in preventing overflow attacks on the return address, where the buffer is overflowed sequentially from a local buffer. However, they cannot prevent local variable corruption attacks such as those shown in Figure 2.2. Regardless, canaries are often a simple and relatively effective method of preventing sequential buffer overflows affecting the return address of a stack frame.

2.2.2 Fortify Source

This section refers to the `D_FORTIFY_SOURCE` flag, which adds size checks in standard library functions [?]. This prevents out of bounds writes with standard library functions in situations where the buffer size can be determined. `D_FORTIFY_SOURCE` has 3 modes with the most secure being `D_FORTIFY_SOURCE=3`, and decreasing options providing more compatibility with legacy code. In Listing 2.4, we see a disassembled `strcpy` function when `D_FORTIFY_SOURCE` is enabled. In line 15-17, we see the additional check added before calling `strcpy` which ensures that source array is smaller than the destination array

```
1 ; setup frame
2 push   %rbp        ; save previous %rbp
3 mov    %rsp,%rbp    ; new frame pointer for __strcpy_chk()
4 push   %r13
5 mov    %rdx,%r13 ; %r13 = %rdx | %rdx stores size of dest array
6 push   %r12
7 mov    %rdi,%r12
8 mov    %rsi,%rdi
9 push   %rbx
10 mov    %rsi,%rbx
11 sub    $0x8,%rsp
12
13
14 ; perform check
15 call   0x7ffff7dd7510 <*ABS*+0xafb80@plt> ; calls size on source array
16 cmp    %r13,%rax    ; size(src) - size(dest)
```

```
jae    0x7ffff7ed8470 <__strcpy_chk+64> ; jump to last line if
  source size is greater or equal to destination size

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

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

Listing 2.4: x86 assembly code for the fortified `strcpy` function (__strcpy_chk) after compilation

This is a simple transformation, and is effective in making `glibc` functions such as `strcpy` and `memcpy` safer. This option relies on the size of the source array being discoverable at runtime, which may not be true[**?**].

2.2.3 Shadow Stack

A shadow stack [4] attempts to prevent any stack based overflows from hijacking execution by enforcing memory integrity on the return address. This is achieved by separating the control information and the stack allocated data. If the control information, such as the return address, has changed, we know that the stack has been tampered with. The shadow stack stores an immutable copy of the control information of the program, and if there is a mismatch between the program stack and shadow stack, program execution is halted. By protecting the integrity of the return address, the shadow stack enforces backward-edge Control Flow Integrity (CFI). This means that return statements correctly return control flow back to the call site. For clarity, Figure 2.5 shows an example control flow graph of a subroutine, and highlights the stage at which the shadow stack acts.

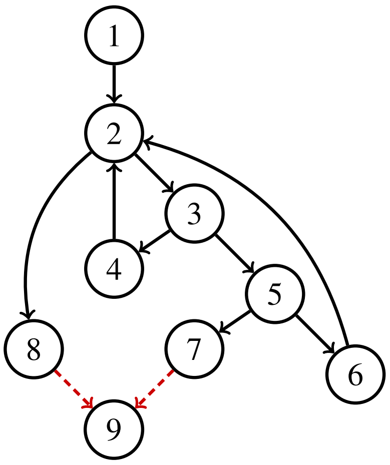
The prevalence of attacks on the return address has motivated manufacturers to incorporate hardware support for shadow stack, into processor architectures such as x86_64 [14], RISC_V [35] and AArch64 [7]. GCC activates the x86_64 shadow stack feature with the flag `-fcf-protection=return`, which is provided by Intel's Control Enforcement Technology (CET) [14]. CET is supported for 11th generation Intel CPUs and onwards [15], while OS support for hardware shadow stack was added for kernel versions `6.6-rc1` onwards [34].

2.2.4 Indirect Branch Tracking

Indirect Branch Tracking (IBT) [13] is a weak CFI scheme, which attempts to thwart control flow attacks by reducing the amount of valid targets for branches. It inserts an

