jamk.fi

Evaluating Approaches for Detecting and Eliminating Memory Safety Errors in Linux Kernel Programming

Ilja Sidoroff

Master's Thesis
June 2019
School of Technology, Communication and Transport
Degree Programme in Information Technology
Cyber Security

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s) Sidoroff, Ilja	Type of publication Master's Thesis	Date June 2019
		Language of publication English
	Number of pages 77	Permission for web publication: yes

Title of publication

Evaluating Approaches for Detecting and Eliminating Memory Safety Errors in Linux Kernel Programming

Degree programme

Information Technology

Supervisor(s)

Mika Rantonen

Mika Karjalainen

Assigned by

-

Abstract

Memory Safety means that a program cannot access unintended memory regions. Lack of memory safety continues to be a major source of security related software errors.

The problems arising from the use of memory unsafe C programming language are reviewed, both in general and Linux kernel programming context. Various methods for detecting and eliminating memory safety problems are then evaluated. Methods chosen for testing were static and dynamic analysis, and using a memory safe language as a programming language. The methods were tested during a process of writing a Linux kernel module.

Unfortunately, none of the methods tested proved to be a comprehensive solution to the problem of memory unsafety. Each method had their own strengths. Static analysis is easy to include in the development process; however, it does not detect problems very efficiently. Dynamic analysis, on the other hand, is good at finding bugs; yet it requires manual testing. Memory safe languages are very promising; however, they would require significant, changes to the existing code which can be difficult to achieve in practice both due to economic and social reasons.

Keywords/tags

Memory Safety, Spatial and Temporal Memory Safety, C-programming Language, Rust Programming Language, Linux Kernel Programming

Miscellaneous

Kuvailulehti

Tekijä(t) Sidoroff, Ilja	Julkaisun laji Opinnäytetyö, ylempi AMK	Päivämäärä Toukokuu 2019
		Julkaisun kieli Englanti
	Sivumäärä 77	Verkkojulkaisulupa myönnetty: kyllä

Työn nimi

Muistiturvallisuudesta johtuvien virheiden havaitseminen ja ehkäisy Linux-kernel-ohjelmoinnissa

Tutkinto-ohjelma

Insinööri (YAMK), Kyberturvallisuus

Työn ohjaaja(t)

Rantonen, Mika (JAMK)
Karjalainen, Mika (JAMK)

Toimeksiantaja(t)

-

Tiivistelmä

Muistiturvallisuus tarkoittaa sitä, että ohjelma ei voi käyttää väärää muistialuetta. Muistiturvallisuuden puute on edelleen merkittävä tietoturvaa vaarantavien ohjelmistovirheiden lähde.

Opinnäytteessä tutkittiin ongelmia, jotka aiheutuvat C-ohjelmointikielen muistiturvattomuudesta, niin yleisellä tasolla kuin Linux-kernel ohjelmoinnin-yhteydessä. Lisäksi tutkittiin erilaisia menetelmiä, joilla muistiturvattomuutta voidaan havaita ja ehkäistä. Tutkittavat menetelmät olivat staattinen ja dynaaminen analyysi, sekä muistiturvallisen ohjelmointikielen käyttäminen. Menetelmiä testattiin Linux-kernel modulin ohjelmointiprosessissa.
Valitettavasti mikään testatuista menetelmistä ei osoittautunut kaikenkattavaksi ratkaisuksi muistiturvattomuuden aiheuttamiin ongelmiin. Jokaisella testatulla menetelmällä oli omat vahvat puolensa. Staattinen analyysi voidaan integroida helposti ohjelmistokehitysprosessiin, mutta se ei havaitse ongelmia kovin tehokkaasti. Dynaaminen analyysi on tehokas menetelmä virheiden löytämiseen mutta edellyttää erillistä testausta. Muistiturvalliset ohjelmointikielet ovat erittäin lupaavia, mutta niiden käyttö edellyttäisi suuria muutoksia olemassa olevaan koodiin, mikä voi olla käytännössä vaikeaa sekä taloudellisisten että sosiaalisisten tekijöiden vuoksi.

Avainsanat

Muistiturvallisuus, Paikallinen ja ajallinen muistiturvallisuus
C-ohjelmointikieli, Rust-ohjelmointikieli, Linux-kernel -ohjelmointi

Muut tiedot

Contents

Figures

Tables

1 Introduction

1.1 Motivation and Overview

Operating systems and their kernels are one of the central points of attacks in the computing environment. They control access to the underlying hardware, and with hardware assistance, they also enforce the separation between user space processes, and privileged kernel code. If the operating system or its kernel is compromised or subverted, the applications running on the operating system can no longer guarantee their confidentially, integrity nor availability.

Current operating systems and their kernels are typically very complex. The Linux kernel currently consists of approximately 20 million (Löhner 2017) lines of C and assembly code. The complexity means that it is very difficult to ensure that there are no security critical bugs in the kernels (see below for a brief discussion on kernel security).

There have been various attempts to create kernels, which can provide more security guarantees. On the one end of the spectrum, there is seL4, which a formally verified microkernel (G. Klein et al. 2009). OpenBSD, on the other hand, is a more traditional, monolithic kernel and operating system, which states explicitly its goal to be security oriented (OpenBSD Developers 2017). However, the most popular and widely used operating system kernels remain to be based on the traditional monolithic and complex architecture. The most prominent examples are various versions of Microsoft Windows and Linux kernels, but even Apple's macOS and iOS, although partly based on Mach microkernels, share the same monolithic architecture (Chisnall 2010).

Since the operating systems landscape moves slowly and current operating systems are not likely to be replaced by e.g. seL4, OpenBSD or other variants, securing the current operating systems will probably have more short term utility, than creating new systems, and there is interest in exploring and implementing techniques, which can make the traditional systems more secure.

The purpose of this thesis is to review and compare some of the mitigations that can be used to eliminate some of the bug classes in the Linux kernel. Linux was chosen as the target of this study for two reasons. First, following the growing number of security critical bugs in the Linux kernel, there has been an increasing interest in developing new solutions for eliminating different error classes, and not just single bugs (see below for more discussion). Secondly, since the Linux kernel is free and open source software, testing these solutions is much easier than it would be in the case of Microsoft Windows or Apple macOS.

1.2 Security in Linux Kernel Development

Safety and security were previously widely considered as the advantages of GNU/Linux operating system over the competition, which consisted mostly of different versions Microsoft Windows. Versions of Windows and other Microsoft products contained indeed a plethora of exploitable vulnerabilities during the 1990s and 2000s. Whether this was due to the overwhelming popularity of Windows over Linux, which allowed for bugs to be found more easily, or better code quality in Linux, is difficult to estimate and open to debate.

The perception of the security situation has, however, changed. Already in 2002, Microsoft began to divest resources in making the Windows operating system more secure (Howard & Lipner 2003) and has since implemented many exploitation mitigation techniques in the operating system, e.g. Address Space Layout Randomization, Data Execution Protection (Howard 2006). Hall, Méndez, and Lich (2017) provide an overview of the current situation.

Even though Linux also implements many of the same, quite standard exploitation mitigation techniques (Stack canaries, Writable or executable memory, Address space randomization etc.), and has two different mandatory access control mechanisms, SELinux (Smalley, Vance, & Salamon 2001) and AppArmor (AppArmor Authors 2017), the security performance of the GNU/Linux has not been as stellar as its former reputation would imply. There has been a steady flow of exploitable security vulnerabilities in the Linux kernel, and the perception of the kernel development community's attitude to security problems has grown negative.

Torvalds and other prominent kernel developers have stated on several occasions that they do not consider security to be something that needs special attention from the kernel community. An often-cited quote from Torvalds summarises this attitude well: “I personally consider security bugs to be just ’normal bugs’. I don't cover them up, but I also don't have any reason what-so-ever to think it's a good idea to track them and announce them as something special” (Andrews 2008).

It is difficult to say if the general attitude to security or the inherent complexity of the Linux kernel have been contributing factors to high impact security vulnerabilities, found in the kernel. For a recent example, see e.g. exploitable bug in `unsafe_put_user` function (Corbet 2017b), tracked by CVE-2017-5123.

In 2015, Washington Post published a feature article on the perceived faults on kernel developer's attention to security problems, which brought out the issues in Linux kernel security to wider audience. (Timberg 2015). The article succeeded in creating controversy. It also prompted efforts, which try to increase the security of the kernel. One of the efforts is called Kernel Self Protection Project (KSPP), which was announced by Kees Cook in November 2015 (Cook 2015). The aims of the project include taking a more pro-active approach to security, and instead of fixing security critical bugs as they are found, the project tries to eliminate whole bug classes that can appear in the Linux kernel. (Corbet 2015). The project also tries to incorporate its changes directly in the mainline kernel, unlike an earlier Grsecurity/PaX project (Grsecurity 2017), which while providing several advanced hardenings as a separate patch, has never kept high priority in trying to merge them into mainline kernel (Edge 2009).

KSPP and Grsecurity/PaX try to harden the kernel systematically, and in the best case, hope to eliminate whole bug classes. However, there has been another approach to improve the quality of kernel code by improving toolings for finding programming errors. This work has been carried out both within the academic and kernel communities. Examples for such work include Trinity system call fuzz tester (Jones 2013), used for random testing of Linux system calls, Kernel Address Sanitizer KASAN, Linux Kernel Developers (2017) for detecting out-of-bounds memory accesses and lately, an implementation of Control Flow Integrity for kernel (kCFI; Moreira, Rigo, Polychronakis, & Kemerlis 2017). There are also efforts to utilize protections provided by newer hardware, in e.g. Reshetova, Liljestrand, Paverd, and Asokan (2017). A more complete list of security oriented initiatives and tools, although not exclusively limited to Linux kernel in scope, is found in (ibid., 7).

1.3 Research Problem and Methodology

In this thesis, the focus is on the problems caused by the memory unsafety. Memory safety is defined below; however, it can be noted here, that the term is used in inclusive meaning, namely, when using the term memory safety, on the occasion, problems are discussed which either arise as a result of memory unsafety (e.g. format string bugs), or problems, which can be used in the conjunction of memory unsafety problem (e.g. integer overflows or information leaks). This slightly inaccurate terminology is used in order to avoid cumbersome expressions such as “memory safety and related issues”. The context should always provide enough information to discern the exact meaning of the terms used.

Initially, the author wanted to concentrate on the security problems dealing with the lowest levels on computing, i.e. operating system interfacing with the hardware. During the writing of this thesis, two serious bug classes surfaced, which invalidated the security assumptions about trusted hardware. *Meltdown* and *Spectre* attacks showed that microprocessors cannot isolate the memory from reading when there are multiple execution processes running at the same time (Kocher et al. 2019; Lipp et al. 2018). This discovery opened a new avenue for research in memory safety problems. Spectre and Meltdown related hardware problems are not discussed in this work to keep the already large scope manageable; however, it is noted that even if all the security issues discussed in this work were solved, to achieve complete memory safety, also the issues brought forward by the two discoveries must be solved. This in the author’s opinion underlines the importance of the research in the area of memory safety issues.

The structure of the thesis is the following. In this introductory chapter, a general overview of the state of security in Linux kernel development is given.

The second chapter provides a broad overview of different bug classes and low-level exploitation techniques, which are enabled partly by lack of memory safety in C programming language. Exploitation techniques and their mitigations are surveyed widely. The third chapter discusses how the exploits presented in the previous chapter can manifest in the Linux kernel context. The first three chapters form the theoretical part of this thesis.

The fourth chapter describes a simple kernel module written by the author, which illustrates some of the bug classes described in the previous chapters. Some static and dynamic analysis techniques are then tested, to see if they can detect the bugs implanted in the module, written in C language, using the normal kernel tooling. Finally, Rust, a relatively new memory safe language is used to rewrite the kernel module. The then analyzed if this approach is generally feasible. This chapter forms the practical part of the thesis. Finally, conclusions are presented in the chapter five.

While discussing the exploitation of the memory unsafety quite widely in the chapters two and three, the exploitation techniques are not discussed in the practical part of the thesis, again to keep the scope of the thesis manageable. Some targets are, however, included here for exploitation to the code produced in conjuction with this thesis and the reader is encouraged to study the exploitation possibilities.

The research methodology used in this thesis has two parts. The theoretical part of the thesis follows a systemization of knowledge (SoK) approach. Even though there is much literature on different bug classes caused by memory unsafety and exploitation techniques, the study attempts to comprehensively summarize most of them, including their mitigations. A minor contribution of this thesis is to gather information about the applicability of the memory safety bugs to the Linux kernel context, and discussion of the mitigations currently in place. The second part of the thesis follows experimental, prototyping, methodology. As simple prototype as possible is built, which should contain the properties under study. The author tries to inductively form knowledge from the study of this prototype. The novel contribution in this thesis is an experience report of using Rust code in Linux kernel module.