```
int binsearch(int x, int v[], int n)
{
  | int low, high, mid;
1 | low = 0;
  | high = n - 1;
    while (low <= high) | 2
    { |
    3 | mid = (low + high)/2;
      | if (x < v[mid])
            high = mid - 1; | 4
    5 | else if (x > v[mid])
            low = mid + 1;  | 6
    7 | else return mid;
    }
    return -1; | 8
} | 9
```

(a)



(b)

Figure 2.5: Example of a control-flow graph from [2], with the branches protected by Shadow Stack highlighted with red dashed lines. Each node in (b) refers to a section of code in (a), delineated by "`n` |".

`endbr64`, or end branch, instruction at the beginning of each valid branch location. If the CPU branches to a memory location without an `endbr64` instruction, program execution will be halted. This allows the hardware to catch jumps to unintended locations, cutting down the amount of gadgets available for a ROP attack.

Figure 2.5 provides some context as to where these `endbr64` instructions are placed. In this case, they would be placed before each node's associated assembly code on the control-flow graph, as each node represents a valid branch end point. From this, it is simple to see that IBT simply checks if the control-flow is at a valid node on the graph and *not* whether the correct control-flow to this node is followed.

GCC generates IBT compatible assembly with the flag `-fcf-protection=full`, which is provided by Intel's CET. As mentioned in the previous section, CET is supported from 11th generation Intel CPUs onwards, and OS support for IBT was added for kernel versions `6.2` onwards [43].

Chapter 3

Methodology

This chapter describes the methodology used to examine different combinations of security flags. First, the set of hardening measures tested in this project is formulated. Next, we discuss the main approaches to evaluating security hardening measures: benchmarking and case study analysis. Finally, the strategy for benchmarking and case study curation is presented and the framework for case study replication is described.

3.1 Goal and Problem Statement

The goal of this project is to accept or reject the following hypothesis:

H_1: Combinations of stack canaries, fortify source, shadow stack and IBT do not interfere with each other.

To evaluate H_1, the set of hardening measures first needs to be defined. This should provide the ability to compare between baselines on multiple levels, so the combination set can be evaluated against each individual hardening measure. As such, it is formulated as a composition of the following sets:

BASELINE: This is the baseline configuration for GCC without *any* measures activated. This is included in the set to ensure a comparable baseline for each of the combinations and catch any invalid test cases. It is defined as:

(baseline): `gcc -g -w -D_FORTIFY_SOURCE=0 -no-pie -fno-stack-protector -z execstack -z norelro -fcf-protection=none`

INDIVIDUAL: This set is the baseline with each individual hardening measure introduced in Section 2.2. This is done by appending the following flags for stack canaries (stkpro), fortify source (forti), shadow stack (shstk) and full CET (ibt+shstk) respectively:

(stkpro): `-fstack-protector-strong`
(forti): `-D_FORTIFY_SOURCE=3 -O2`
(shstk): `-fcf-protection=return`
(shstk+ibt): `-fcf-protection=full`

COMBINATION: This set is composed of the increasing combinations of the individual hardening flags. This is defined as following the following set:

(stkpro+forti): `-fstack-protector-strong -D_FORTIFY_SOURCE=3 -O2`
(stkpro+forti+shstk): `-fstack-protector-strong -D_FORTIFY_SOURCE=3 -O2 -fcf-protection=return`
(stkpro+forti+shstk+ibt): `-fstack-protector-strong -D_FORTIFY_SOURCE=3 -O2 -fcf-protection=full`

This project uses two approaches to evaluating H_1, benchmarking and case study analysis. The first method uses established benchmarks, such as Juliet [17] or RIPE [39], which provide large collections of synthetic test cases to test for "good" or "bad" behaviour. Benchmarking allows consistency across multiple studies, and provide a baseline to test the relative effectiveness of a measure. The second method uses publicly accessible exploits or CVE proof of concepts (PoCs), and tests whether the transformation prevents it. Case study analysis provides a more grounded approach, as it tests real life cases as well as the complexity that comes with them.

3.2 Benchmarks

Security benchmarking involves utilising an existing collection of test sets to provide a relative analysis of the effectiveness of some hardening measure. A few different security benchmarks exist, although they have varying aims. As of writing, the security benchmarks available are:

Juliet [17] is a security benchmark proposed by NIST in 2012, aimed towards training static analysis tools. These are tools that find vulnerabilities in programs by directly analysing the source code, and are normally used in the secure development life cycle. As of version 1.3, released in 2017, Juliet has 64,099 C/C++ test cases, 28,942 C# test cases and 28,881 Java test cases. Test cases are organised by CWE, and consist of source code containing "good" and "bad" implementations of functions. Test cases are artificial, but are defects sourced from real-world CVEs. As Juliet is aimed towards static analysis tools, the given test cases do not have associate PoCs attached.

RIPE [39] is a synthesised test bed aimed at evaluating the effectiveness of buffer overflow prevention techniques. In 2018, Rosier [31] ported RIPE to 64 bit architectures. RIPE aims to mimic attacker behaviour, essentially attacking itself along 6 different dimensions of attack. It describes attacks as a linear combination of these dimensions of attack, implying that any unique combination of these dimensions produces a novel attack. These dimensions are buffer location, target code pointer, overflow technique, attack code/payload and function abused. With all the attack dimensions selected, RIPE provides 850 different buffer overflow attacks. It has been used extensively to demonstrate the viability of hardening measures [**? ? ?**].

RecIPE [16] was proposed as a successor to RIPE, addressing limitations in the attack framework RIPE proposed. It provides a modernised framework, generating attacks against specific attributes and mimicking an outside attacker more closely. RecIPE is also less fragile than RIPE, and test cases do not break when the environment is changed. This provides more realistic analysis of memory exploits. It has 174 test cases, which test a variety of targets and data segments. However, due to its novelty, it has not been used for analysis; to date one paper has used it for evaluation [33].

Of these benchmark suites, RIPE and RecIPE were selected for analysis. While Juliet is large and well-researched, its structure does not lend itself well for dynamic testing, given it is composed of static test cases. RIPE has been used in past literature extensively to test the effect of hardening flags. However, as it was originally built for 32 bit machines, and the CET based hardening measures are only available for 64 bit machines, the 64 bit port RIPE-64 was used. While RecIPE has not been used extensively, it remains the most modern security benchmark available at the time of writing. Therefore, it is the most compatible with modern systems, meaning a greater proportion of test cases are relevant to modern software.

3.2.1 Experimental Setup

The following platform was used to collect results from the benchmarks. We used GCC 13.3.0 with Ubuntu GLIBC `2.39-0ubuntu8.4` on Ubuntu 24.04.2 LTS, kernel version `6.8.0-53-generic`, running on a 12th gen Intel Core i5-12500. All test cases were run with the extra environment variable `GLIBC_TUNABLES=glibc.cpu.hwcaps=SHSTK`, to enable Shadow Stack on the system. IBT is not supported by this version of Linux[1], so IBT Intel SDE [**?**], an emulator for Intel CPUs, was used to collect results for IBT results. Intels CET based hardening measures (Shadow Stack and IBT) are only supported from 11th generation Intel CPUs **?**].

To enable the support for Intel CET features, the RIPE and RecIPE source code had to be altered slightly, to either add in the extra environment variable or emulate with SDE, depending on the combination of hardening measures. Some examples are shown in Appendix **??**. Any test cases that did not compile under the chosen platform, e.g. RecIPEs `hook` targets, were discarded. Further to this, any test cases that failed under the baseline conditions were marked 'broken' and removed from further analysis.

3.3 Case Study Analysis

CVE case studies are a widely used technique of measuring the effectiveness of security measures [**? ?**]. They provide a more representative view of exploits in the 'wild', and allow investigation of more complex interactions. Further to this, they are diverse and can be a result of multiple chained vulnerabilities, which in this case increases the number of interactions possible. Despite these benefits, reproducing CVE case studies is *laborious* [21]. Moreover, the quality of the data available can be poor [9, 21], increasing the challenge with reproduction. Therefore a robust framework is required to ensure that the data produced within this project is usable for future research.

[1] I could not get access to a machine with both OS IBT support and hardware IBT support

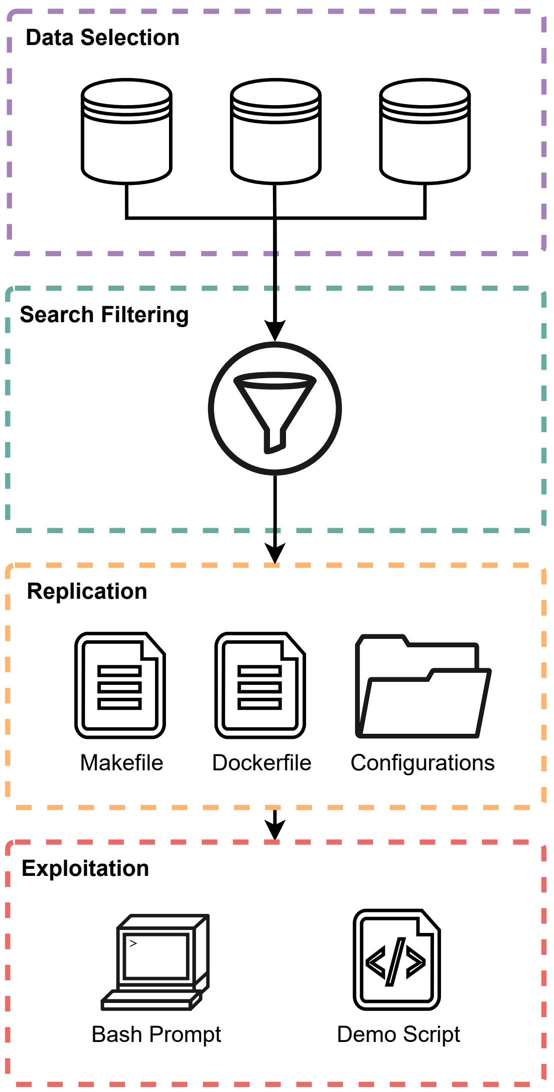


Figure 3.1: The framework used to reproduce the cases studies examined.

To further specify these aims, the framework must:

- Produce a set of case-studies that are not limited by this studies scope, and can be user for other applications, such as teaching.
- Ensure that set is concretely reproducible, from source to exploitation.
- Be repeatable itself, such that others can produce similar results.

Figure 3.1 shows the framework used to achieve these aims. First, the data source is fixed, allowing for specific filtering in the search filtering phase. Then, the quantity of the CVE exploits is cut down with the search filtering outlined in Section 3.3.2. Finally, the exploit is reproduced and tested with the tools outlined in Section 3.3.3.

3.3.1 Data Sources

There are a few different sources of publicly available exploits online. These include, but are not limited to:

ExploitDB [25] is a large open source database of crowd-sourced exploits maintained by Offensive Security. ExploitDB is one of the most widely-used archives, offering an extensive collection of publicly available exploits and PoCs. It contains over 40000 exploits as part of the archive, starting from 1988, with additional metadata such as platform, exploit type included with each entry. Each entry is largely unstructured with some information, such as the affected OS, stored as metadata, but other information, such as affected software version, stored in the entry.

OpenWall [26] is an online repository of security resources, composed of mailing lists and archives dedicated to security advisories, vulnerability disclosures, and exploit code. While unstructured, its OSS-Security mailing list is one of the largest online sources of exploits and vulnerabilities. There are 17 years worth of exploits and vulnerability reports stored on the mailing list archives.

RedHat Bugzilla [30] is a bug-tracking system maintained by RedHat. Bugzilla contains detailed records of reported vulnerabilities, bugs, and patches specifically within the RedHat ecosystem. Each entry related to vulnerabilities varies in completeness, with some providing PoCs and other descriptions of the vulnerability.

PacketStorm [29] provides a robust archive of security exploits, PoCs, advisories, and security tools. It was launched in 1998 and has been consistently added to since. It consists of largely unstructured data, with each entry consisting of vulnerability reports, and in some cases exploit PoCs.

While there are a variety of sources, the associated data quality is often poor. Mu et al. [21] attempted to reproduce 368 exploits from RedHat Bugzilla, Exploit-DB and OpenWall combined, finding that 95.1% of vulnerabilities were missing at least one required information field, 87% did not include software installation options or configurations and 22% did not even include the vulnerable OS. Inconsistency is an issue as well, Dong et al. [9] found that under 50% of vulnerability reports from OpenWall and under 80% of reports from Exploit-DB strictly matched the vulnerable software version numbers found on the standardized NVD. Indeed, even the standardized NVD can be wrong, as will be shown in Section 4. Therefore choosing the best data source is important, as it will partially ease some the challenges associated with CVE reproduction. Exploit-DB has the least amount of inconsistencies with the NVD [9], and was the second most well-reproduced in Mu et al. [21]. Therefore, Exploit-DB was used as the primary source of the exploits for the case studies.

3.3.2 Search Strategy

Exploit-DB has over 40000 exploits stored on it, so selecting a specific set of exploits to explore is difficult. To cut down the amount of exploits that either lacked information, or were unrelated to the aims of the project, the following search strategy was followed:

1. An intial filter of the dataset using the pre-existing tags ‘platform’ and ‘type’. The platform was set to Linux, to maintain consistency with the benchmark approach, and the type was set to Local to reduce the complexity of exploit setup.
2. All exploits published before 1st January 2018 were discarded to keep the dataset relevant to modern systems.
3. Manual tagging of the remaining subset, to ensure the following rules:
   - Entries have open source targets, which is necessary to build the target from source.
   - Entries are written in C/C++, which is neccessary to enable building the target with the chosen hardening measures.
   - Entries do not require instrumentation, such as address sanitisation to function, as this is not idicative of a normal run environment.
   - Entries do not require the Metasploit[2] framework, as it abstracts the details of exploit away from the code, which makes the results harder to interpret.

After the initial keyword filtering of “Linux” and “local” 1087 entries remained and after the date cutoff, 180 entries remained. After manual tagging, 24 were discarded as they were not open source, 58 were discarded as they were Metasploit based, and finally 15 were discarded as they were not written in C or C++, leaving 81 remaining. Finally, any exploits that required instrumentation such as address sanitisation to demonstrate were discarded. After filtering 69 entries remained, and after deduplication 64 entries were left. `TODO: Maybe put list in the appendix`

3.3.3 Replication and Exploitation

Each case study is implemented as a Docker[3] container, which can build the vulnerable program from source and then exploit it with the exploit. Using Docker allows the results to be consistent across machines, as well as repeatable. Each case study Docker image is created by the following process:

1. Setup minimal test environment, using the Ubuntu Noble image (`ubuntu:noble`).
2. Download build requirements and build source code.
3. Configure source to support customized build flags, and test install as root.
4. Download requirements for exploit and download exploit.
5. Configure base system for exploit.
6. Boot into the system as an unprivileged user.

This results in an image that can be used to reproduce the case study. Each Docker image has a partner Makefile, which allows for the exploitation stage of the framework shown

[2]A framework for developing and sharing exploits, produced by RedHat `https://www.metasploit.com/`

[3]Software that runs an alternate disk image on the base kernel ().