Finally, a note on used terminology follows. Usually *operating system* means a program that manages and abstracts hardware resources and provides system calls to user space programs; whereas *operating system kernel* is the part of the operating system that deals with the hardware. However, in the Linux community, *Linux kernel* is often used of the whole the operating system program, and operating system in conjunction with Linux, is used to denote also user space utilities tightly coupled with the kernel, such as C library. The term Linux kernel is used in the latter sense. (Linux Information Project 2017)

2 Memory Safety as Security Problem

2.1 Definition of Memory Safety

Informally, a violation of memory safety usually means usually a buffer or stack overflow, caused by absence of bounds checking in programs written in C, C++, or assembly programming language. Security bugs caused by buffer overflows have usually high impact, and can allow arbitrary code execution in remote targets. One of the earliest examples of a malicious program using memory safety violations, is Morris Worm from 1988, which used a buffer overflow in `fingerd`-daemon to gain remote code execution (Spafford 1989). Different types of memory safety

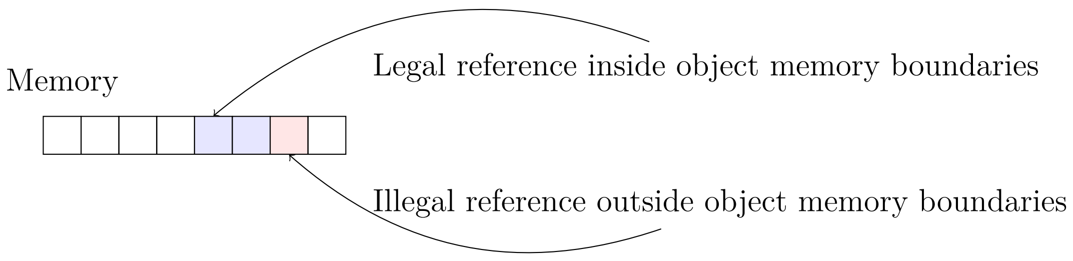

Memory
Legal reference inside object memory boundaries
Illegal reference outside object memory boundaries

Figure 1. Spatial memory safety violation

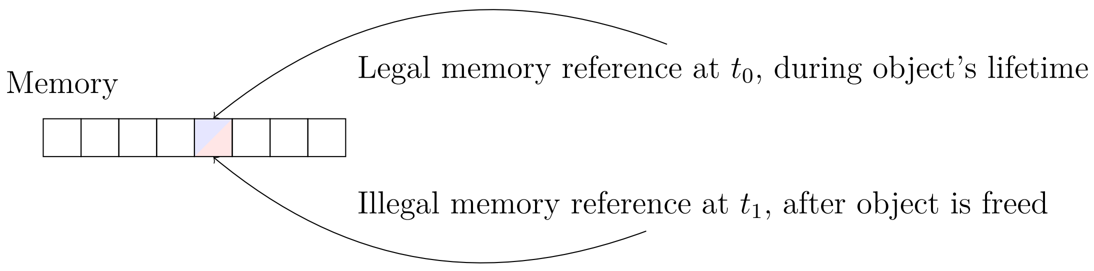

Memory
Legal memory reference at t_0, during object's lifetime
Illegal memory reference at t_1, after object is freed

Figure 2. Temporal memory safety violation

errors have since caused countless security incidents, and memory safety bugs are still found regularly. For recent examples, there is e.g. CVE-2018-6789 (2018), a buffer overflow in popular Exim-mailer program, or a heap overflow in Windows operating system CVE-2018-0825 (2018), both allowing remote code execution. An overview and history of memory corruption bugs is provided in Meer et al. (2010).

The term *memory safety* is used various way in the literature and industry. This thesis follows Szekeres, Payer, Wei, and Song (2013) and defines two cases of memory safety: *spatial* and *temporal* memory safety. A violation of spatial memory occurs when a pointer dereferences memory outside of the memory area allocated to the object (cf. Figure 1).

A temporal memory safety violation occurs, when the pointer dereferences object's memory area correctly, but after the object has been released or freed (cf. Figure 2). There are two common types of temporal violations are *use after free* and *double-free* referring to library function `free`, which is used to free memory in C language. (Szekeres et al. 2013, section II.A, Note: refers only to use after free).

Violations of memory safety can lead to multiple kinds of vulnerabilities. Reading memory outside of proper application context can lead to violations of confidentiality, whereas writing can lead to loss of integrity. Illegal memory accesses most of the time also leads to the crashing of the program, which can lead to a loss of availability.

Memory safety violations can occur, when the programming language does not validate memory accesses or defers the tasks of managing memory to the programmer. In exchange for the loss of safety, the programmer gets more performance and can control memory usage directly. Non-memory safe languages are typically used for performance reasons. They also can offer direct, non-abstracted access to the underlying hardware. Specifically, operating system kernels, such as kernels of Linux and Windows, are written in non-memory safe languages, in this case C, C++ and assembly, making them susceptible to memory safety violations.

In the rest of this chapter, the focus is on C programming language used in Linux kernel, and an overview of its unsafe features is given, which can directly or indirectly cause memory safety violations. Then the main bug classes related to memory safety in the general setting, and their possible mitigations are covered. Finally the chapter discusses how the bug classes manifest themselves in the Linux kernel concentrating on the kernel specific features and variations.

2.2 Unsafe features of C Programming Language

Writing secure programs in C programming language (Kernighan & Ritchie 1978) has proven to be quite a difficult task. The C programming language was developed from BCPL and B languages the main purpose of which was to facilitate writing compilers and operating systems usually earlier written in assembly language (Ritchie 1993, 2). C was first used to write successfully utilities for UNIX operating system under development in the Bell Labs and later, when it proved to be successful in this task, the whole operating system was rewritten in C (Ritchie & Thompson 1974, 366). Due to its and UNIX's relatively easy availability and its good portability, C language soon became the language of choice in the systems and low-level programming.

C language, however, was not written with safety in mind and does not give spatial or temporal memory safety guarantees, when used incorrectly. The main drivers in C's evolution were features, which were deemed necessary in implementing UNIX utilities, and later the whole operating system, which its authors freely admit (Ritchie 1993, 6-8). For instance, C's descendance from unityped BCPL and B languages resulted in rather weak typing system in the C language itself – early C language did not check types of structure members or function signatures (Ritchie 1993, 9-10) and even the latest ISO C standard (C11) allows much freedom in casting between types (*Information technology - Programming languages - C* 2011, chapters 6.3 and 6.3.2.3).

Due to C's informal evolution, which took place partly due to a creation of compiler and standard library ports to different processor architectures, later standardisation process also became difficult. While the overall language was the same under different architectures and compilers, the behavior for various corner cases, such as integer overflows and pointer arithmetic differed between implementations, and in many cases, it was deemed better to leave varying behavior unstandardized, or in more familiar terms, *undefined*, *unspecified* or *implementation-defined* behavior (*Information technology - Programming languages - C* 2011, chapters 3.4.3, 3.4.4, 3.4.1).

Undefined behavior is a code execution setting for which the standard does not impose any requirements how the code should be compiled to executed at the runtime. For instance, the standard does not define, what happens when the result of an integer calculation is too big to fit to the allocated data type. In practice, this means that the compiler or runtime system can handle the compilation or execution in any way it deems desirable.

Unspecified and implementation defined behaviors are more restricted forms of relaxing specification, which are typically less problematic in the terms of predictability. When encountering unspecified behavior, the compiler or runtime system can choose between some possible behaviors and in the case of implementation defined behavior, the implementor is mandated to specify the behavior by themselves. An example of unspecified behavior is evaluation order of function arguments, and an example of implementation specific behavior is the propagation of the high-order bit in signed integer right-shift (*Information technology - Programming languages - C* 2011, chapters 3.4.3, 3.4.4, 3.4.1).

Another example of undefined behavior is division by zero. In x86 processor architecture, division by zero will result in hardware exception. However, in PowerPC processor architecture, the processor will ignore the instruction (X. Wang et al. 2012, chapter 2.1). By leaving the language behavior undefined, the C standard committee makes the language implementation easier, when for instance, PowerPC architecture is not required to generate code for checking potential divisions by zero.

```
static unsigned int tun_chr_poll(struct file *file, poll_table *wait)   1
     {                                                                  2
        struct tun_file *tfile = file->private_data;                    3
        struct tun_struct *tun = __tun_get(tfile);                      4
        struct sock *sk = tun->sk;                                      5
        unsigned int mask = 0;                                          6
                                                                        7
        if (!tun)                                                       8
            return POLLERR;                                             9
```

Figure 3. Eliminated null pointer check in Linux kernel

Initially, C's trade-off between simpler implementations and full semantical definitions was deemed acceptable or even beneficial to the language, because besides easier portability, undefined behavior enabled better performance in some cases (Lattner 2011a). However, as compilers evolved and became more efficient in optimizing C and C++ programs, undefined behavior combined with aggressive optimizations produced can lead to surprising code generation and even potentially introduce security vulnerabilities (Lattner 2011b; X. Wang, Zeldovich, Kaashoek, & Solar-Lezama 2013, chapter 2.3).

An example of a security vulnerability introduced by an optimizing compiler is shown in Figure 3. The example is from Linux kernel 2.6.30. In TUN tunneling network device driver (`/dev/net/tun`), there is a potential NULL pointer dereference, which causes later check for NULL pointer to be optimized away. More specifically, the pointer `tun`, which gets assigned in line 4 and dereferenced in line 5, causes the compiler erroneously to believe that the value of `tun` cannot be zero, since dereferencing NULL is undefined behavior and should not happen (*Information technology - Programming languages - C* 2011, chapter 6.5.3.2). However, since the compiler now believes that `tun` is not zero, it can optimize away the NULL pointer check in lines 8-9. This error can then be exploited further for privilege escalation (Corbet 2009; CVE-2009-1897 2009).

2.3 Stack Corruption and Arbitratry Code Execution

The most serious class of security vulnerabilities that the C language makes possible are illegal memory accesses, often via buffer under- or overflow. Unifying the concept of arrays and memory pointers was actually one of the innovations of the C language (Ritchie 1993, 7). However, when combined to the fact that the array access is not checked for boundaries, this leads to the fact; that access-

```
#include <stdio.h>
#include <string.h>

void restricted()
{
    // execution should not normally come here
}

void vulnerable (const char *attacker_controlled)
{
    char  buffer[8];
    strcpy(buffer, attacker_controlled); // allows overwriting return
         address
}

int main (int argc, char **argv)
{
    vulnerable(argv[1]);
}
```

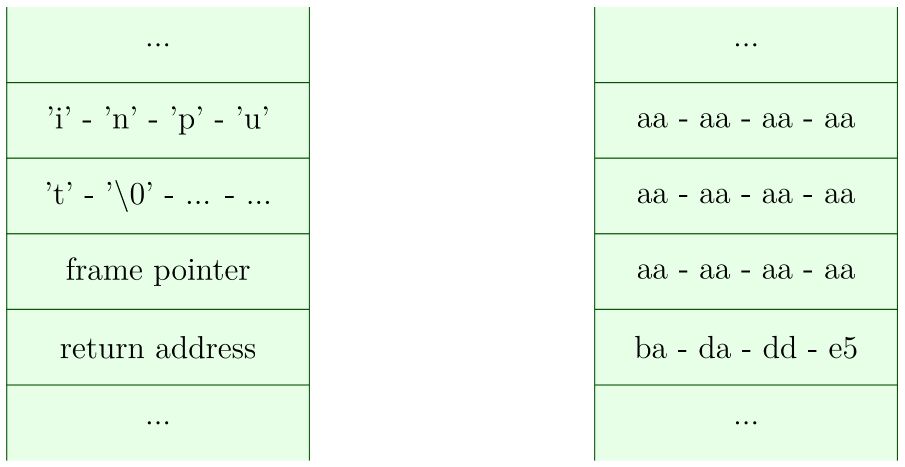
Figure 4. An example of stack overflow

ing non-intended memory locations or creating spatial or temporal memory access violations is very easy to do either by accident or by design.

There are many variants of buffer overflow exploits. A hacker using pseudonym Aleph One popularized a stack corruption exploit strategy, which uses a stack buffer overflow, which can allow an attacker to execute arbitrary code (Levy 1996).

A simple example of a C program vulnerable to stack buffer overflow is shown in Figure 4. The function `vulnerable` allocates 8 bytes of memory from the stack to variable `buffer` and fills it with contents of string `attacker_controlled` given to function as a parameter, which is user supplied. The return address of the function, along with the frame pointer, which is not important in this case, resides also in the stack, below the `buffer`-variable. Since neither the C language, nor the example program does not do automatic bounds checking, function will accept strings longer than 8 bytes, and allow spatial memory access violation. The lack of bounds checking allows user to supply input that overwrites function's return address, for example with address to `restricted`-function, where the programs execution would continue.

Figure 5 illustrates the described situation showing the contents of the stack during different program executions. In the first case (a), the program is passed a string `input`, which occupies 6 bytes of memory (five for the text and sixth byte for



(a) Normal stack **(b)** Overflown stack

Figure 5. Stack overflow

```
#include <unistd.h>

int main() {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

Figure 6. A program spawning shell

string terminator). Six bytes fit in the allocated memory and the program runs as expected. However, in the second case (b), the program is passed an at least 24-byte long string. In the example, the input contains first 18 bytes with hexadecimal value 'aa', followed by function's new return address (ba-da-dd-e5, ignoring here the endianness issues).

In this example, stack corruption was used to subvert the program control flow. Stack corruption can be also used in other ways to gain attacker supplied code execution. One could, for instance, supply input data, which could be interpreted as machine code instructions. The function's return address should be overwritten to be the starting address of the user supplied 'program', which gets executed when the function returns.

For example, the following program in Figure 6 can be considered. The program spawns a shell and does nothing else. The code is slightly modified version of Aleph Null's original example (Levy 1996).

When compiled to machine code, the program can translate for instance into fol-

```
main:
  0:    55                      pushq   %rbp
  1:    48 89 e5                movq    %rsp, %rbp
  4:    48 83 ec 20             subq    $32, %rsp
  8:    64 48 8b 04 25 28
        00 00 00                movq    %fs:40, %rax
  11:   48 89 45 f8             movq    %rax, -8(%rbp)
  15:   31 c0                   xorl    %eax, %eax
  17:   48 8d 05 00 00 00
        00                      leaq    (%rip), %rax
  1e:   48 89 45 e0             movq    %rax, -32(%rbp)
  22:   48 c7 45 e8 00 00
        00 00                   movq    $0, -24(%rbp)
  2a:   48 8b 45 e0             movq    -32(%rbp), %rax
  2e:   48 8d 4d e0             leaq    -32(%rbp), %rcx
  32:   ba 00 00 00 00          movl    $0, %edx
  37:   48 89 ce                movq    %rcx, %rsi
  3a:   48 89 c7                movq    %rax, %rdi
  3d:   e8 00 00 00 00          callq   0 <main+0x42>
  42:   b8 00 00 00 00          movl    $0, %eax
  47:   48 8b 55 f8             movq    -8(%rbp), %rdx
  4b:   64 48 33 14 25 28
        00 00 00                xorq    %fs:40, %rdx
  54:   74 05                   je      5 <main+0x5B>
  56:   e8 00 00 00 00          callq   0 <main+0x5B>
  5b:   c9                      leave
  5c:   c3                      retq
```

Figure 7. Assembly code for shell-spawing program (GCC 7, GNU/Linux, x86_64)

```c
char shellcode[] = "\x55\x48\x89\e5 .../bin/sh";
```

Figure 8. Shellcode as Cstring

lowing form, where assembly code is on the right, and corresponding opcodes, or program representation in the memory, on the left in Figure 7. The memory representation can be converted to following string representation in Figure 8, which is usually called *shellcode* (Levy 1996). Jump instruction must be added, which directs the program flow to the start of our shellcode in the place of function's return address. If one then inputs the constructed shellcode to our program, instead of returning from function `vulnerable`, one executes the code, which spawns a shell for user.

To make this attack actually work against the given code, there are a few technical details. First, since function `strcpy` is used to copy the shellcode, it cannot contain any `NULL`-bytes, which act as string terminators. This can be accomplished by slightly modifying the code spawning the shell. Second, one needs to be able to place the jump to the start of the shellcode at the right position in the stack, which can be done by calculating the stack layout in the attacked machine, and padding the shellcode with `NOP`-instructions (so called *NOP-sled*). An explanation of technical details can be found in Levy (1996).

There are many proposed mitigations against stack corruption, the most common and generic mitigation techniques are outlined here. An overview of specific defenses is presented in e.g. Cowan, Wagle, Pu, Beattie, and Walpole (1999) and Kuperman, Brodley, Ozdoganoglu, Vijaykumar, and Jalote (2005).

The first mitigation technique is called *stack canary*. It works by writing a special value before the function's return address. This special value is checked before returning from the function, and if the check fails, the program aborts. Figure 9 presents an example of stack layout with canary. Canary value can either be a value which is difficult to write by usual exploit techniques (e.g. a null or cr/linefeed character) or a random value generated when the program starts and stored in a location, which is difficult for the attacker to access (Cowan et al. 1999, chapter 3.4.2).

A downsize of stack canary is a slight performance penalty, which means that canary protection might not be turned on for all functions (Edge 2014). There are ways of bypassing stack canary. The first and simplest way is to try to brute

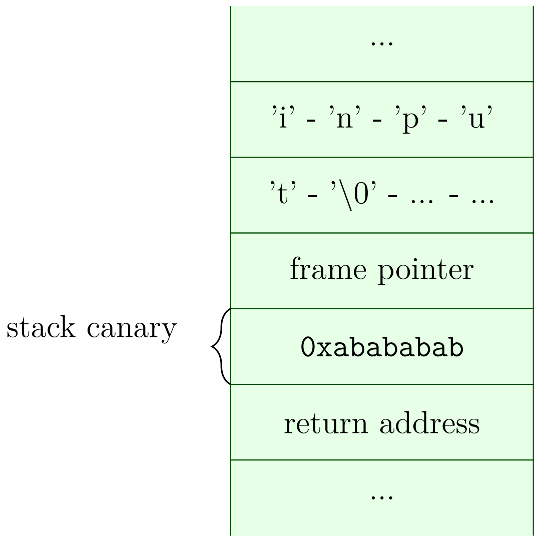


Figure 9. Stack canary

force the stack canary value as long as the attack succeeds, which can be feasible in 32-bit architectures, but is not generally doable in a 64-bit environment. Another way is to utilize information leaks, which can reveal the stack canary values. Third, the most generic way, is to avoid overwriting the stack canary at all, or to rewrite it after it first has been overwritten as further discussed in e.g. Strackx et al. (2009).

Another mitigation approach is to make the execution of the shellcode more difficult. One way of doing this is to randomize the address space layout, i.e. *address space layout randomization, ASLR*, so that the attacker cannot know in advance what the memory addresses of running programs or shared libraries are. Originally developed for Linux by PaX team (PaX Team 2001), Address space layout randomization is now implemented in most current systems. Address space layout randomization does not make the execution of the shellcode impossible but rather forces the attacker to try several times to mount a successful attack, which can potentially allow easier detection of attacks. The effectiveness of address space layout randomization is hampered by the low amount of entropy some 32-bit systems can use to generate different address layouts. In some cases, the address space layout can only get 8 bits of entropy (i.e. 256 different choices), which is easy to bypass with brute force techniques. In 64-bit systems the available entropy is much higher and address space layout randomization techniques are more effective. However, various information leaks about the memory structure can still reduce the effectiveness of address space layout randomization even in 64-bit systems (Marco-Gisbert & Ripoll 2014). To be effective, address space layout randomization requires that there are no information leaks, which give away information about randomized structures, and that the randomization is done with enough randomness or entropy (Jang, Lee, & Kim 2016).

Both stack canary and address space layout randomization are probabilistic mitigation techniques, which means that the attacker cannot be guaranteed to absolutely fail. A third mitigation technique, which does not rely on random data, is to make stack and possibly other memory areas either executable or writable. This means that if a malicious process can inject shellcode into memory, it can only do it to such memory areas, which cannot be used in executing the program code. Conversely any executed program code cannot be modified by a malicious program. This technique is called *Write XOR execute* or $W \oplus X$. The first software implementations were RedHat's *ExecShield* (van de Ven 2004) and OpenBSD's W^X (de Raadt 2004). W $\oplus$ X can also be implemented in hardware if the processor supports NX (no execute) bit for page table or similar memory management objects (NX bit has various marketing names, e.g. Intel uses XD, ARM has eXecute Never, XN bit, while AMD's term is Enhanced Virus Protection, EVP). NX protection is reasonably fast compared to software implementations, however, it is unfortunately incompatible with just-in-time-compilation method, which is used e.g. in modern Javascript and Java implementations, which sometimes causes situations where it cannot be fully used (Szekeres et al. 2013, section II.B). Another way of circumventing NX is to reuse already executable code in the program, which is further discussed below. A related mitigation, *Relocation read only, relro*, done at link time, should also be mentioned. In relro, when binaries are loaded before execution, some (*partial relro*) or all possible sections *full relro*) of the binary are marked read only (Vermeulen 2011). Unfortunately, full relro is seldom used due to high runtime penalty, and the mitigation partial relro gives is limited (T. Klein 2009).

A final countermeasure mentioned here is *control flow integrity* (CFI). It means controlling the program's control flow in such a way, that any unexpected deviations from the usual program control flow are detected and prevented. We will also discuss this mitigation below when discussing code reuse attacks, since while control flow integrity is in principle effective against stack buffer overflows, it aims for more general protection.

There are also other proposed and implements mitigations, such as *shadow stacks*, where function's return address is stored in a separate memory area, which the attacker cannot access. For this and details on other mitigations, see e.g. Burow, Zhang, and Payer (2018), Szekeres et al. (2013).

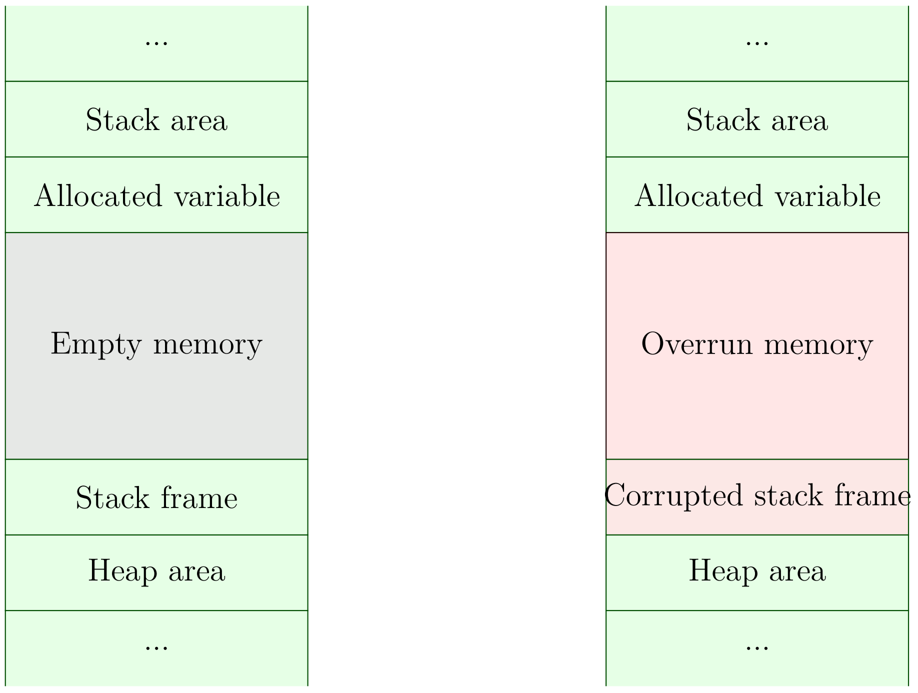
2.4 Heap misuse and corruption

Heap corruption or misuse is another vulnerability class in C programs. It is closely related to stack corruption, but takes place in the dynamically allocated memory, or heap. Whereas stack corruption attacks usually result in spatial memory access violations, corrupting or misusing the heap memory can result in both spatial and temporal violations.

Heap is the memory area that can be dynamically allocated and freed by using C functions `malloc`, `free`, `calloc` and `realloc`. The implementation of the heap is left unspecified in the standard (*Information technology - Programming languages - C* 2011, chapter 7.22.3), but is typically implemented as a contiguous memory area that is divided into blocks by user demand, as seen Figure 10 (case a) (Arpaci-Dusseau & Arpaci-Dusseau 2014, Chapters 14.1, 14.2, 14.3). Depending on the implementation, blocks also typically contain bookkeeping information on the state of the heap and addresses of adjacent blocks. Since there is no bounds checking when accessing dynamically allocated memory, it is possible to overwrite contents of adjacent dynamic variables, or even the stack area, since the stack and heap typically reside in the same memory area, as seen in Figure 10 (case b), leading to spatial memory access violation.

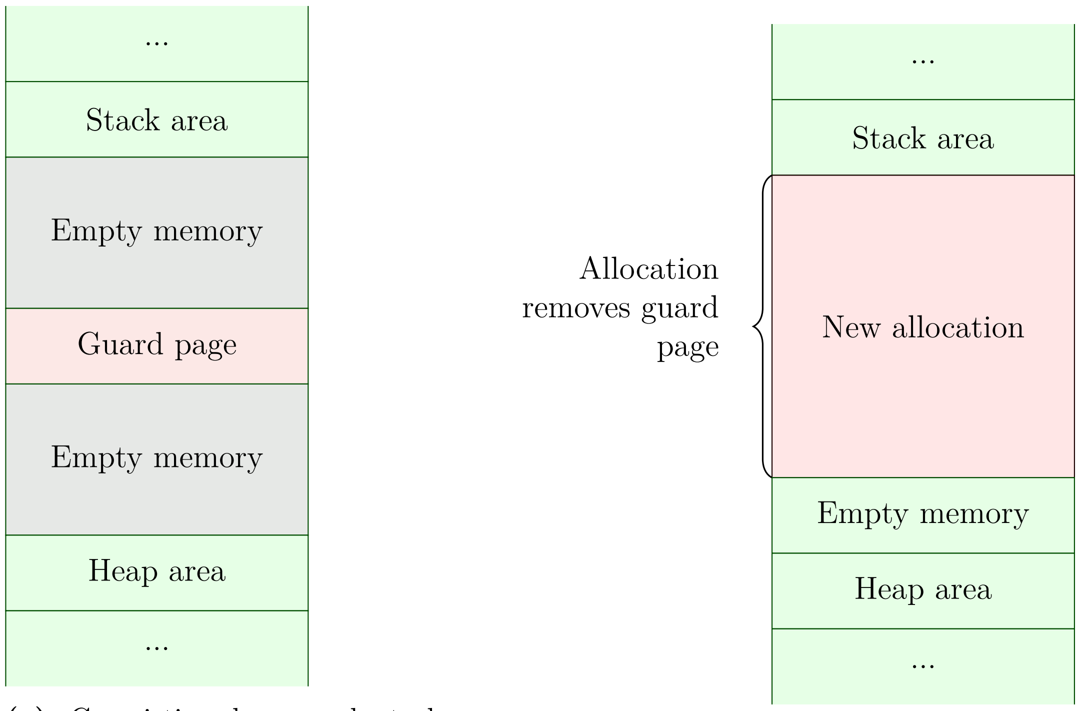
To prevent heap and stack collisions, one can insert a guard page between the memory areas, which will trigger a segmentation fault when accessed. However, this guard page can be bypassed in various ways, for instance forcing a new allocation for variable length array allocated in stack, or using `alloca` (nonstandard) C library function, when the new allocation is larger than the guard page, usually 4 or 8 kilobytes. Delalleau demonstrated these attacks in Delalleau (2005) and they are discussed in Qualys (2017a, 2017b) and illustrated in Figure 11. Bypassing guard page can be mitigated by increasing its size to make creation of the memory allocation more difficult, however, this does not remove the root cause of the vulnerability, which is using the same address space for stack and heap, without enforcing memory safety.

There is also another avenue for heap exploitation, which uses programming logic errors in allocating and freeing memory. *Use after free* error occurs when the application uses memory, which has already been released by the application. This



(a) Co-existing heap and stack **(b)** Stack overrun by heap

Figure 10. Heap and stack layout



(a) Co-existing heap and stack, with guard page **(b)** Stack overrun by heap

Figure 11. Heap and stack layout with guard page