```
#syntax=docker/dockerfile:1
FROM ubuntu:noble
WORKDIR /home/

# Install core dependencies
RUN apt-get -y update && apt-get
   -y install git make
   build-essential libxml2-dev
   sudo  zsh meson ninja-build
   wget ca-certificates

# -- SETUP SOURCE --

# Install source dependencies
RUN apt-get -y install ...

# Add source code
RUN ...

# Instrument build file to
   allow for hardening
   culpable file
RUN ...

# -- ADD EXPLOIT --

RUN ...

# -- ADD DEMO --
ADD demo-exploit.sh ./
RUN chmod +x demo-exploit.sh
ENV HARDEN_FLAGS=""
```

Listing (3.1): Source file: `Dockerfile`

```
# Purpose:  Makefile for
   docker image
IMAGE_NAME=_template
EXITED:=$(shell sudo docker ps -a
   -q -f status=exited)

clean:
sudo docker rm -v $(EXITED)

build:
sudo docker build -t $(IMAGE_NAME)
   .

run/zsh:
sudo docker run -it --entrypoint
   /bin/zsh $(IMAGE_NAME)

run/bash:
sudo docker run -it --entrypoint
   /bin/sh $(IMAGE_NAME)

run/demo:
sudo docker run -it -e
   HARDEN_FLAGS="$(HARDEN_FLAGS)"
   --entrypoint ./demo-exploit.sh
   $(IMAGE_NAME)
```

Listing (3.2): Source file: `Makefile`

Figure 3.2: Template setup for framework used, on the left is the Dockerfile, which is used to build the case study image (equivalent to Replication in Figure 3.1). On the right is the Makefile, which is used to experiment and test the case study (equivalent to Exploitation in Figure 3.1)

in Figure 3.1. All Docker images have almost identical Makefiles, which standardizes the method of interacting with the case studies, ensuring consistency. All case studies used a template I created, shown in Figure 3.2. Using this template ensures that the testing platform is fixed across case studies.

3.3.4 Experimental Setup

As Docker is used, testing with Dockerfile requires `sudoer` privileges. Unfortunately, I was unable to get access to a machine with support for Shadow Stack or IBT for whch I had `sudoer` privileges. Therefore, the experimental setup is different than the one used for testing benchmarks in Section 3.2.1. As nearly half of the set of hardening measures require support for Shadow Stack and IBT, all testing used the Intel SDE emulator. As Docker only relies on the base machines kernel, the only relevant specification is the testing platforms kernel - version `6.13.5-200.fc41.x86_64`.

Chapter 4

Results

This chapter presents the results of the methodology described previously. The analysis is split into two sections, benchmarks and case studies. Section 4.1 presents the results of the RIPE and RecIPE benchmark suites, and Section 4.2 presents the reproductions of exploits found in Exploit-DB.

4.1 Benchmarks

The following sections will present the results obtained from the RIPE benchmark suite and the RecIPE suite. The RIPE and RecIPE suite were edited to add support for CET, shown in Appendix `TODO`. All results have been run on hardware unless otherwise specified.

4.1.1 RIPE

As RIPE does not have support for running programs with Intel Shadow Stack or IBT enabled, I patched in support for these features so that they could be tested. All IBT interactions were emulated using Intel SDE. Table 4.1 shows the results from the benchmark run, with both indirect and direct attacks and 3 runs per attack, giving a total

Setup	**Exploited**	**Attempted**	**Broken**
(baseline)	565 (96.8%)	19 (3.3%)	0 (0%)
(stkpro)	306 (52.4%)	10 (1.7%)	268 (45.9%)
(forti)	187 (32.0%)	7 (1.2%)	390 (66.8%)
(shstk)	497 (85.1%)	23 (3.9%)	64 (11.0%)
(shstk)+(ibt)*	373 (63.9%)	15 (2.6%)	196 (33.6%)
(stkpro)+(forti)	177 (30.3%)	3 (0.5%)	404 (69.2%)
(stkpro)+(forti)+(shstk)	173 (29.6%)	7 (1.2%)	404 (69.2%)
(stkpro)+(forti)+(shstk)+(ibt)*	154 (26.4%)	6 (1.0%)	424 (72.6%)

Table 4.1: Results from the RIPE benchmark with different combinations of hardening flags. Entries marked with a ‘*’ are emulated with SDE

Setup	Exploited	Attempted	Failed
(baseline)	164 (100.0%)	0 (0.0%)	0 (0.0%)
(stkpro)	100 (61.0%)	36 (22.0%)	28 (17.1%)
(dforti)	124 (75.6%)	16 (9.8%)	24 (14.6%)
(shstk)	113 (68.9%)	51 (31.1%)	0 (0.0%)
(ibt)+(shstk)*	116 (70.7%)	48 (29.3%)	0 (0.0%)
(stkpro)+(dforti)	100 (61.0%)	36 (22.0%)	28 (17.1%)
(stkpro)+(dforti)+(shstk)	92 (56.1%)	44 (26.8%)	28 (17.1%)
(stkpro)+(dforti)+(shstk)+(ibt)*	92 (56.1%)	44 (26.8%)	28 (17.1%)

Table 4.2: Results from the RecIPE benchmark with different combinations of hardening flags. Entries marked with a '*' are emulated with SDE.

of 1334 test cases. Of these 1334 test cases, 750 failed with the baseline conditions, so were marked as 'broken', and excluded from further analysis. 'Exploited' refers to attacks that successfully worked, 'Attempted' refers to attacks that modified the control flow or memory state but crashed due to the mitigations, and 'Failed' refers to attacks that were thwarted by the mitigations.

The results are inline with the current assumptions that more is better. As more measures are combined, their overall effectiveness increases. Indeed, these benefits compound, meaning that two measures are better than any single measure. For example, using both stack canaries and fortify source thwarts 404 test cases, while by themselves they thwart 268 and 390 respectively. This may indicate that the measures are complementing each others respective weaknesses. Finally, the modest reduction in attacks when adding SHSTK alone (from 96.8% to 85.1%) suggests that while SHSTK is effective at controlling return addresses, it does not mitigate other exploit vectors like format string vulnerabilities.

4.1.2 RecIPE

The RecIPE benchmark was modified slightly as well to allow testing with Intel SDE and to enable shadow stack protections. The targets `hook` and `exit` were removed from the benchmark. `hook` as it has been depreciated since GLIBC 2.24, and `exit` was removed as it is a less reliable target in the RecIPE benchmark. In this benchmark, 'Exploited' refers to successful testcases, 'Attempted' refers to test cases where the mitigation succeeded, and 'Broken' refers to test cases that didn't start with the mitigation enabled. The results are shown in Table 4.2.

Overall, the results follow the same trend as RIPE: adding more mitigations reduces the number of successful exploits. The baseline setup is completely vulnerable, with all 164 test cases succeeding. Again, the compounding effect of combined measures is shown, with the strongest configuration better than any single one. When combining stack protector, dynamic fortify, and shadow stack, that number drops to 92 successful cases, with 44 thwarted and 28 failing to start at all.

One notable discrepancy between hardware and emulated configurations appears when comparing `-fcf-protection=full` (IBT+SHSTK via SDE) to `-fcf-protection=return`

Weakness Type	**Kernel**	**Application**	**Total**
Memory Error	9	10	19
Input Validation	3	6	9
Authorisation/Permission Error	4	5	9
Race Conditions	2	5	7
Other	2	7	9
Uncategorised	3	17	20
Total	23	50	73

Table 4.3: Distribution of associated CWE with entry, there are 8 more CWEs than entries as some entries have multiple CWEs

(SHSTK only on hardware). Specifically, three attacks succeed under the full protection configuration that do not succeed with SHSTK alone: `Stack_NBoundOFlow_jmpbuf_memcpy`, `Stack_NBoundOFlow_jmpbuf_homebrew`, and `Stack_NBoundOFlow_jmpbuf_bcpy`. These cases all involve `setjmp` and `longjmp` usage, and the discrepancy may be due to differences in how SDE emulates control-flow transitions compared to hardware-enforced shadow stack behavior.

Further attempts to investigate this were inconclusive. Running the culpable test cases in SDE with only SHSTK (without IBT) caused a `SIGSEGV` before execution could even begin, preventing a clean comparison. This highlights a limitation of using SDE for evaluation: while useful for simulating unsupported hardware features, it can introduce behavioral mismatches or stability issues. Moreover, it is difficult to confirm the existance of an interaction here.

4.2 Case Studies

The following sections will discuss the exploits found and replicated on ExploitDB. In total, after the search strategy (outlined in Section 3.3.2), 64 entries remained. Of those entries, 22 are kernel exploits, and 42 are application exploits. Table 4.3 shows the dis-

Setup	4.2.1	4.2.2	4.2.3	4.2.4	4.2.5	4.2.6
(baseline)	E	E	E	E	E	E
(stkpro)	E	E	E	NV	NV	NV
(dforti)	E	E	E	NV	NV	NV
(shstk)	-	-	E	NV	NV	NV
(ibt)+(shstk)	-	-	E	NV	NV	NV
(stkpro)+(dforti)	E	E	E	NV	NV	NV
(stkpro)+(dforti)+(shstk)	-	-	E	NV	NV	NV
(stkpro)+(dforti)+(shstk)+(ibt)	-	-	E	NV	NV	NV

Table 4.4: Results from the reproduced case studies using various combinations of hardening flags. The column headings are the associated case study section numbers. SDE emulation was used to collect all data, apart from the case studies in Section 4.2.1 and 4.2.2. E = Exploited, NV = Not Vulnerable, '-' = Not Applicable.

tribution of weaknesses for each remaining entry. Notably, after ignoring uncategorised exploits, 45% of kernel exploits and 30% of application exploits are memory errors, demonstrating the prevalence of memory errors in exploits. The reproduction effort was not limited to just memory error CVEs, to make the subset more representative of real exploits.

Reproduction of the entries occurred in two passes. The first pass omitted kernel exploits and looked at application exploits involving memory errors, to quickly get a set of entries that were relevant to the hardening flags. The second pass went chronologically through entries, to obtain a set of entries that were more representative of seen exploits. Of 14 attempted exploits, 5 application exploits were successfully reproduced, and 1 kernel exploit was reproduced. The following sections present case studies of the successfully reproduced exploits. Each case study presents the exploit, a account of the reproduction effort, and then a short description of the impact of the hardening flags. For some of the simple exploits, the explanation has been omitted in the interest of space. The final case study, outlined in Section 4.2.7, provides an example of a unsuccessful attempt, to provide context as to the difficulties involved in reproduction. Table 4.4 provides a summary of the results of each mitigation combination on each case study. Appendix **??** shows some example output from the case studies. `TODO`

4.2.1 CVE-2022-0847 - Dirty Pipe

Dirty Pipe is the name of an exploit that impacted Linux kernels from version 5.8 to 5.10.102, 5.15 to 5.15.25 and 5.16 to 5.16.11. CVE-2022-0847 is a flaw within the page handling structures of the Linux pipe mechanism. The vulnerability received a CVSS v3.1 base score of 7.8 (High), and can allow an unprivileged user to write to any file that they have read only permissions on. There are multiple exploits available for this vulnerability, but the Exploit-DB entry is the original one published by Max Kellerman []. CVE-2022-0847 was patched by properly initialising the vulnerable `flags` variable when creating a new `pipe_buffer` structure. The entry on Exploit-DB comprises just the exploit code, with an external link to an explanation of the exploit. This case study was successfully reproduced as reported on Exploit-DB.

4.2.1.1 Exploit Analysis (adapted from [18])

CVE-2022-0847 exploits optimizations within the pipe mechanisms in the Linux kernel. Pipes are data channels that allow inter-process data communication. They are unidirectional, and are implemented as circular queues, with data being chunked in non-contiguous pages in memory. Each pipe consists of a ring of `pipe_buffer` structures, shown in Listing 4.1. Each `pipe_buffer` holds a pointer to a page in memory, alongside a `flags` variable that stores certain properties about the associated page. As a space optimisation, if the previous page's `pipe_buffer` struct has the `PIPE_BUF_FLAG_CAN_MERGE` flag set, subsequent writes will attempt append to the page, in order to utilise all remaining space. This prevents the kernel from unnecessarily allocating a page for the write. Under normal operation, this does not cause any problems.