```c
#include <stdio.h>
#include <stdlib.h>

struct user_t {
    unsigned int uid;
    unsigned int token;
};

void check_token(struct user_t *user) {
    if (user->token == 1) {
        printf("Token ok\n");
    } else {
        printf("Token not ok\n");
    }
}

int main(int argc, char** argv) {
    // user with 'token'
    struct user_t *user = malloc(sizeof (struct user_t));
    user->uid = 1;
    user->token = 1;

    check_token(user);

    // free allocated memory
    free(user);

    // memory is still usable
    check_token(user);
}
```

Figure 12. An example of use after free

situation can lead to program crash, or other attacker controlled code execution (CWE-416 2019). *Double-free* can occur, when the program frees the same allocated memory area more than once (CWE-415 2019). In this case, the internal bookkeeping structures of the heap can be manipulated so that the attacker can overwrite arbitrary memory locations. Double-free errors can be seen as a special case of errors or attacks against the memory allocator implementation. Exploitation of many of these errors require other incorrect uses of memory allocator functions and possibly spatial memory safety violations in the program in question. Even though the term *double-free* is often used as a catch-all term for these kinds of errors, which can be used to attacks the heap memory, they are referred to as *heap corruption errors* and discussed further below. Both use after free and heap corruption errors can be classified as temporal memory safety violations.

Figure 12 shows a simplified example of use after free bug. In the example program the user is allocated a memory entity with a ’token’, which remains valid even after the memory entity is freed. In realistic programs the actual errors and exploits are more complicated, but the idea is the same. For example Evans (2017) discusses more realistic examples of use after free errors in further detail.

Heap corruption vulnerabilities are much more complicated bugs to classify as stack overflows or use after free, since they depend on the implementation details of the memory allocators. However, in practice, there have been attacks on all widely used platforms, and exploits for the bug class seem to be widely generalizable.

The name double free comes from the situation, where the same memory object is incorrectly freed two times. The second invocation of the `free` function overwrites some non-heap memory, possible controllable by the attacker. There are many variations of these attacks, and they do not always rely on incorrect calls to `free`, but can occur when there is e.g. a buffer overflow, which allows overwriting some of the bookkeeping data in the heap.

Heap corruption was first popularized in Peslyak (2000) and the techniques were further developed e.g. in Anonymous (2001) and Phantasmal Phantasmagoria (2005). Here, an early attack is illustrated, which works on Doug Lea's malloc -implementation (dlmalloc; Lea & Gloger 1996), which has since been patched. However, even the current Linux glibc malloc implementation (ptmalloc; Loosemore, McGrath, Oram, & Stallman 2001) is based on Lea's work, and there are similar but more complicated current attacks (Xie, Zhang, Li, Liu, & Gu 2016).

Figure 13 shows slightly simplified memory layout of heap, as implemented in glibc, with three empty memory bins. Heap area is implemented as doubly-linked list of free memory bins with pointers to previous and next free bins. When memory is allocated, a pointer to a bin is given as a result of `malloc`-call, and adjacent free memory bin pointers are updated, as shown in Figure 14 (Lea & Gloger 1996).

If an attacker is able to modify the memory which contains either a pointer to a free bucket, by either overflowing a buffer located in adjacent memory location, or if the program contains a `free` function call to the same memory address, the attacker can cause the `free`-function to modify not a cell in the doubly-linked list, but another memory location of attacker's choosing.

The specific mechanism of the attack varies, however, for instance, if the attacker can modify the memory shown in red in Figure 14 and then cause some bin management operation to be triggered, memory bins are incorrectly marked as used, split or combined, and as a side effect, the operation will write an arbitrary value

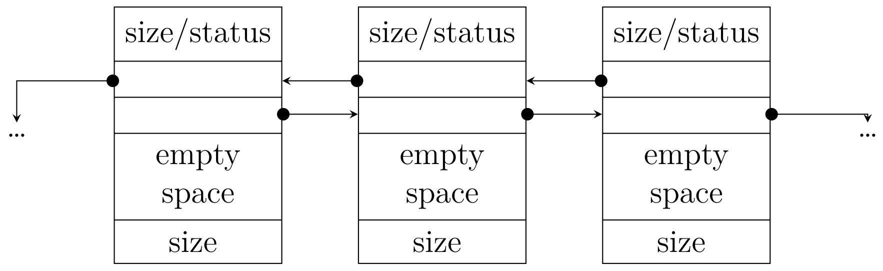


Figure 13. Heap with three empty bins

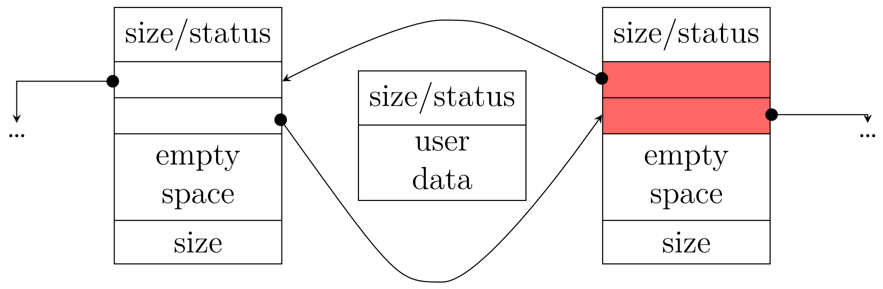


Figure 14. Heap with two empty and one used bin

to an arbitrary memory location. For instance Anonymous (2001), Kaempf (2001, 3), Phantasmal Phantasmagoria (2005) describe the details of the original and similar attacks.

The main ways of mitigating the heap corruption vulnerabilities is to ensure the integrity of the heap metadata, for instance with similar methods as stack canaries. Heap allocation can also contain randomness to make exploiting the corruption harder (Szekeres et al. 2013, sections V.A, VII.A). Heap exploitation is, however, highly implementation dependent and there is no generally applicable mitigation technique against it. M. Miller (2009) and M. Miller (2013) provide an overview of heap corruption problems on Windows platform. Eckert et al. (2018) informs about a static analysis technique for evaluating heap implementations from security viewpoint.

2.5 Other Types Common of Vulnerabilities

Buffer overflows and stack or heap corruption present common and well-known type of vulnerabilities, however, C language offers more techniques for creating memory safety violations. They are present them here in less detail as the stack and heap corruption. Some of them are used in the practical section of the thesis, and serve as an example of showing how difficult it is to combine the aspects of safety and perceived efficiency in the implementation of the programming languages.

Format string vulnerabilities is a class of vulnerabilities related to `printf`-family of functions (*Information technology - Programming languages - C* 2011, chapter 7.19.6). `printf` and its siblings can convert numerical and other values into strings for printing. The functions perform this by taking a format string that contains formatting directives that are placeholders for the printed values (e.g. `%d` for digits, `%c` for characters etc.). For each directive, the function takes a corresponding argument, and converts it to specified string format.

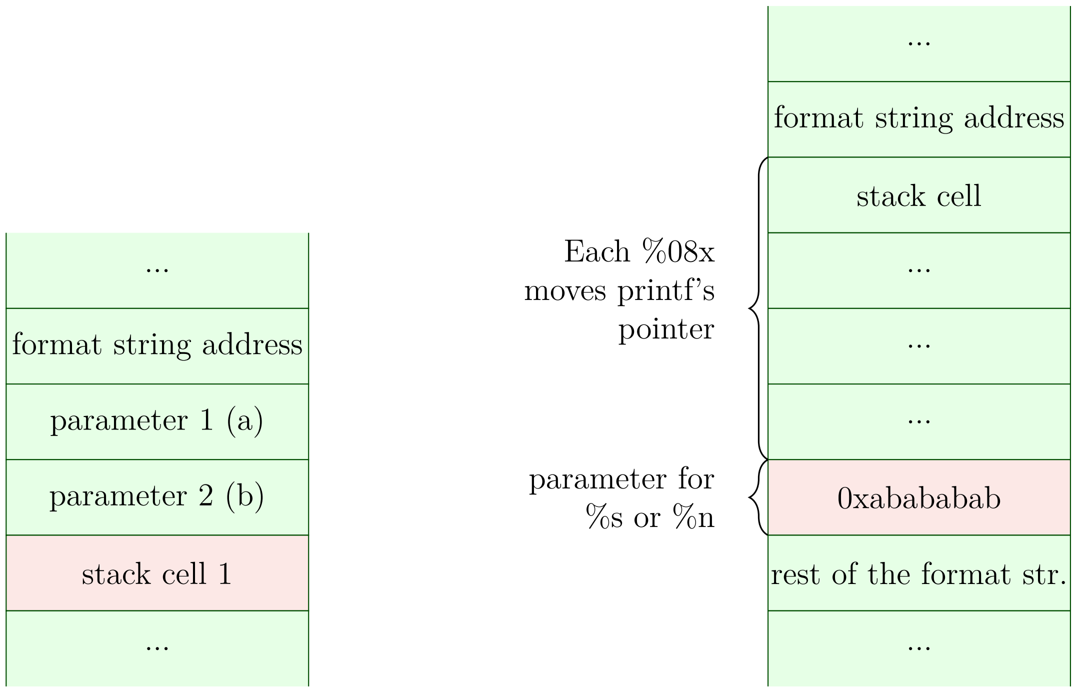
C does not have function overloading for different number of parameters, instead it has variadic functions, for which the number and types of parameters are left unspecified (*Information technology - Programming languages - C* 2011, 6.9.1). Combined with the possibility that format string can be controlled by the attacker opens an attack opportunity (Cowan et al. 2001).

In practice, this means supplying format strings with more formatting directives than supplied parameters. The missing directives use values placed in the stack as their input, which can lead to information disclosure. In addition, `printf`-family of functions has also directive `%n`, which writes the number of characters written so far to the location given as parameter (*Information technology - Programming languages - C* 2011, 7.19.6.1.8). With crafted format string and this directive, the user can write an arbitrary value to a chosen memory location.

Figure 15 illustrates some possible format string attacks. In figure 15a a possible stack layout for `printf`-function is shown. The pointer to format string is stored to the top of the stack, other parameters are stored below it. A `printf`-call corresponding to the layout could be for instance `printf("%d %d", a, b)`, which prints the values of variables `a` and `b`. However, if the attacker can change the format string to, for instance `%d %d %d %d`, `printf` would print in addition the value stored in the stack cell 1, unrelated to the `printf`-function call.

Reading arbitrary memory location can be done by first writing a memory address to the stack, followed by directive `%s`, which will fetch the address of a string from the stack and print it. `printf`'s internal stack counter must be matched with correct number of '%08x', which moves it four bytes towards the top of the stack. For instance, `printf("\xab \xab \xab \xab%08x%08x%x08%08x%s")`, would write an address `0xabababab` to the stack, four `%08x`s move `printf`'s internal pointer to that address, and `%s` prints contents of the address, until NULL byte is encountered (Newsham 2001, chapter 3.3.2). Figure 15b illustrates this.

Similarly, writing to memory can be implemented with directive `%n`. By replacing



(a) Reading stack **(b)** Reading/Writing memory

Figure 15. Examples of stack layouts in format string attacks

the `%s` with `%n` in the same format string as the above example, address `0xabababab` would get written with some small integer value (the exact value depends on the number of bytes output with the `printf` function). Newsham (2001, chapter 3.3.4) discusses this in more detail, e.g. on writing large values.

While the C compilers will typically emit warnings on mismatched between format strings and function parameters, the C runtime offers no protection, if the format string is supplied at the run time.

Integer over- or underflows are another potential source of vulnerabilities in C programs. Integer values are stored in fixed size memory locations. The size of memory allocated for an integer is implementation and platform dependent. Integer type can, for instance, be stored in 32 bits of memory, which means that an unsigned integer value can store values from 0 to $2^{32} - 1$ (`UINT_MAX`), and signed values from -2^{31} to $2^{31} - 1$. If the result of an arithmetic operation cannot fit in this value range, there are several things, that can occur. In the case of unsigned integer, the result is wrapped, for instance the result of addition `UINT_MAX` + 1 would be 0 (*Information technology - Programming languages - C* 2011, chapter 6.2.5.9). In the case of signed integer, the behavior is left unspecified (*Information technology - Programming languages - C* 2011, chapter 6.5.5), which means that the compiler is free to assume that signed overflows will not occur and optimize accordingly as described in e.g. Taylor (2008).

It is possible that integer over/underflows can be used to maliciously control the program logic, but in practice over/underflow bugs are utilized in conjunction with other bugs or memory management to cause temporal or spatial memory access violations. For instance, the program can be tricked to create zero-length memory allocation, which can be used as a building block to more complex exploits (see e.g. Vanegue (2010). Dietz, Li, Regehr, and Adve (2012, section VI) also argue that undefined behavior caused by signed over/underflows represent “time bombs”, as the compilers will use more aggressive optimization strategies, while assuming that undefined behavior will never happen.

Integer over/underflows can mainly be mitigated by detecting them either with software or hardware. Both methods incur some performance penalty and they are not universally used as described in Dietz et al. (2012, V). It is also possible to use static program analysis for finding over/underflows. Even though the problem is generally undecidable, this approach can detect a subset of over/underflow bugs as seen in e.g. Rodrigues, Campos, and Pereira (2013)

As a final bug class *data races* or *race conditions* are discussed. A data race occurs, when two or more concurrent instructions try to write the same memory location at the same time, whereas a race condition will occur, when timing or ordering issues affect a program’s correctness. The use of terms data race and race condition vary in the literature, but this thesis follows usage in Netzer and Miller (1992) However, instead of *general race*, *race condition* is used as above. In some contexts, the term *race condition* is also used for *data races.* Regehr (2011) discusses and defines the terms.

Race conditions and data races are not unique to C programs, but are present in any concurrent programming. C programming language leaves the behavior of data races undefined (*Information technology - Programming languages - C* 2011, chapter 5.1.2.4), which means programs with data races can run in an unpredictable manner. Security bugs due to data races are especially problematic with programs with high concurrency, such as operating system kernels.

Data race detection, similarly to integer over/underflowing is in general an undecidable problem, however, static analysis tooling can detect some race conditions, as described in e.g. Serebryany and Iskhodzhanov (2009).

2.6 Code Reuse Attacks

The bug classes and related exploits surveyed above are all well known, and even if they still occur frequently, they have relatively mature mitigation techniques. Especially the use of no-execute bit makes exploiting ordinary buffer overflows much more difficult than it was when the exploit was first found (Roemer, Buchanan, Shacham, & Savage 2012). However, there is a final class of exploits, which utilize memory safety bugs as a first step, but can bypass the common defenses. These exploits reuse the attacked program's own program code in the attacks, and can generally be called *code reuse attacks* (Bletsch 2011, chapter 2.1). In the context of binary exploitation, the current state-of-the-art code reuse attacks is generally known as *return-oriented programming* (Shacham 2007).

The concept behind the current code reuse attacks was first presented in Peslyak (1997) to counter the non-executable stack memory and was then called *return-to-libc*-attack. The main idea is, instead of writing executable shellcode direction into stack, to fill stack with pointers to suitable parts of program's own code and linked libraries, which will function similarly as direct shellcode would have. Figure 16 illustrates the concept. The original return address of the current function is replaced with address of `system`-function call from `libc`. On top of it is the return address for it and its parameter. When the original function returns, instead of resuming the program flow from the calling point, the subverted program instead calls function `system`, with parameter `/bin/sh`, which spawns a shell, and if the system-function returns, resumes the program flow from attacker supplied return address. More complete details are described in Peslyak (1997).

Return-to-libc -attack has some limitations. First, after calling the first function, the attacker can only call functions, which do not have any parameters. Also, since the parameters are supplied usually by string functions, which cannot transmit NULL-byte. Unfortunately, these limitations can be overcome in certain contexts (Wojtczuk 2001, section 3.1). Mitigations for return-to-libc include stack canary, for detecting stack corruption, address space layout randomization for making the address of libc functions harder to find, and "ASCII armoring" the linked libraries, which means locating them to memory area with a NULL byte in their addresses, which makes it difficult to get their address into stack. There are also other ways of mitigating the attack, and ways of circumventing the mitigations as seen in e.g. Rocha (2016).

The transition to 64-bit architecture also provides another complication for return-

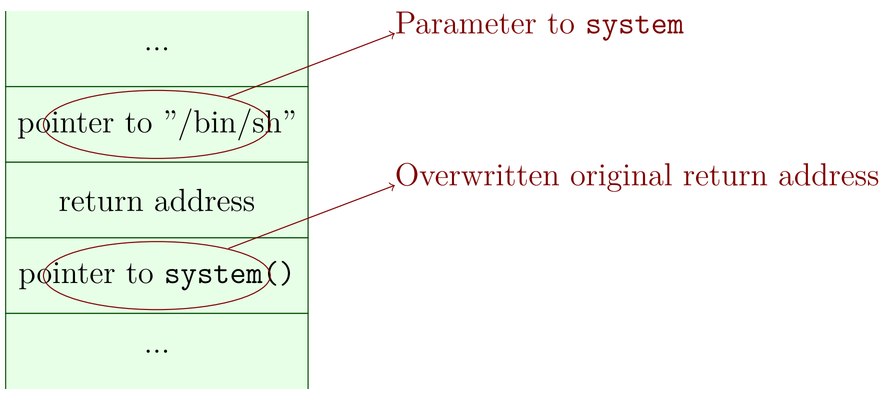


Figure 16. Stack in return-to-libc -attack.

to-libc -attacks. In 32-bit (and earlier) x86-architecture's calling convention dictated that the parameters to functions were passed in stack, where they were easy to corrupt by buffer overruns. However, in 64-bit x86-64 there are more available registers, which are used for passing parameters (Fog 2018, 6,7). To modify a function parameter, the attacker would first need to move it to a correct register. This can be achived by first returning to a code snippet, which performs a suitable `mov`-instruction followed by a `ret`-instruction (cf. Wojtczuk 2001, section 3.2). This insight can be generalized as exploit technique known as *return-oriented programming*, or ROP.

Return-oriented programming was first presented in Shacham (2007) and its feasibility is partly based in the facts, that the Intel's instruction set used in x86 architecture is dense, and that the instructions are encoded in varying amount of bytes. This means that if a random byte sequence is taken, it can be interpreted as a valid byte sequence with a high probability (Shacham 2007, chapter 1.2.1). The main idea of return-oriented programming is to find byte sequences which end in byte value `0xc3` (`ret`-instruction), and the bytes preceding the `ret` can be interpreted as valid assembly instruction sequences. The byte sequences are called *gadgets* and they can be chained to perform arbitrary computations (Roemer et al. 2012; Shacham 2007, 1). A concrete example of ROP-exploitation is given in Reece (2013).

The root cause enabling the ROP-style of exploitation remains in the memory unsafety. Spatial memory violations allow the attacker to subvert the program flow, which is enough to gain complete control of the attacked system. The main idea behind return-oriented programming is the insight that it is not enough to contain malicious *code execution*, but the defenders must also protect against malicious *computation* (Roemer et al. 2012, 1). It is not surprising that return-oriented programming has been generalized into various other techniques, which try to introduce ways to subvert the intended computations the programs are supposed to do. Bletsch (2011, 2) provides a brief overview and an example of similar technique called *jump-oriented programming.* The main insight in jump-oriented programming is to subvert the program flow with indirect jump instructions instead of `ret`.

There are ways of mitigating code-reuse attacks. The first technique, which can be applied is address space layout randomization (ASLR), which can also be used to make ordinary stack overflow type attacks more difficult. In the ROP-case, address space layout randomization makes finding the gadgets and stack addresses more difficult; however, it does not increase the essential complexity of the attack. If the attacker can attempt the attack repeatedly, they are likely to succeed even with address space layout randomization in place. However, as seen above, attacks trying to bypass address space layout randomization can be more easily detected by the defender and can help in building *defense-in-depth.* Instruction set randomization, originally proposed as a counter to code-injection attacks (Gaurav, Keromytis, & Prevelakis 2003) has also been proposed as a counter code-reuse (Bletsch 2011, chapter 2.2), however, to the author's knowledge, this has never been implemented in practice.

A generic and currently most used mitigation against code reuse attacks is control flow integrity (CFI) (Abadi, Budiu, Erlingsson, & Ligatti 2009). It was proposed approximately at the same time as ROP-attacks were popularized, hence the original paper does not consider defenses which specifically target ROP; nevertheless, the techniques still apply.

The main idea in CFI is to ensure, either dynamically at runtime or statically at compile-time, that the program flow does not take paths which are not intended by the programmer. There are various techniques to achieve this, with varying compile or runtime cost. Abadi et al. (2009, section 4.1) proposed instrumenting assembly instructions which transfer the control flow to include checks of source or destination addresses with known ID values.

Another generic approach to mitigate or eliminate ROP attacks is to recompile all the program code in such a way, that the resulting instructions cannot be used (Onarlioglu, Bilge, Lanzi, Balzarotti, & Kirda 2010). This approach is called G-Free. The limitation of this approach is that all the code executed by the program must be compiled with G-Free; modified return instructions have some performance overhead and compiled executables are bigger than the ones compiled without G-Free (Onarlioglu et al. 2010, 7).

There are also many other mitigation techniques, which vary from specific defenses to ROP to more generic CFI type of techniques. At the same time, there are attacks which bypass old defenses as seen in e.g. Onarlioglu et al. (2010, 3), Bletsch (2011, 6), Ionescu (2017) and Pierce (2016).

The rapid changes in code reuse exploitation/mitigation landscape make it difficult to assess which particular exploitation and mitigation techniques will have the most lasting impact and need to be considered more carefully. Academic and commercial authors have incentives to publish new techniques and products whereas exploit authors usually do not want their latest techniques in public, and it remains to be seen whether the different code reuse attacks continue to be a serious threat. There is, however, no shortfall of more complex exploits which combine various techniques presented here, the root enabler of which is either the lack of temporal or spatial memory safety.

3 Memory Safety Issues in Linux Kernel

3.1 Linux Kernel Specific Considerations

In the previous chapter, an overview of common memory safety violations, which can occur in programs written in C, was given. This chapter we discusses how the same violations can occur in Linux kernel.

The Linux kernel is written mostly in C, and partly in Assembly language, which means that it is susceptible to the issues discussed in the previous chapter. However, there are some differences when dealing with vulnerabilities in the operating system kernel, contrasted to user application programs.

The first difference is the absence of C Standard Library in the kernel space. Kernel supplies its own memory management, string handling and other functions. In some cases, the functions provided are similar to their C Standard Library counterparts, as is the case with string handling functions, but in other cases, as with memory management, the functions and interfaces provided are very different (Linux Kernel API 2018).