```
[..]
26 struct pipe_buffer {
  struct page *page;
  unsigned int offset, len;
  const struct pipe_buf_operations *ops;
  unsigned int flags;
  unsigned long private;
};
[..]
```

Listing 4.1: Source file: `include/linux/pipe_fs_i.h`

However, when the `splice` system call is used, arbitrary writes can occur. The `splice` system call sends a file to a pipe by first loading the file into the page cache, and then passing the page reference to the pipe. This reference is then added to the pipes circular queue, in a new `pipe_buffer` structure. This page should not be written to, as it would update the source file directly, without checking permissions.

`splice` uses the library functions `copy_page_to_iter_pipe()` and `push_pipe()` to add a `pipe_buffer` structure. These functions do not initialise the `pipe_buffer->flags` variable, which means that the object can erroneously have the `PIPE_BUF_FLAG_CAN_MERGE` flag set. If there is space left on the cached page, the next write using `pipe_write()` can end up merging into the cached page. If this is set up correctly a user can splice a file with read only permissions, and write arbitrarily to any location within the file.

The actual exploit goes through the following steps:

1. Create a pipe.
2. Fill the pipe with random data, setting the `PIPE_BUF_FLAG_CAN_MERGE` flag in all `pipe_buffer` structures as user has write permissions on self-created data.
3. Drain the pipe (leaving the flag set in all structure `pipe_buffer` instances in the circular queue).
4. Splice data from the target file, opened with read permissions.
5. Write arbitrary data into the pipe; this data will overwrite the cached file page instead of creating a new anomalous structure `pipe_buffer` because `PIPE_BUF_FLAG_CAN_MERGE` is set

4.2.1.2 Reproduction

The entry on Exploit-DB did provide any environment details, instead just stating a vulnerable kernel version. Therefore, a substantial amount of effort was required to setup and reproduce this exploit. First of all, the kernel had to be built from source with necessary modifications to allow hardening the culpable file. Secondly, a valid boot image had to be created, which allowed binaries to run. Finally, demonstrating the exploit visibly on the kernel required further effort. The result of this successfully reproduced the exploit as reported on Exploit-DB. The following subsections discuss the process I used to achieve each of these steps.

Building the kernel: The chosen kernel version was version 5.8.0 [], as this was a full release that was within the range of vulnerable programs. Docker was used to ensure a consistent build platform for the kernel, and GCC 10.2.1 was used to build the kernel, with GLIBC 2.31. These versions were chosen as newer versions of GCC and GLIBC failed to successfully build the kernel. To allow compiler flags to be easily applied to the Linux kernel, the Linux Makefile was instrumented, to add in the ability to compile some files with a specified hardening flag. Default configurations were used to reproduce this exploit's platform, generated using `make defconfig`.

Initial RAM disk: For the Linux kernel to boot, it requires an initial filesystem (referred to as a initial RAM disk) to boot into. To aid in generating a minimal initial RAM disk, Busybox 1.34.1 was used, which provides basic binaries for a Linux system like `mv` or `cp`. I wrote a script to setup the rest of the file system, as well as writing a script to properly initialise the kernel, and then packaged this into a format the kernel would recognise. Creating an initial RAM disk this way requires that all binaries on the kernel have to be built statically, as GLIBC is not included. Therefore, the exploit was built statically in the Docker container, and then added to the initial RAM disk.

Exploiting the Kernel: Ensuring that the exploit could run on the kernel, and demonstrate an actual effect required some manipulation of it. Two wrapper scripts were written to test the functionality of the exploit. The first script uses the Exploit-DB exploit to overwrite the write-protected `etc/passwd` file, thereby creating a new user with root privileges. The second script makes a copy of the `passwd` file and then checks to see if has changed, to validate the success of the exploit.

4.2.2 CVE-2019-18634 - Sudo

Sudo is a utility widely employed on Unix-like systems to grant limited administrative privileges in a controlled manner. CVE-2018-18634 is a vulnerability from version 1.7.1 in the way Sudo parses passwords when password feedback is turned on with `pwfeedback`. This vulnerability received a CVSS v3.1 base score of 7.8 (High), and can have serious impact, allowing a user to gain a root shell. CVE-2018-18634 was patched in version 1.8.30, by updating the password parsing code to account for write failures. The entry on Exploit-DB provides a PoC exploit script, with no environment details. The NVD incorrectly states that this is a stack-based buffer overflow, but it is actually an overflow in the BSS segment.

4.2.2.1 Exploit Analysis

When Sudo's feedback functionality is turned on, a crafted input can cause an overflow. This is caused by a flaw in Sudo's `getln()` function, highlighted in Listing 4.2. If the program fails to write a backspace character to the password source (line `396`) when handling a line erase character, it will reset the buffer size (line `400`), but does not reset the current point to point to the head of the buffer (added line `+1+`. This leads to a situation where the current pointer points *beyond* the end of the buffer. By itself, a large enough write out of bounds causes a segmentation fault in Sudo, but there is the possibility of exploitation.

```
[..]
368 static char *
369 getln(int fd, char *buf, size_t bufsiz, int feedback,
370     enum tgetpass_errval *errval)
371 {
372   size_t left = bufsiz;
373   ssize_t nr = -1;
374   char *cp = buf;
[..]
394       } else if (c == sudo_term_kill) {
395         while (cp > buf) {
396           if (write(fd, "\b \b", 3) == -1)
397             break;
398           --cp;
399         }
+1+         cp = buf;
400         left = bufsiz;
401         continue;
[..]
417       *cp++ = c;
[..]
```

Listing 4.2: Source file: `src/tgetpass.c`, Version `1.8.25`. Lines added via patch are indicated with a prepended **+n+** to the line, where **n** provides a reference for each patch line.

This overflow occurs on the BSS segment in memory, which resides underneath the heap, and contains static variables. The BSS segment is structured like the stack, with variables residing in contiguous addresses in memory. Figure 4.1 shows the layout of the space next to the buffer. The exploit overwrites from the `buffer` to the structure `user_details`, which stores the user id of the calling user, among other information. By overwriting with zeros, the exploit effectively gives the attacker root permissions (`uid=0`.

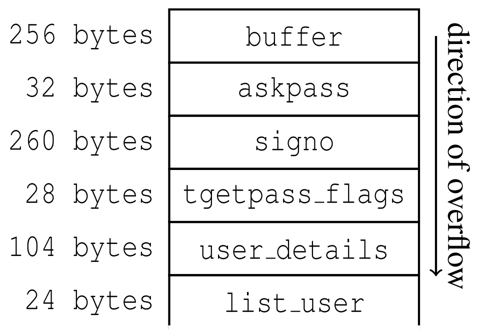


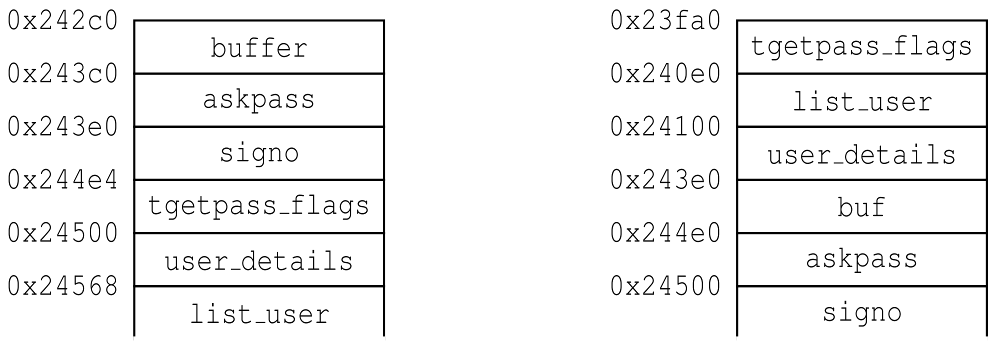
Figure 4.1: Structure of BSS segment at time of the function call. The important variables are `buffer` and `user_details`.

4.2.2.2 Reproduction

The exploit on Exploit-DB did not provide any environment details, instead just providing the source code of the exploit. To replicate the exploit, Ubuntu 20.04 was used as the base Docker image, and GCC 9.5.0 and GLIBC 2.31 were used to build Sudo. Finding the correct GCC version required a substantial amount of effort, as the exploit did not work with other GCC versions, and there was no indication of what the failure was. Moreover, no sources mentioned that the exploit only worked when Sudo was built with certain versions of GCC, instead just citing vulnerable versions of Sudo. This made replicating this exploit difficult.

This problem was caused by differences in how versions of GCC organise the BSS segments, To demonstrate this, we can compare the locations of the variables on the BSS segment between one of the initial versions, GCC 10.2.1, and the correct version GCC 9.5.0. Using `objdump` to inspect the compiled `sudo` executable, we see the results in Figure 4.2. As shown, the two versions organise the BSS segment differently to each other. Notably, Sudo compiled with GCC 10.2.1 is not actually exploitable as the `user_details` structure is located before the buffer, so cannot be overwritten by an overflow.

Currently, there are no sources that specify the platform, or mention the differences in BSS structure. Therefore, finding the correct GCC version for the exploit required a substantial amount of debugging and testing after I realised that this was the issue. With the correct version of GCC, the exploit functions as specified on ExploitDB.



(a) Structure of BSS segment when compiled with GCC 9.5.0

(b) Structure of BSS segment when compiled with GCC 10.2.1

Figure 4.2: Comparison of the BSS segments between GCC versions.

4.2.3 CVE-2023-24626 GNU Screen

GNU Screen is a terminal multiplexer that allows users to manage multiple shell sessions within a single terminal window. CVE-2023-23626 is a flaw in `socket.c` until version 4.9.0 that allows local users to send privileged SIGHUP signals. The vulnerability is rated with a CVSS v3.1 base score of 6.5 (Medium), due to the limited scope of the exploit. The issue was addressed by modifying the signal handling in the 'socket.c' file to properly check the permissions of the signal sender. The entry on Exploit-DB comprises a Python script that leverages the vulnerability to send a SIGHUP signal to a specified PID, demonstrating the potential for privilege escalation. This case study was successfuly reproduced as reported on Exploit-DB. Due to the simplicity of the exploit, an analysis is not provided.

4.2.3.1 Reproduction

Reproducing this exploit required some manual configuration of the build process, as well as further setup to effectively demonstrate the exploit. The entry on Exploit-DB just provides the exploit and software version, with no versioned environment. Building and installing GNU Screen was simple, GCC 13.3.0 and GLIBC 2.39 were used to build the binary. The build file was instrumented as before to apply specific hardening flags.

The exploit shared in the entry required some minimal modification to work inside the Docker environment. To visually demonstrate the exploit, I wrote a Python listener script that prints when it receives a SIGHUP signal. Further to this, I wrote a Bash script to automate the process of checking whether the exploit was successful.

4.2.4 EDB-44331 - Crashmail

CrashMail II is a more portable successor of CrashMail, a utility originally developed for Amiga systems that automates the process of collecting and sending crash reports. In 2018, an exploit was published on Exploit-DB that demonstrated a stack-based buffer overflow in CrashMail II from version v1.6. There is no available CVSS v3.1 score, and this vulnerability has not been patched downstream. The entry on Exploit-DB just provides an example exploit, and a link to the version of CrashMail affected. There are no attached environment details, but this case study was partially reproduced.

4.2.4.1 Exploit Analysis

CrashMail II has a stack buffer overflow in the `LockConfig` function that is called during argument parsing. If a filename larger than the buffer size is passed into `LockConfig`, it will get written to the stack, writing arbitrary data out of bounds. This can be exploited to overwrite the return address of the function and setup a ROP chain. As CrashMail II runs with root privileges, any processes opened by the ROP chain also have root privileges. Therefore, a ROP chain that executes `execve('\bin\sh')` will spawn a root shell. Listing 4.3 highlights the vulnerable code.

```
[..]
bool LockConfig(char *file)
[..]
   char buf[200];
   osFile fp;

   strcpy(buf,file);
   strcat(buf,".busy");
[..]
   if(!(fp=osOpen(buf,MODE_NEWFILE)))
   {
      printf("Failed to create lock file %s\n",buf);
      return(FALSE);
501     }
[..]
```

Listing 4.3: Source file: `src/crashmail.c`. The buffer is initialised in line 481 and then overflown in line 484 due to no bounds checking.

4.2.4.2 Reproduction

Reproducing the reported exploit proved to be incredibly challenging. However, the original vulnerability could be demonstrated as a segmentation fault. To build the source code, a Docker container with GCC 10.2.1 and GLIBC 2.39 was used. The entry on Exploit-DB did not provide any environment details, so Debian Bullseye was used. Moreover, after reading through the exploit, it appears to be aimed at either 32 bit Linux, or Crashmail cross-compiled with the `-m32` flag. As it is marked as a 64 bit exploit by its category on Exploit-DB, it is likely the later. Regardless, after cross-compilation, the exploit still failed to produce a root shell.

An investigation of the available ROP gadgets with Ropper failed to find most of the gadgets used in the exploit. A snippet of this is shown in Figure 4.3. Due to the lack of these gadgets, the exploit could not successfully run. This could be due to the compiler version being different to the one used to build the target for the exploit on ExploitDB. It may be the case that changing the compiler version causes the gadgets to break as they generate different assembly code from the same source code. The entry on Exploit-DB does not specific the compiler version used. Despite this, the exploit does demonstrate the original vulnerability, overwriting the return address, and causing a segmentation fault.