```
-static void l2cap_add_conf_opt(void **ptr, u8 type, u8 len, unsigned
    long val)
+static void l2cap_add_conf_opt(void **ptr, u8 type, u8 len, unsigned
    long val, size_t size)
 {
        struct l2cap_conf_opt *opt = *ptr;

        BT_DBG("type 0x%2.2x len %u val 0x%lx", type, len, val);

+       if (size < L2CAP_CONF_OPT_SIZE + len)
+               return;
```

Figure 17. Partial patch for CVE-2017-1000251

The second difference is the processor privilege level. The program code in the kernel mode runs with higher privileges than the user mode code, and compromising the kernel can yield bigger rewards for the attacker than attacking user programs.

The third factor is the high concurrency in the kernel. Typical user mode programs do not have high or complex concurrency but the kernel, which controls system multitasking and interaction with the hardware is highly concurrent with complex interactions. In some cases, this can make practical exploitation more difficult, but in other cases, it can open possibilities for new vulnerabilities.

In this chapter, the vulnerability classes introduced above are revisited and examples are given of occurrences in the Linux kernel. Later on, a kernel module containing similar vulnerabilities is implemented. The module is later used to analyze the detection methods of the same vulnerabilities.

3.2 Stack and Buffer Overflows

Linux kernel is vulnerable to ordinary stack corruption as well as other types of buffer overflows. In this section, the stack buffer overflow discussed above is revisited. A separate error condition, stack overflow, which occurs when the stack runs out of space can happen more easily in the kernel context than in user mode programs; and is also discussed in this chapter. Finally, specific mitigations the kernel takes to guard against buffer overflows in some settings are discussed.

An example of a stack buffer overflow due to the lack of bounds checking is CVE-2017-1000251 (2017), which occurred in the Linux Bluetooth subsystem. Vulnerable code would read a configuration packet from the network and store it into stack variable, which could be overflown.

The root cause for the vulnerability is shown in Figure 17, in a patch format, where '-'s indicate removed lines and '+'s code lines added. The full patch can be found in Seri (2017). The reason for error is the lack of bounds checking of input data, which is read from the network, and thus attacker controlled. The impact of the possible exploit can either be a denial of service, where malformed packet can corrupt the kernel stack and crash the system or the attacker can also manipulate the stack to run arbitrary code, in a similar way as demonstrated above.

Linux kernel implements several of the stack buffer overflow mitigations discussed in the previous chapters. Kernel can be configured to use stack canaries but they are not included in every function, for performance reasons. Depending on the kernel configuration options, they can not be used at all, used in functions, which declare a character array of eight bytes or more (`CONFIG_STACKPROTECTOR`), or used in functions using any arrays at all, or use local variables assigned (`CONFIG_STACKPROTECTOR_STR` as seen in Cook (2014), Edge (2014). One should note that the kernel option names have changed during the development process and sources may refer to old option names.

A generic protection against buffer overflows is implemented in functions `copy_to_user` and `copy_from_user` which copy data between user and kernel space. Configuration option `HARDENED_USERCOPY` can be enabled to check for buffer overruns when those functions are used, with a small runtime penalty. The feature was originally developed by PaX team with name of `PAX_USERCOPY` (Edge 2016).

Linux also implements kernel address space layout randomization (KASLR, config option `CONFIG_RANDOMIZE_BASE`), although its implementation is more complex than user space address space layout randomization and benefits are smaller. To be effective, address space layout randomization needs to guard against information leaks about the randomization and that the entropy used must be sufficient. In the case of kernel, there are multiple entities, the location of which must be randomized such as, the actual kernel code, kernel's data structures, modules and other kernel objects (Edge 2013a, 2013b). Currently, only the kernel code, or text-section, and modules are randomized. The entropy for kernel text depends on the processor architecture and memory configuration, and can be between 8 and 30 bits (Biesheuvel 2016). For kernel modules this is currently 10 bits; however, a patch has been proposed to extend it to 17 bits (Edgecombe 2018).

Another class of problems is the information leakage from the kernel. Local users can be assumed to gain knowledge of randomized kernel location and kernel ad-

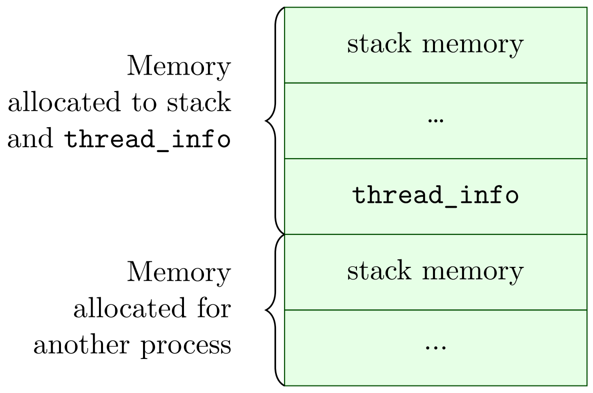


Figure 18. Kernel process stack

dress space layout randomization has been critiqued on these grounds to be ‘cargo cult‘ security (Spender 2013). However, the kernel address space layout randomization is usually considered an effective mitigation against remote attackers (Edge 2013a).

Information leaks can also be indirect. An attack was developed, which broke the confidentiality of kernel address space layout randomization with processor feature called Intel’s Transactional Synchronization Extensions (TSX). TSX allowed a user to probe kernel memory via timing side channel and allowed the attacker to derandomize KASLR in few seconds (Jang et al. 2016). This problem was mitigated by separating the kernel and user mode address spaces (Gruss et al. 2017). As a side note, the mitigation presented was not implemented in Linux kernel to guard against KASLR derandomization, but as a countermeasure to widely published Meltdown processor vulnerability; nevertheless, it effectively helps in the previous problem as well.

Kernel process stacks are relatively small, usually only 8 or 16 kilobytes of size, which means that they can overflow more easily than user mode stacks. The memory area reserved to the process stack contains also `thread_info`-structure, which contains information about the current process, so if the attacker can control the stack content, they also has a known target for modification (Corbet 2016; Horn 2016), as seen in Figure 18.

The figure also shows another possible attack method. The kernel is used to allocate memory reserved for stack allocations in so-called directly mapped memory range, which is a physically contagious memory area. This means that besides clobbering `thread_info` structure, a malicious process can also access the stack area of another process (Corbet 2016). This has been mitigated since Linux

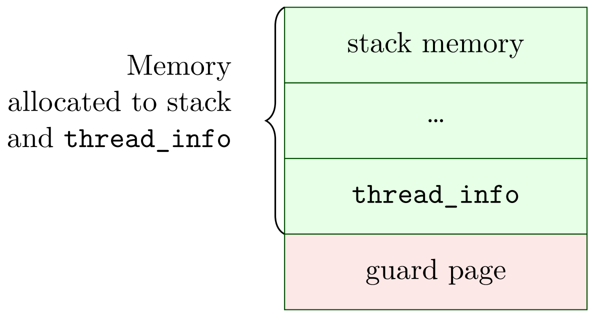


Figure 19. Virtually mapped stack

version 4.9 by moving stack to memory area allocated with `vmalloc` as seen below, where memory is allocated one page at a time. In this case, accessing the memory area outside of the allocated stack will hit the guard page, which will trigger a kernel oops and kill the offending process (Corbet 2016). See also figure 19). This feature is controlled by kernel config option `CONFIG_VMAP_STACK` and is currently available for x86-64 and arm64 architectures. Unfortunately, the option cannot be used together with Kernel Address Sanitizer (KASAN) as discussed below, which can be used to dynamically instrumentate kernel to detect memory violations.

There are also a few other hardenings against stack misuse. Kernel configuration option `CONFIG_THREAD_INFO_IN_TASK`, available since kernel version 4.9, moves the sensitive parts of the `thread_info` structure from the stack to the `task_struct` structure, which is allocated elsewhere. Kernel stack is also susceptible to Gaël Dulalleau's and the Stack Clash attacks as seen above, which leads to bypassing the guard page or memory gap (Corbet 2017a). This can be mitigated with recent STACKLEAK patch, available since 4.20. STACKLEAK also zeroes stack memory after each system call to prevent information leaks (Popov 2018). Linux kernel has also recently removed most of the variable length arrays, which have could have been used to jump over the stack boundaries (Larabel 2018).

There are several examples of bypassing the stack boundaries in the kernel, as seen in e.g. CVE-2010-2240 (2010), where a bug in X.org allowed clashing the heap and stack, and resulted in implementing a guard page between them (Packard 2010). Another well-known example is (CVE-2010-3848 2010), a vulnerability in ECONET network driver documented in Pingios (2010).

3.3 Dynamic Memory Allocation

Dynamic memory allocation in Linux kernel is susceptible to similar problems as the user mode memory allocation, even though the overall memory allocation process is much more complex than in the user mode programs. The most basic way of allocating memory in kernel is to allocate memory by fixed size memory pages, which is in described e.g. in Gorman (2004, chapter 6). When the kernel tries to access the memory outside of the allocated page, the kernel will print a kernel error message, or so called *kernel oops* and kill the process which was running when the illegal memory access happened, or in some cases hung the whole system as seen in Corbet, Rubini, and Kroah-Hartman (2005, 94-98), with more details and also in Duarte (2009) for description of virtual and physical memory omitted here.

Illegal page access can be used to bring the whole system down, causing a denial of service attack. Kernel's dynamic memory allocation can also be attacked similarly as the heap in user mode programs, however, the details are different. Besides the page allocation, kernel provides a mechanism to allocate memory chunks of fixed size, called *slab allocator*. The idea of slab allocator is to allocate large memory area called *slab*, which is partitioned into fixed size memory areas and return pointers to these areas as responses to memory allocation requests. By keeping the size of the allocated memory areas fixed, memory allocation can be kept efficient, since a free memory area can be returned from the slab and no new allocation is needed. If the slab does not provide allocations for multiple objects at once, there is no need to split or expand the slab objects, which does not incur memory fragmentation (Bonwick & Sun Microsystems 1994).

Figure 20 shows a schematic view of Slab allocator memory management. Different slab areas are allocated in memory pages, which contain objects of similar size. The slab allocation is controlled by control or metadata, which is stored outside of slab areas. Each slab also contains some metadata for used and free objects. Allocator implementations may also contain caches and reserved objects for fast-path allocation, separate allocators for physical processors and other optimizations (J. Kim 2012).

The slab allocator was originally developed for SunOS 5.4, however, implemented in other operating systems as well, including Linux. Linux has since gained two other variants of the SLAB allocator (SLAB is usually spelled with capitals when referring the Linux kernel implementation, and in lower case, when referring to

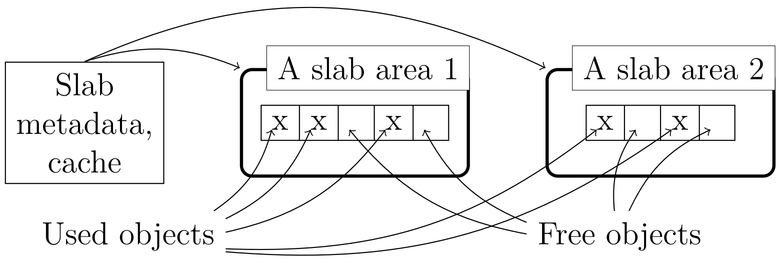


Figure 20. Slab allocator

the general concept), which provide allocators optimized for different situations. The SLOB allocator is optimized for resource constrained devices (Mackall 2005) and the SLUB allocator is a more performant, but also a more complex variant of the original SLAB allocator (Lameter 2007). One difference between the slab implementations is the location of each slab area's metadata. In SLAB, it is stored at the start of the slab area, in SLUB, it is stored in the underlying memory page's control structure and in SLOB's case, the metadata is stored in the slab area itself, much like in user space heap allocator (Argyroudis 2012). There are also other proposed allocators such as SQLB and SLEB, but they are not currently included in the mainline kernel (ibid.) and not considered here.

The most straightforward way the memory unsafety can lead to exploitable bugs is to have one slab object overflow to an adjacent object. In the same way, it is also possible to corrupt adjacent memory page of a slab object, if one is mapped. The third and the most complex way of exploiting the slab allocation is the corruption of the slab metadata, in a similar way as the heap corruption in the user mode programs, but the actual details of the attack are different and depend on the allocator used (Argyroudis 2012). No detailed account is provided of the possible ways to exploit different allocators, which can be found e.g. for SLAB in sgrakkyu and twiz (2007, section 2.2), SLOB in Rosenberg (2016) and SLUB in Perla and Oldani (2011, 160-177).

3.4 Data Races

The third major problem caused by the memory unsafety in the kernel is the problem of data races and race conditions. The problem with data races is much more severe in the kernel than in the user mode programs, since the kernel is by nature much more concurrent than a typical user mode program.

A data race can typically be exploited when the attacker can control two processes, which compete for the same resource, for instance a memory buffer, which has no concurrent access control. Linux kernel has different mechanisms for controlling concurrent access in critical code sections. The most common one is *a semaphore*, which allows only a certain number of readers or writers to access the critical section. When a semaphore is available, or *up*, a process can access the critical section. When the semaphore is unavailable, or *down*, the process trying to acquire the semaphore and access the critical section will sleep until the semaphore is again available. Semaphores can allow for different number of readers and only one writer, but the most common type of scenario is *a mutual exclusion* or *mutex*, which will allow only one process to access the critical section (Corbet et al. 2005, 109-114).

An alternative to semaphore is *a spinlock*. The functionality of a spinlock is similar to a semaphore, however, when failing to acquire the lock to the critical section, the calling process will not go to sleep, but will instead try to acquire the lock again in a tight loop. Spinlocks are used when the calling process cannot go to sleep for some reason (e.g. code in interrupt handlers), they have higher performance, but they are limited to mutual exclusion locks, and they need a pre-emptive kernel scheduling to avoid resource starvation (Corbet et al. 2005, 116-121).

There are two problems with locking. First, a programmer must manually acquire and release locks in the code, since C and other low-level languages do not give guidance to the programmer when they should use locks, and consequently, the locks are easy to miss. A recent example of a missing lock can be seen in CVE-2018-1000004 (2018). Almost the whole patch fixing the race condition is shown in Figure 21, omitting only the declaration of `ioctl_mutex` structure (Iwai 2018).

In this particular case the data race can lead to a deadlock, which is another problem when using locks. A deadlock can occur, when two or more processes compete for the same resources. For instance, if both the processes `P1` and `P2` want to acquire resources `R1` and `R2`, and process `P1` has acquired resource `R1`, and process `P2` the resource `R2`, both the processes will sleep to and wait for the resources the other process has, causing a deadlock. The above bug is caused by the same underlying problem, however, the actual details are more complex and, in practice, easy to miss. E.g. Quan (2018) describes this in further detail.

The complex logic of concurrent kernel programming and the lack of support from the programming language level has been a constant source for data race bugs.

```
        rwlock_init(&client->ports_lock);
        mutex_init(&client->ports_mutex);
        INIT_LIST_HEAD(&client->ports_list_head);
+       mutex_init(&client->ioctl_mutex);

        /* find free slot in the client table */
        spin_lock_irqsave(&clients_lock, flags);
                // ... code omitted ...
                return -EFAULT;
        }