```
11c62570b102# ropper
(ropper)> file crashmail
[INFO] Load gadgets for section: LOAD
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] File loaded.
(crashmail/ELF/x86)> search pop eax;
[INFO] Searching for gadgets: pop eax;
[INFO] File: crashmail
0x0001c2d0: pop eax; add al, 0; add byte ptr [ecx], bh; ret 0x773;
0x0001acff: pop eax; add al, byte ptr [eax]; add al, ch; ret 0xfe7e;
0x0001046d: pop eax; jp 0x1046f; call dword ptr [eax - 0x73];
0x0000442a: pop eax; sbb eax, 0; add byte ptr [eax], al; nop; pop ebp;
    ret;
0x0001ab24: pop eax; xchg bh, bh; call dword ptr [eax + 0x6a];
(crashmail/ELF/x86)>
```

Figure 4.3: The first stated ROP gadget of the chain is a clean `pop eax; ret;`, however a search for this was fruitless

4.2.5 EDB - MiniFTP

MiniFtp is a minimalist FTP server implemented in C, providing basic file transfer capabilities in lightweight environments. A flaw in MiniFtps `parseconf_load_setting()` can cause an overflow when parsing malicious configuration files. There is no official CVSS score and no downstream patch is currently available. The exploit available on Exploit-DB consists of a Python based exploit, and a link to the vulnerable code with an example tested distribution. This exploit was partially reproduced, demonstrating the vulnerability but not spawning a root shell as described.

4.2.5.1 Exploit Analysis

MiniFTP has a stack buffer overflow in the `parseconf_load_setting()` function that is called during configuration parsing. As part of the initial parsing, the function will split the `setting` string into a key value pair using `str_split()`. Critically, `str_split()` does not perform bounds checking, so both the `key` and `value` buffer can be overflow if the `setting` string is too long. This can be exploited to overwrite the return address of the function and setup a ROP chain. As MiniFTP runs with root privileges, any processes opened by the ROP chain also have root privileges. Listing 4.4 highlights the vulnerable code.

```
[..]
void parseconf_load_setting(const char *setting){
  while(isspace(*setting)) setting++;
  char key[128] = 0, value[128] = 0;
  str_split(setting, key, value, ’=’);
  if(strlen(value) == 0){
  fprintf(stderr, "missing value in config file for : %s\n", key);
  exit(EXIT_FAILURE);
}
[..]
```

Listing 4.4: Source file: `parseconf.c`. Function `parseconf_load_setting()`

4.2.5.2 Reproduction

The entry on ExploitDB provides only the exploit, and test platform, it does not indicate the build platform, which is crucial in this case. The exploit is shellcode stored on the stack, which is thwarted by labelling the stack as non-executable. This is on by default in GCC, but the entry does not mention that the build process had to be changed. Moreover, similar to Section **??**, the exploit is written for 32 bit Linux after closer inspection. Regardless, the initial vulnerability can be reproduced by passing in a large enough string into the `setting` string.

4.2.6 CVE-2018-18957 - libIEC61850

libIEC61850 is an open-source library that provides the server client protocols GOOSE, MMS and SV. CVE-2018-18957 is a stack-based buffer oveflow in libIEC61850 from version 1.3, specifically the implementation of the GOOSE protocol . It received a

CVSS v3.1 base score of 9.8 (Critical) due to how exposed the buffer was to a potential attacker, and how simple it would be to exploit. The vulnerability was patched in version 1.6 by replacing a faulty `strcpy` with `strncpy`, to ensure bounds checking. An exploit for this CVE has not been seen in the wild. The entry on ExploitDB provides a simple PoC of the vulnerability, demonstrating the buffer overflow, but not exploiting it. This case study was succesfully reproduced with a Docker environment and correct versioning. As there is no attached exploit, no analysis is provided.

4.2.7 CVE-2021-22555 - Netfilter

Netfilter is a framework within the Linux kernel, added in version 2.6.19, which allows the inspection and modification of network packets. In 2021, a heap out-of-bounds write was discovered in the `iptables` and `ip6tables` modules when converting 32 bit structures to 64 bit. This could lead to privilege escalation or denial of service. CVE-2021-22555 received a CVSS v3.1 base score of 7.8 (High), and affected versions 2.6.19-4.4.266, 4.5-4.9.266, 4.10-4.14-230, 4.15-4.19.187, 4.20-5.4.112, 5.5-5.10.30 and 5.11-5.12. This was patched by removing the line responsible for writing too many zeros out of bounds. The entry on Exploit-DB comprises the exploit source, alongside listing the OS versions that the exploit was tested on. This exploit was not successfully reproduced, but it provides a useful case study of the difficulties in reproducing exploits.

4.2.7.1 Exploit Analysis

When `iptables` set/replace functions are used in compatibility mode, `iptables` structures need to be converted from 32 bit to 64 bit so they can be used by the native kernel. The function `xt_compat_target_from_user` provides this functionality, and this is where the vulnerability occurs, highlighted in Listing 4.5. In line 1129, when `memset()` is called with `target->targetsize` as an offset, it can write a few bytes of `'\x00'` out of bounds. This line intends to correctly align the resultant structure in memory, but does not check if the padding goes outside of the intended bounds.

```
[..]
1112 void xt_compat_target_from_user(struct xt_entry_target *t, void
   **dstptr,
1113                 unsigned int *size)
1114 {
1115    const struct xt_target *target = t->u.kernel.target;
1116    struct compat_xt_entry_target *ct = (struct
   compat_xt_entry_target *)t;
1117    int pad, off = xt_compat_target_offset(target);
[..]
1121    t = *dstptr;
1122    memcpy(t, ct, sizeof(*ct));
[..]
1127    pad = XT_ALIGN(target->targetsize) - target->targetsize;
1128    if (pad > 0)
1129        memset(t->data + target->targetsize, 0, pad);
[..]
```

Listing 4.5: Source code file: `net/netfilter/x_tables.c`, Function: `xt_compat_target_from_user()`

When the structure is of a larger size, this out of bounds write can increase, as the size of the padding is larger (line 1127). This leads to more consistent overwrites. The exploit listed on Exploit-DB is quite complicated, going through a few separate steps to obtain a root shell. Therefore, the following just illustrates how to achieve a Use-After-Free or dangling pointer, to demonstrate the vulnerability.

To achieve a Use-After-Free, the heap overflow needs to overflow some other object on the heap, in such a way that a dangling point is created. The first step is getting a target object allocated next to the overflow. To achieve this, the exploit creates a large amount of Sys V `msg_msg` structures on the heap, as they are allocated in the same segments as `iptable` structures. By spraying the heap with these structures at consistent intervals, and then deallocating one, the location of the `iptable` structure on the heap can be controlled when it gets created in `xt_compat_target_from_user`, shown in Figure 4.4. As shown, this writes the `next` pointer to zero bytes, which normally points to the next message in the queue, but here will now point instead to the first message in the heap. Freeing the first message will now create a Use-After-Free.

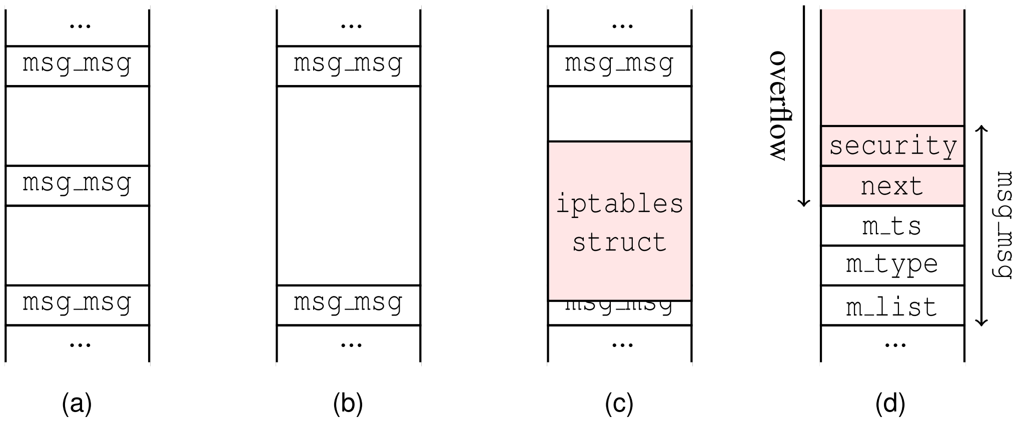


Figure 4.4: The step-by-step of causing an adjacent overwrite on the heap. In (a) the heap is sprayed, then a hole is made in (b) and finally when `xt_compat_target_offset` is called, the bytes overrun in the `msg_msg` header, shown in (c). (d) shows the 0 byte overflow

4.2.7.2 Reproduction

This section discusses the work that went into attempting to reproduce this exploit. Unfortunately, no experimental results can be presented, but this does provide a case study into the difficulties of reproducing CVEs from the source code. Similar to Section 4.2.1, this entry is a kernel exploit, which requires building the Linux kernel from source to provide the target. Unlike Section 4.2.1, this was an exploit in a module packaged with the kernel, which made replication quite complicated. Moreover, as discussed below, this exploit requires the target to have non-default configurations to be vulnerable. These details are not mentioned on the entry on Exploit-DB, or the associated CVE entry in the NVD.

The initial attempts followed the same process outlined in Section 4.2.1: finding a vulnerable kernel version, building just the base image with default configurations and then attempting to exploit it. However, I soon realised that this was not sufficient as the exploit would not run correctly within QEMU. At this point, I found through debugging that the particular vulnerable blocks had to be included in the build by altering the Linux build configurations. More debugging indicated that the exploit code required the `msg_copy` function to turned on with `CONFIG_CHECKPOINT_RESTORE` and `CONFIG_-PROC_CHILDREN`. After more debugging and enabling user namespaces, the exploit ran without errors but did not spawn a root shell as specified.

At this point, I realised that this may be caused by the ROP chain in the exploit not working with the offsets of this minimal kernel. To solve this, I looked at the source package for the specified Ubuntu version in the entry (Ubuntu 5.8.0-48-generic). This kernel was then built with the default configurations with all modules. Even with the exact version of the kernel with the default configurations, the exploit did not work. The same kernel with the configurations identified previously activated, still did not work. Unfortunately, without further information, it appears impossible to reproduce this exploit from source.

Chapter 5

Discussions and Conclusions

5.1 Summary

In this study, I have created a extensible framework for reproducing and researching exploits from minimal code. I used this framework to reproduce 5 application level exploits, and 1 kernel level exploit, indicating the flexibility of the framework. Further to this, the RIPE and RecIPE benchmarks were modified to allow testing of modern hardening features. These results were used to test whether increasing combinations of hardening measures actually results in increasing levels of security.

5.1.1 Benchmarks

I found that benchmark results were as expected in most cases - increasing combinations of hardening measures led to increasing security. Notably however, on the RecIPE benchmark, three more test cases were exploited with full Intel CET measures than with only Intel Shadow Stack. This could be due to the former being tested with Intel SDE and then latter being run directly on hardware. The actual cause could not be determined, as the culpable test cases did not run with only Intel Shadow Stack enabled when emulated with SDE. This meant a comparable test platform could not be reached, so it could not be determined whether this was caused by emulating with SDE or by some interaction between the hardening measures.

Other than this discrepancy, results were as expected, combinations of hardening measures proved more effective than any single measure. In the RIPE framework we see that stack canaries thwart 45% of test cases, fortify source thwarts 66.8% of test cases, Shadow Stack thwarts 11.0% of cases, and full CET thwarts 33.6% of cases. This compares with the RecIPE framework, where stack canaries thwart 39.1% of test cases, fortify source thwarts 34.4% of test cases, Shadow Stack thwarts 31.1% of cases, and full CET thwarts 29.3% of cases. Notably, fortify source is more effective than stack canaries on the RIPE benchmark versus the RecIPE benchmark. On both benchmarks the combinations of measures are more effective than any single measure, however the measures are in general much more effective on the RIPE benchmark than the RecIPE benchmark. This can be seen with the best combination (stkpro+forti+shstk+ibt), which thwarts 72.6% of test cases on RIPE in comparison to 43.9% on RecIPE.

5.1.2 Case Studies

I attempted 14 reproductions, and reproduced 6 case studies. Of the 6 case studies, 3 were mitigated by any combination of hardening measure. The results also show that increasing combinations of hardening measures remain effective. The remaining 3 case studies, CVE-2022-0847, CVE-2019-18634, and CVE-2023-24626 were not mitigated by any combination of flags. The main problem in this study was reproducing results from the information available. None of the exploits were ‘plug-and-play’ i.e. simple to set up. Each one required comprehensive research, often collating information from multiple different sources. In part, this was due to many entries on Exploit-DB not providing any build information when the vulnerability was caused, or exposed by a deviation from standard build procedure. Multiple exploits also required further manual editing to demonstrate effectively, or adjust for the particular build platform. In addition to this, many problems were due to inaccurate information, such as with the Sudo case study (Section 4.2.2), where multiple sources claims that the vulnerability is a stack overflow [20, 23, 37], when it was an overflow in the BSS segment. If this was the case, the version of GCC should not matter as they organise the stack the same, but the version did matter. This led to extensive debugging based on incorrect information, making replicating the exploit more difficult. This is not limited to just one case study, in the Netfilter case study (section 4.2.7), the NVD cites the wrong range of vulnerable kernel versions [32] which led to wasted effort on non vulnerable versions. In the Crashmail case study (Section 4.2.4), the reference ROP gadgets could not be located in the compiled binary files.

5.2 Discussion

The benchmark results demonstrate the effectiveness of each hardening measure in their own respect. In both benchmarks, almost all combinations of compiler flags are more effective than any single flag alone indicating the benefits of combining sets of hardening flags. However, the testing with the RecIPE benchmark indicate that in some cases compiling with full CET could open a program up to more attacks. This is *not* conclusive, as comparable results could not be produced upon further investigation due to platform restrictions. However, if this is the case, it could have far-reaching impact in the future. Intel CET is being touted as a new measure that can only improve security [], but any indication of it leading to more vulnerabilities should be taken seriously. Future research should investigate these potential interactions on a platform that can support both IBT and Shadow Stack natively, to determine the cause of the interactions.

The results from the case studies exemplifies the replication problem with exploits in security research. The normal formulation of sharing exploits for research either as a single script, or a blog post describing the problem theoretically is not adequate for further research. The framework outlined in this study not only produces exemplars to demonstrate exploits, but also allows future work to investigate these exploits beyond the scope of this particular study. By providing a containerized environment which is standardized where possible, exploits are effectively future-proofed. Future researchers can use the same case study images to produce results that are not limited to just this research question, e.g. applying new hardening measures. Moreover, it remains repeatable between different base machines and setups, allowing for consistency with results.

Finally, the differences between RIPE and RecIPE in the final ranking of how effective the hardening measures are indicates that they may not be measuring the relative effectiveness of hardening measures accurately. If they were measuring the same attribute of security, they should produce somewhat comparable results. Instead RIPE over inflates the effectiveness of a hardening measure in comparison to RecIPE, and they disagree on which hardening measures are more effective than each other. This indicates that both may not be effective measures of the relative effectiveness of different flags. This finding should be taken into consideration for future research.

5.3 Evaluation of Project

This project set out to investigate whether combinations of security hardening flags in GCC interact in unintended ways, potentially reducing program security. While the central goal remained consistent throughout, there were changes in scope and methodology over time. Initially, the project also intended to evaluate the performance overhead of different hardening configurations using the SPEC benchmark suite. However, due to time constraints, this goal was not met. Despite this, the performance angle remains an interesting direction for future work.

In terms of overall project execution, I am satisfied with how it progressed, though it was significantly more time-consuming and technically involved than expected. The bulk of the effort went into three key areas:

Reproducible Exploit Framework: Creating a generalized, extensible Docker-based framework capable of rebuilding and demonstrating CVE exploits took considerable effort. Each exploit required careful manual research, environment recreation, and debugging. The lack of standardized, complete exploit metadata (particularly for older exploits) meant that reproducing even a single CVE often required weeks of reverse engineering and testing.