+       mutex_lock(&client->ioctl_mutex);
        err = handler->func(client, &buf);
+       mutex_unlock(&client->ioctl_mutex);
        if (err >= 0) {
                /* Some commands includes a bug in 'dir' field. */
                if (handler->cmd == SNDRV_SEQ_IOCTL_SET_QUEUE_CLIENT
                    ||
```

Figure 21. A patch adding lock for CVE-2018-1000004

Data race bugs of varying severity are further discussed in with descriptions and examples in e.g. CVE-2014-0196 (2014); Groß (2014), CVE-2016-5195 (2016); COW (2016), CVE-2016-8655 (2016); Pettersson (2016) and CVE-2017-2636 (2017); Popov (2017).

There are some mechanisms to help to deal with data races. The kernel contains a lock validator, `lockdep` (Corbet 2016) that can detect some incorrect locking patterns, which can lead to deadlocks. Kernel also provides some alternatives to locking, such as lockless algorithms, atomic variables, `seqlocks` and *read-copy-update*-mechanism (Corbet et al. 2005, 123-130). The is also work to provide the kernel with formal memory-ordering model, which can be used to prove that the kernel works correctly in concurrent situations (Alglave, Maranget, McKenney, Parri, & Stern 2017a, 2017b).

3.5 Other Vulnerability Classes

The memory violations and data races discussed above are the most important vulnerability classes in the Linux kernel. Their exploitation also typically differs from the user mode counterparts. The other vulnerability classes discussed in the previous chapter, namely integer overflows and format string vulnerabilities, are also present in the kernel; however, the kernel context does not make them very different from those occurring in the user mode.

As discussed previously, integer overflows come in two flavors, unsigned and signed. When signed overflow occurs, the result is undefined and can, for example, lead compiler to generate code with security vulnerability, as seen above in Figure 3.

In some cases, the signed integer can be overflown so that instead of using some pre-defined maximum positive value for the variable, overflown variable wraps to a negative value. A recent example of this can be seen in Qualys (2018), where signed integer overflow was used to create a spatial memory violation. CVE-2016-5344 (2016) is an example of unsigned integer overflow in an Android device driver. In this case, the bug allowed to manipulate the memory received from the user, which was erroneously copied to kernel by manipulating the size of the copied memory area.

Format string vulnerabilities in kernel are also similar to those occurring in the user mode, since the kernel print functions implement similar format string modifiers as the standard C library does. An example of kernel format string vulnerability can be found e.g. in CVE-2013-1848 (2013). The Linux kernel does not implement highly vulnerable `%n`-modifier, which allows writing to memory with with an invalid format string, but allows for reading memory. Reading arbitrary kernel memory is usually more dangerous in the kernel context than it is in the user process context, since it can allow both attacking the probabilistic defenses in the kernel and potentially leaking information from other user processes.

4 Detecting and Eliminating Memory Safety Violations and Other Errors

This section surveys the potential approaches to detect and possibly eliminate the problems, which memory unsafety causes. A vulnerable kernel module is implemented and static and dynamic analysis methods, which can be used to either detect or eliminate memory safety issues in the code, are studied. Finally, there is an attempt to reimplement the same kernel module in memory safe language and analyze whether that approach is generally feasible.

4.1 A Faulty Kernel Module

For the purposes of this study, a vulnerable kernel module containing simple examples of the types of programming faults discussed above was implemented. The implementation was done in C language and normal kernel programming tools were used. The module can be tested in any Linux installation, which allows using non-signed third party kernel modules. The module was compiled and tested using Ubuntu 18.10 (GCC version 8.2.0 Ubuntu 8.2.0-7ubuntu1, kernel 4.18.0) and Debian 9 (GCC version 6.3.0 20170516 Debian 6.3.0-18+deb9u1, kernel 4.9.0).

The full source code for the module is available at https://github.com/isido/kernel-module-with-faults and the most important parts are included in the appendices. The source code distribution contains instructions on how to build the module and test it in a virtual machine. The faults in the source code are annotated with text `FAULT: <name>` and they are summarized in Table 1.

Table 1. Programming fault types and their identifiers in the source code

Fault type	**Comment identifier**
Stack buffer overflow	`FAULT: stack buffer overflow`
Slab buffer overflow	`FAULT: heap buffer overflow`
Unsigned integer overflow	`FAULT: unsigned overflow`
Signed integer underflow	`FAULT: signed underflow`
Format string vulnerability	`FAULT: format-string`
Stack area overflow	`FAULT: stack overflow`
Potential data race	`FAULT: race`
Double free	`FAULT: double free`
Use after free	`FAULT: use after free`
Information leak	`FAULT: infoleak`

The faults included in the module are intentionally very simple, since they try to model the essential problem of each bug class and at the same time try to be very simple to understand. The code contains some comments, which aims to clarify how the code works. The code can be interfaced from the user space via `debugfs` endpoints and potentially exploited. In this study, the focus is on the detection and elimination of the selected bug classes, and the author will not detail the possible exploitation and interaction with the module. A small exception to this is the analysis of the code reuse potential of the module, discussed later.

Each of the fault tries to be essentially similar to the already discovered bugs in the Linux kernel, even if in simplified form. A brief explanation of the bugs follow, however, for the full details, the source code is to be studied.

Stack buffer overflow and **Heap buffer overflow** bugs both result from incomplete bounds checking, which allows overwriting other objects in memory. In the case of stack buffer overflow, the return address of the function could be overwritten, if the stack canary value can be bypassed. The heap overflow takes place in the SLAB allocated memory, and can be used to corrupt an adjacent allocated object.

Unsigned integer overflow and **signed integer underflow** both provide a counter whose value can be incremented or decremented by reading the provided debugfs endpoint. When the counter value over- or underflows, the kernel ring buffer will display a message about executing a function, which should not be reachable under the normal execution. Signed integer underflow is also undefined behavior according to the C standard, which means that the compiled code might have some surprising properties as well.

The **format string vulnerability** simply copies verbatim a user-supplied string to `printk`-function. It can be noted that the newer interface for ring-buffer-logging, the `pr_` family of functions is not as vulnerable to format string problems, nevertheless, the older `printk`-function is used to demonstrate the problem, even if the use of `printk` is not the best current practice in the kernel programming.

Stack area overflow is simply a static buffer allocated from the process stack, whose size might exceed the kernel process stack area. The allocation takes place in data race write endpoint and is not separately available.

Potential data race endpoint copies a value from the user to two different buffers, separated by a small (1,000-microsecond) delay. It can be triggered by having two separate processes writing simultaneously different values to the endpoint. When data race happens, kernel ring buffer will display a message.

Double free and **Use after free** bugs provide endpoints that trigger buggy behavior. With double free, reading from the endpoint makes an allocation, and writing to it triggers deallocation. The endpoint does not provide good input for the debugs-operations but kernel ring buffer will provide the feedback. Use after free is simply triggered by reading from the endpoint.

Finally, **information leak** simply returns the contents of uninitialized memory area when the endpoint is read.

4.2 Static analysis

First, it was analyzed whether different static analysis tools would detect the coding faults we implemented. There is a great variety of static analysis tooling and it differs in sophistication, from very simple syntax checkers to formal proof systems. The baseline for the static analysis tooling was the compiler. At the second level, the author tried various simple static analysis tools, which could easily be included in a standard kernel development workflow, which require little or no configuration. Finally, we tested briefly two more sophisticated static analysis tools, one of which was an academic project, and another a tool developed in the industry. Both of the tools did not require any alterations to the original source code.

The author did not test any sophisticated formal methods based tools in this study. One reason for this the was limited amount of time. The tools usually require considerable familiarity with some formal proof system and might also require annotations in the source code for the properties or invariants which they are trying to prove correct and just testing a single tool would have expanded the scope of this thesis considerably. Another factor, which can be derived from the previous one, is that since the tooling is nontrivial to use, it is not very likely to be used by the kernel developers.

The baseline for the tests was the GNU Compiler Collection's C-compiler, **GCC** (GCC 2019). GCC is currently the only C compiler that can be used to build Linux kernel for all platforms, even though there is a work in process to enable building the kernel with other compilers as well (Larabel 2018). GCC has many heuristics, which can be used to emit warnings for problems in the program code. In this study, all warnings were enable to provide a baseline for the experiment. The only fault detected was stack area overflow, which generated a warning in compilation phase. GCC version 6 used with Debian also generated another warning about unused result from function in potential data race fault; however, the warning did not point to the actual problem in the function.

Basic static analyzers form the first class of tested tools. The static analyzers try to analyze the source code at compile time and warn about suspicious code patterns or buggy code. There are static analyzers of varying complexity and sophistication. The simpler tools try to find code patterns, which are known to be dangerous, whereas more sophisticated analyzers form a formal model of the code execution and try to prove some properties of the model.

Two other tools used are originally developed specially for the Linux kernel. The first, **sparse** is called a “semantic parser” (Sparse 2019), which means that it tries to infer the meaning of code, instead of just its syntax. Sparse was originally developed by Linus Torvalds, and it is used for instance to notify about incorrect memory handling between kernel and user space (Brown 2016c). It can be used to find bugs from data races and to annotate the code so that certain bugs are easier to find, cf. Brown (2016a).

The main functionality of Sparse is in the form of a C-library that can be used to build more tools. The tool for kernel static analysis is also called ’Sparse’ and it is this latter Sparse, which is studied in this work. **Smatch** or Source Matcher is a tool, built on top of Sparse to provide more comprehensive static analysis capabilities than original Sparse. As is case with Sparse, the Smatch tool can be extended to detect more unsafe or erroneous code patterns manually, but in this study, Smatch was used as provided by the upstream. (Brown 2016b; Smatch 2019).

Clang C compiler (Clang 2019), which is built on top of the LLVM Compiler Infrastructure (LLVM 2019), is a mature C compiler, which can be also used to build Linux kernel on some platforms, even though on some platforms, the kernel relies on GCC-specific extensions. Clang compiler includes also a static analyzer for C and C++ code, called **scan-build**, which can be used to test the faulty kernel module.

Finally, the author evaluated a stand-alone static analyzer called **Cppcheck** (Marjamäki 2019). Cppcheck is a general static analyser for C and C++ programs, not specifically aimed to work with Linux kernel, yet, it has been successfully used to find bugs in it (Cppcheck 2019).

The results for the basic static analysis tools, alongside with the GCC baseline are summarized in Table 2. The versions used in the test are given in Table 3.

The analyzers tested here did find just few of the faults implanted in the module. Sparse noticed an information leak resulting from the use of uninitialized memory. Smatch noticed a use after free error and potential information leak related to it, but not the implanted information leak. Otherwise, the tools did not detect the errors. Sparse, however, was helpful in separating pointers to kernel and user space, when correct annotations were used correctly. Sparse contains also other annotations beside those used in this test, and it seems that using them can prevent some of the memory unsafety issues.

On the usability perspective, all of the tools were relatively easy to use with kernel. The tools required just a normal installation procedure and they could be run directly against the source code, with no special requirements with project setup.

One of the reasons for the lack of findings, or *completeteness* can be in the nature of the implanted bugs. Buggy code works correctly as long as the user input or interaction is withing certain limits. To actually notice buggy behavior requires either generating counterexamples or proving the code correct, which is not something that can be usually done with basic static analysis, which is based on analyzing code patterns and semantic behavior of the code.

Table 2. Fault detection by basic static analysis

Fault type	**GCC**	**sparse**	**Smatch**	**Clang**	**Cppcheck**
Stack buffer overflow	-	-	-	-	-
Slab buffer overflow	-	-	-	-	-
Unsigned integer overflow	-	-	-	-	-
Signed integer underflow	-	-	-	-	-
Format string vulnerability	-	-	-	-	-
Stack area overflow	yes	-	-	-	-
Potential data race	-	-	-	-	-
Double free	-	-	-	-	-
Use after free	-	-	yes	-	-
Information leak	-	yes	yes	-	-

Table 3. Tool versions used for static analysis

Tool	**Version**
GCC	8.2.0 (Ubuntu 8.2.0-7ubuntu1)
Sparse	0.5.2 (Ubuntu: 0.5.2-1)
Smatch	git commit 10ef4ac
Clang	scan-build 7.0-43ubuntu1
Cppcheck	1.84
Frama-C	Sulfur-20171101i
Infer	v0.15.0

There is much research on more sophisticated static analyzers, however, in general, the more advanced tools are not easy to integrate in the development process; they might require some specific expertise to run, and might require modifications to the source code. As a result, the use of more formal static analyzers tend to be limited to academic settings rather than ordinary kernel development process. Some of the better known tools include **Coverity**, **Dr. CHECKER**, **Frama-C** and **Infer** (CEA LIST & Inria 2019; Facebook 2019; Synopsys 2019a; The Computer Security Group at UC Santa Barbara 2019).

Coverity is possibly the best known and the most mature of more advanced static analyzers. It was originally developed in Stanford University but has since been morphed into a commercial project. Coverity has been used to find defects also in the Linux Kernel (Corbet 2014). Coverity is not freely available; however, it provides scans of Linux kernel as a free service (Synopsys 2019b). Coverity was not tested in this study, since it is not freely available.

Dr. CHECKER is a relatively new static analyzer aimed to specially find bugs in Linux kernel drivers. It uses *soundy* analysis methodology, which means that its results are almost, but not completely free of false positives, or *sound*. It has been successfully used to find bugs in the Linux kernel (Machiry et al. 2017). Unfortunately, when evaluating the tool, the author could not configure it to process only a single module, within a reasonable time; hence it could not be included the results of this study.

Frama-C is a modular framework for analyzing C programs. In practice, this means that Frama-C is a collection of different analyzers, which can work together and provide a single analysis. It has been widely used in different contexts, and has also been used to analyze Linux kernel code (Khoroshilov & Mandrykin 2017).

Infer is a static analyzer developed by Facebook. Unlike the other analyzers tested, besides C/C++ code, it can also analyze Java and Objective-C code. It is not specifically aimed at Linux kernel, and to the author's knowledge, it has not been reported to find bugs in the kernel.

There has been some work in applying formal methods to prove the correctness of parts of Linux kernel; however, but this has been mainly an academic endeavor and not yet widely used. Since the scope of this work was limited to tools, which could be used relatively easily in the normal kernel development workflow, they were not included in this study. Some examples of these approaches, however, can be found e.g. in (Efremov, Mandrykin, & Khoroshilov 2018; Penninckx, Jacobs, Smans, & Muhlberg 2019).

Neither of the advanced static analysis tools tested, Frama-C and Infer detected any of the planted faults. There may be many reasons for this. It is possible, that testing Linux kernel module would have required a more complex setup than the basic tool documentation described, and the tools themselves might have required some special settings. The output from the tools, however, did not indicate any problems in running the analysis.

Especially in the case of Frama-C, it quickly became apparent that the tool is not intended to be an out-of-the-box static analyzer, but its effective would require modelling the wanted properties of the source code and then implementing the logic to analyze these properties within the Frama-C framework.

Based on the testing, it seems that the more advanced static analyzers would require a significant investment in time to become effective in finding the kinds of bugs present in the test case. It is also possible that the bugs implanted are just too difficult for the tools to find. If this is the case, static analysis in its current form is not a complete solution for detection and elimination of memory safety bugs. On the other hand, it can be noted that Sparse and Smatch, probably the best-known tools in the Linux kernel development community, were able to find some of the bugs, which can be seen as a good thing. Since the two tools are well integrated in the kernel coding process, they are also likely to be used during normal kernel development.

4.3 Dynamic Analysis

Dynamic analysis means running the code under analysis, and trying to observe its behavior. When compared to static analysis, dynamic analysis is usually more complex and possibly more time consuming than static analysis, which can take place at the same time as code compilation. Dynamic analysis usually requires setting up special runtime environment, so that the tested code can be observed. Dynamic analysis also requires some knowledge of the properties of the program under test. For instance, dynamic analysis might need a set of inputs and outputs, which can be used to perform the analysis. There is also a specialized subset of dynamic analysis called *fuzzing*, where given input sets are modified or mutated randomly or according to a specific patterns and fed to the program so that they might trigger unusual code paths and expose errors.

One of the best-known dynamic analysis tools is **Valgrind**. It runs a program in a special execution environment, where each memory access is instrumented. Instrumentation can reveal memory handling errors when program is run, although the program's performance is slowed approximately 20 to 30 times the program's normal speed (Seward 2019).

Valgrind as such cannot be used to analyze Linux kernel components, such as the author's vulnerable module; however, Linux kernel currently has similar functionality called **Kernel Address Sanitizer** or **KASAN**. KASAN depends on GCC compiler to create similar instrumentation to Valgrind, which detects temporal and spatial memory violations and reports them to the kernel ring buffer log, instead of crashing the kernel. KASAN is intended as a debugging aid and cannot be used in the production kernels due to performance penalty it incurs. To use KASAN, kernel must be specially compiled using configuration option `CONFIG_KASAN` and then the part of the kernel user test must be somehow invoked.

KASAN is very well suited to test the vulnerable module developed in this thesis project, since all the functionality of the module is exposed via debugfs-endpoints and testing the module can simply be done by giving the module legal and illegal inputs and monitoring the kernel ring buffer log.

Linux kernel also has similar functionality to detect undefined behavior with compiler instrumentation, called the **Undefined Behavior Sanitizer** or **UBSAN**. UBSAN works by inserting checks for code that might exhibit potential undefined behavior at compile time, and invoking the checks at runtime. Similarly to KASAN, UBSAN can also be turned on with kernel configuration option `CONFIG_UBSAN`.

Finally, there is a kernel fuzzing tool called **Syzkaller**, which has been used to find bugs in Linux kernel and later also in other operating system kernels (Google 2019). Syzkaller works by enabling kernel instrumentation for kernel code coverage and initial C programs using Linux system calls. Based on the coverage information, Syzkaller mutates the programs to find a kernel crash. When a crash is found, it tries to minimize the mutated C program to as short form as possible (Konovalov 2019).

In this work, both KASAN and UBSAN were tested against the vulnerable test module. In this experiment, Syzkaller was excluded due to the fact that each of the exposed debugfs endpoints only implements simple read-write functionality and actual system calls are not really used. In addition, creating crashes with the exposed endpoints is already trivial, and generating in some sense minimal inputs which reproduce the crash is not very helpful in our case, when the main interest is in locating the invalid code.

In the tests, UBSAN did not detect any undefined behavior. The potential undefined behavior we had in the test code was signed integer underflow, which was not detected. KASAN, however, easily noticed the problems related to illegal memory accesses. The findings are detailed in Table 4. KASAN error reports are somewhat helpful for finding the problematic code, since they detail the function called when

Table 4. Fault detection by dynamic analysis

Fault type	**KASAN**
Stack buffer overflow	yes
Slab buffer overflow	yes
Unsigned integer overflow	-
Signed integer underflow	-
Format string vulnerability	-
Stack area overflow	-
Potential data race	-
Double free	yes
Use after free	yes
Information leak	-

the error occurred and contents of the problematic memory. Figure 22 gives an abridged log of KASAN error message. The first part gives general information of the error type, the second part contains the call trace of the offending kernel process and the third part is a memory dump of the related memory.

KASAN seems to be a good tool for finding spatial and perhaps temporal memory safety violations, at least in the current simple test case. In the general setting, triggering memory problems when code is more complex would require finding execution and input patterns that trigger the faults, so it is unclear how well the current result can be generalized. It is possible that in general more complex approaches for testing should be employed, for instance using Syzkaller or using some sort of fuzzing framework for input manipulation.

KASAN is also not a comprehensive solution for all the bug classes enabled by the lack of safety in C programming language; it detected only memory errors. However, KASAN was the tool which detected more errors than any other tool tested and its use was relatively straight-forward, so it can be considered to be a good, however not necessarily sufficient solution for writing safe kernel code.

4.4 Memory-Safe Languages

Unsafe C or C++ are not the only languages operating system kernels could be written. There has long been talk of using some memory safe language. Despite the talk, there have been very few attempts to use other languages in kernel setting. Perhaps the best-known effort has been the use of OCaml in special purpose Mirage *unikernels* (MirageOS 2019). Unikernel is not a full operating system,