Benchmark Integration and Modification: RIPE and RecIPE both required modification to support modern GCC features such as CET (Shadow Stack and IBT). RecIPE in particular was poorly documented, and adapting it to run in emulated CET environments involved patching its runtime logic. This significantly extended the benchmarking phase but was essential to get meaningful results.

Data Collection and Analysis: Testing multiple hardening configurations across both benchmark suites and case studies generated a large volume of data. Organizing and interpreting this data, especially in light of occasional inconsistencies (e.g., with emulation vs. hardware results), demanded careful validation and iteration.

Throughout, I maintained a strong focus on reproducibility and transparency. Every result is backed by a containerized build that can be rerun with minimal configuration, shown in Appendix `TODO`. This not only meets the project goals but also contributes a reusable research tool for others in the field.

One major difficulty encountered was the unreliability and inconsistency of exploit metadata on platforms like Exploit-DB and NVD. In many cases, builds failed not because of code issues, but because the environment details were incomplete or incorrect. I also spent a substantial amount of time debugging unexpected behavior, such as exploits only working under specific GCC versions due to differences in memory layout (Section 4.2.2). These difficulties limited the size of the curated dataset, as each exploit took significant time to reverse enginner and test. However, the framework is structured to allow future researchers to extend this work easily.

Overall, this project achieved its primary objectives. The framework developed for reproducible exploit analysis functioned effectively, allowing a range of real-world exploits to be recreated in a consistent, standardized environment. This framework is an important contribution, as it addresses key reproducibility challenges in security research. From a benchmarking perspective, a thorough comparison of hardening measures across RIPE and RecIPE was achieved. This project highlights potential discrepancies with certain combinations of hardening measures, another consideration for future work.

5.4 Future Work

A key limitation of this study was the inability to test all combinations of CET features (IBT and Shadow Stack) on native hardware. As a result, the discrepancy on the RecIPE benchmark remains inconclusive. Testing these interactions on a platform with full hardware and OS support for CET would be a priority for follow-up research.

The reproducibility framework developed in this project could be extended to a larger number of exploits. While only a subset of filtered case studies was included due to time constraints, the process is scalable. A broader dataset allows for more robust conclusions, particularly around the generalizability of hardening effectiveness across real-world scenarios.

Understanding the trade-off between security and performance is essential for real-world deployment decisions. Originally, this project also aimed to evaluate the performance overhead introduced by different hardening combinations using the SPEC CPU benchmark suite. Although this was not included due to time constraints, it remains a valuable direction.

5.5 Limitations

I recognise the limitations in this project, which may impact its validity. First, while both RIPE and RecIPE attempt to simulate attacker behavior, neither benchmark fully captures the complexity of real-world exploits. The differing results they produce raise questions about their effectiveness as definitive measures of security. Second, the project was constrained by hardware and operating system limitations. In particular, some results relied on the Intel SDE to emulate CET features such as IBT. This reliance on emulation may have introduced discrepancies that would not be present on native hardware. Third, a substantial amount of effort was required to reproduce existing exploits, often due to poor documentation or inconsistencies in software versioning. These challenges limited the number of case studies that could be feasibly included in the analysis. Fourth, while the selected case studies provide representative data than the benchmarks, they still represent only a small subset of known vulnerabilities. As such, the findings may not generalize to other types of software or system configurations. Finally, the study focused exclusively on exploitation outcomes. It did not evaluate potential side effects such as performance degradation, compatibility issues, or undefined behavior resulting from flag interactions. These aspects warrant further investigation in future work.

5.6 Conclusion

`TODO: finish`

Bibliography

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[2] R. Al-Ekram and Kostas Kontogiannis. Source code modularization using lattice of concept slices. pages 195–203, 04 2004. ISBN 0-7695-2107-X. doi: 10.1109/CSMR.2004.1281420.

[3] Todd M Austin, Scott E Breach, and Gurindar S Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming Language Design and Implementation*, pages 290–301, 1994.

[4] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999, 2019. doi: 10.1109/SP.2019.00076.

[5] Marco Carvalho, Jared DeMott, Richard Ford, and David A Wheeler. Heartbleed 101. *IEEE security & privacy*, 12(4):63–67, 2014.

[6] CISA, FBI, ASD's ACSC, and CCCS. Exploring memory safety in critical open source projects. Technical report, Cybersecurity and Infrastructure Security Agency (CISA), Jun 2024. URL `https://www.cisa.gov/sites/default/files/2024-06/joint-guidance-exploring-memory-safety-in-critical-open-source-projects-508c.pdf`.

[7] Jonathan Corbet. Shadow stacks for 64-bit arm systems. *LWN.net*, August 2023. URL `https://lwn.net/Articles/940403/`. Accessed: 2025-03-30.

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

[9] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. Towards the detection of inconsistencies in public security vulnerability reports. In *28th USENIX security symposium (USENIX Security 19)*, pages 869–885, 2019.

[10] Ulrich Drepper. Security enhancements in redhat enterprise linux (beside selinux). Technical report, 2005. (2005).

[11] Google. A year in review of 0-days used in-the-wild in 2021, 2022. URL https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html.

[12] Open Source Security Foundation (OpenSSF) Securing Critical Projects Working Group. Set of critical open source projects. Technical report, 2023. URL https://github.com/ossf/wg-securing-critical-projects/tree/main/Initiatives/Identifying-Critical-Projects/Version-1.1.

[13] Intel. A technical look at intel's control-flow enforcement technology, 2020. URL https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html.

[14] Intel. Complex shadow-stack updates (intel control-flow enforcement technology), 2023. URL https://www.intel.com/content/www/us/en/content-details/785687/complex-shadow-stack-updates-intel-control-flow-enforcement-technology.html.

[15] Intel Corporation. 11th gen intel® core™ mobile processors product brief, 2020. URL https://www.intel.com/content/www/us/en/products/docs/processors/core/11th-gen-core-mobile-processors-brief.html. Accessed: 2025-03-30.

[16] Yuancheng Jiang, Roland HC Yap, Zhenkai Liang, and Hubert Rosier. Recipe: Revisiting the evaluation of memory error defenses. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, pages 574–588, 2022.

[17] Frederick Boland Jr. and Paul Black. The juliet 1.1 c/c++ and java test suite. (45), 2012. doi: 10.1109/MC.2012.345.

[18] Max Kellerman. The dirty pipe vulnerability, 2022. URL https://dirtypipe.cm4all.com/.

[19] Microsoft. A proactive approach to more secure code, 2019. URL https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/.

[20] MITRE. CVE-2019-18634: Stack-based buffer overflow in Sudo when pwfeedback is enabled, 2020. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-18634. Accessed: 2025-04-01.

[21] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 919–936, 2018.

[22] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[23] National Institute of Standards and Technology. National Vulnerability Database: CVE-2019-18634, 2019. URL `https://nvd.nist.gov/vuln/detail/CVE-2019-18634`. Accessed: 2025-04-01.

[24] National Institute of Standards and Technology. National Vulnerability Database, 2025. URL `https://nvd.nist.gov/`. Accessed: March 31, 2025.

[25] Offensive Security. Exploit Database (ExploitDB). URL `https://www.exploit-db.com/`. Accessed: 2025-03-25.

[26] Openwall Project. Openwall (Security Project Site). URL `https://www.openwall.com/`. Accessed: 2025-03-25.

[27] Aleksandra Orlowska, Christos Chrysoulas, Zakwan Jaroucheh, and Xiaodong Liu. Programming languages: A usage-based statistical analysis and visualization. In *Proceedings of the 4th International Conference on Information Science and Systems*, pages 143–148, 2021.

[28] Hilarie Orman. The morris worm: A fifteen-year perspective. *IEEE Security & Privacy*, 1(5):35–43, 2003.

[29] Packet Storm Security. Packet Storm Security (Exploit and Security Tool Archive). URL `https://packetstormsecurity.com/`. Accessed: 2025-03-25.

[30] Red Hat. Red Hat Bugzilla (Bug Tracking System). URL `https://bugzilla.redhat.com/`. Accessed: 2025-03-25.

[31] Hubert Rosier. Ripe64, 2018. URL `https://github.com/hrosier/ripe64`.

[32] Bonan Ruan, Jiahao Liu, Chuqi Zhang, and Zhenkai Liang. Kernjc: Automated vulnerable environment generation for linux kernel vulnerabilities. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 384–402, 2024.

[33] Lalit Sharma and Neeraj Goel. Risc-v based secure processor architecture for return address protection. In *2025 38th International Conference on VLSI Design and 2024 23rd International Conference on Embedded Systems (VLSID)*, pages 481–486, 2025. doi: 10.1109/VLSID64188.2025.00095.

[34] Linus Torvalds. Merge tag x86_shstk_for_6.6-rc1, 2023. URL `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=df57721f9a63e8a1fb9b9b2e70de4aa4c7e0cd2e`. Linux kernel commit df57721f9a63e8a1fb9b9b2e70de4aa4c7e0cd2e.

[35] B. Traynor. The risc-v instruction set manual volume i: Unprivileged architecture, chapter 35. control-flow integrity (cfi). https://github.com/riscv/riscv-isa-manual/releases/tag/riscv-isa-release-f122839-2025-03-27, March 2025. URL `https://github.com/riscv/riscv-isa-manual/releases/tag/riscv-isa-release-f122839-2025-03-27`. Accessed: 2025-03-30.

[36] Arjan van de Ven. New security enhancements in red hat enterprise linux v. 3, update 3. *Raleigh, North Carolina, USA: Red Hat*, 2004.

[37] Joe Vennix. Sudo 1.8.25p - 'pwfeedback' Buffer Overflow (PoC), 2020. URL `https://www.exploit-db.com/exploits/47995`. Accessed: 2025-04-01.

[38] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, pages 1–7, 2012.

[39] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: Runtime intrusion prevention evaluator. In *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*. ACM, 2011.

[40] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. Warpattack: bypassing cfi through compiler-introduced double-fetches. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1271–1288. IEEE, 2023.

[41] Jianhao Xu, Kangjie Lu, Zhengjie Du, Zhu Ding, Linke Li, Qiushi Wu, Mathias Payer, and Bing Mao. Silent bugs matter: A study of compiler-introduced security bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3655–3672, 2023.

[42] Hengkai Ye and Hong Hu. Too Subtle to Notice: Investigating Executable Stack Issues in Linux Systems. In *Proceedings of the 32nd Network and Distributed System Security Symposium (NDSS 2025)*, San Diego, CA, Feb 2025.

[43] Peter Zijlstra. x86/Kconfig: Enable kernel IBT by default, 2022. URL `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4fd5f70ce14da230c6a29648c3d51a48ee0b4bfd`. Linux kernel commit 4fd5f70ce14da230c6a29648c3d51a48ee0b4bfd.