```
[ 8251.457761]
   ==================================================================
[ 8251.458557] BUG: KASAN: stack-out-of-bounds in
   simple_write_to_buffer+0x58/0xd0 at addr ffff8800165d7d48
[ 8251.459499] Write of size 13 by task bash/1665
[ 8251.459982] page:ffffea00005975c0 count:0 mapcount:0 mapping:
            (null) index:0x0
[ 8251.460783] flags: 0x1ffff8000000000()
[ 8251.461151] page dumped because: kasan: bad access detected
[ 8251.461806] CPU: 1 PID: 1665 Comm: bash Tainted: G           O
   4.9.144 #2
...
[ 8251.461860] Call Trace:
[ 8251.461877]  [<ffffffff81ee79cc>] ? dump_stack+0xaf/0x103
[ 8251.461890]  [<ffffffff81ee791d>] ? _atomic_dec_and_lock+0x9d/0x9d
[ 8251.461903]  [<ffffffff81bb5a36>] ? __dump_page+0x176/0x460
[ 8251.461913]  [<ffffffff81c21495>] ? kasan_report.part.0+0x5a5/0
   x7b0
[ 8251.461925]  [<ffffffff81c20203>] ? save_stack+0x33/0xa0
[ 8251.461935]  [<ffffffff81c20203>] ? save_stack+0x33/0xa0
[ 8251.461946]  [<ffffffff81cb41a8>] ? simple_write_to_buffer+0x58/0
   xd0
[ 8251.461956]  [<ffffffff81c20203>] ? save_stack+0x33/0xa0
[ 8251.461967]  [<ffffffff81cb41a8>] ? simple_write_to_buffer+0x58/0
   xd0
[ 8251.461977]  [<ffffffff81c9ef40>] ? __fdget+0x10/0x10
[ 8251.461988]  [<ffffffff81d16750>] ? locks_remove_posix+0xb0/0x2d0
[ 8251.461998]  [<ffffffff81cb41a8>] ? simple_write_to_buffer+0x58/0
   xd0
[ 8251.462019]  [<ffffffffc06703fe>] ? sbo_write+0x6e/0xa0 [faulty]
[ 8251.462033]  [<ffffffffc0670390>] ? sbo_read+0x20/0x20 [faulty]
...
[ 8251.462175] Memory state around the buggy address:
[ 8251.462677]  ffff8800165d7c00: 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
[ 8251.463393]  ffff8800165d7c80: 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
[ 8251.464108] >ffff8800165d7d00: 00 00 00 00 00 f1 f1 f1 f1 00 02 f4
    f4 00 00 00
[ 8251.464914]                                                   ^
[ 8251.465494]  ffff8800165d7d80: 00 00 00 00 f1 f1 f1 f1 04 f4 f4 f4
    00 00 00 00
[ 8251.466208]  ffff8800165d7e00: 00 00 00 00 00 00 00 00 00 00 00 00
    00 00 00 00
[ 8251.466937]
   ==================================================================
```

Figure 22. A Partial KASAN output for stack buffer overflow

but rather a specialized program, usually dedicated to a single task, that can be run in hypervisor like a virtual machine. Unikernels do not also do task switching, but run in a single address space for the whole time. Madhavapeddy and Scott (2014) provide an overview of the Unikernel concept.

Unikernels can be written in high-level languages more easily than complete operating systems, since they do not need to abstract away the hardware layer, schedule tasks and other operating system kernel tasks. Implementing a full operating system or its part also requires some specific properties from the language. The language must be able to directly access memory via pointers or similar abstraction. The language must also be able to control its memory management and it must be able to function without system calls provided by the operating system.

These conditions rule out many of the common memory safe languages, even those oriented to systems programming. For instance, to use Google's Go-language in operating system kernel, one would have to reimplement Go's runtime system which provides Go's system call interface and memory management. Go's current memory management also relies on garbage collection, which might provide unsuitable in operating systems context.

There are, however, memory safe languages suitable to use in the operating system context. **D**-language is a systems-level language evolved from C and C++, which has also been used in operating systems context (Dlang 2019). Another, not very well known candidate is **ATS**. ATS is a functional programming language, inspired by ML. ATS has strong type system and can use linear types.

Linear types are types the values of which can be used exactly once during the program. Having this type constraint helps compiler to generate very efficient code (Xi 2019). Unlike D, ATS is still largely an academic project, and not generally widely used, despite its impressive features.

Finally, a third candidate suitable for operating system kernels is **Rust**, initially developed by Mozilla (Rust Developers 2019). Rust has been very well received and it has been used in some projects to rewrite code originally written in C or C++, as seen e.g. in Mozilla Developer Network (2019), Remacs developers (2019). Rust contains ideas from functional languages and strong type system. Rust's types are not linear as with ATS, but affine, which means that the type value can be used **at most** one time. This allows efficient memory management without garbage collection. Rust also forbids multiple writes to a single resource; each resource can only be altered by its owner. The ownership system can eliminate data races, however, at the cost of steeper learning curve than other popular languages.

In this work, Rust was chosen to see if the author can port the buggy kernel module written in C to a memory safe language, analyze the challenges in porting and see whether the memory unsafe errors can be eliminated in the porting process.

As a starting point, a bare bones Linux kernel module written in Rust (T. Kim 2017) was used. The kernel module works by generating Rust bindings, to Linux kernel functions and structures defined in C header files. The kernel can be interfaced using generated Rust counterparts to kernel's C functions and structures. The full source code and build instructions are available online at https://github.com/isido/faulty-rust.ko.

Rust version used was 1.35.0-nightly (f69422288 2019-04-01). Stable versions cannot be used to build the module, since some gated features are used.

When porting the C code to Rust, two problems became apparent. First, Rust's binding generator cannot generate Rust bindings to full C header files. For instance, preprocessor macros defined in the header files are not generated into Rust equivalents, which means that Linux kernel constants must be redefined in the Rust code, which leads to code duplication and is a source of potential errors. Likewise, helper functions defined as C macros are not ported and must be redefined, if they are to be used in the Rust code. In addition, functions defined as `static inline`, are not exported to Rust code and they must also be redefined in Rust. A Figure 23 shows an example of redefinition boiler-plate code. The problem of unnecessary redefinitions may be lessened in the future if the program responsible for the bindings, Rust-Bindgen gains more features.

This problem is generally more serious than was apparent in the testing, since Rust-Bindgen cannot in all cases generate binary compatible bindings to all kernel structures, whose structure may depend in used compiler options and optimizations used. This problem did not manifest itself in the testing, since the interfacing to kernel was done only via well-defined function calls.

Another serious problem is that kernel functions and direct pointers to memory, or *raw pointers* in Rust parlance, must be used inside `unsafe`-code blocks. Unsafe blocks in Rust code are parts of the code, where Rust's safety features are turned off, in order to do something that is normally not permitted by the Rust's type system. On one hand, this permits Rust to interact directly with the kernel; on the other hand, unsafe code is not able to benefit from the Rust's safety guarantees. In

```
const ___GFP_IO: gfp_t = 0x40;
const ___GFP_FS: gfp_t = 0x80;
const ___GFP_ZERO: gfp_t = 0x8000;
const ___GFP_DIRECT_RECLAIM: gfp_t = 0x200000;
const ___GFP_KSWAPD_RECLAIM: gfp_t = 0x400000;

const __GFP_RECLAIM: gfp_t = ___GFP_DIRECT_RECLAIM |
    ___GFP_KSWAPD_RECLAIM;
const __GFP_IO: gfp_t = ___GFP_IO;
const __GFP_FS: gfp_t = ___GFP_FS;
const __GFP_ZERO: gfp_t = ___GFP_ZERO;
const GFP_KERNEL: gfp_t = __GFP_RECLAIM | __GFP_IO | __GFP_FS;

const PAGE_SIZE: usize = 4096;

unsafe fn kzalloc(size: usize, flags: gfp_t) -> *mut c_void {
    __kmalloc(size, flags | __GFP_ZERO)
}

unsafe fn kmalloc(size: usize, flags: gfp_t) -> *mut c_void {
    __kmalloc(size, flags)
}
```

Figure 23. Redefinitions of C constants and functions in Rust

our case, unsafe code blocks allowed us to port the faulty kernel module written in C to Rust. Unfortunately, the porting process brought also all the same memory unsafety bugs to our Rust code, which were present in the C code.

Figure 24 shows an example of ported stack buffer overflow in Rust. A positive thing is, that in this particular case the unsafe code is isolated in a relatively small section of unsafe code, here consisting of just one function call. However, in many places in the Rust code, the author was forced to use much larger unsafe code blocks. This may, however, been an artifact of the used setting. The code in kernel modules is typically much more complicated than in our experiment, and the unsafe code blocks may usually be much smaller. Unsafe blocks can also help in localizing possible dangerous or erroneous code segments; however, in the test setting, no evidence was gained for this.

Rust code helped to spot some errors, which went otherwise undetected. Rust compiler noticed integer over- and underflows, although the code was compiled in debug-mode. If the code is compiled in release-mode, it does not have checks detecting these. The compiler also noticed some cases of use of the uninitialized memory, which were not intentional and forced the author to use unsafe code blocks for potentially hazardous code. Table 5 summarizes how the faults were translated into Rust.

```
pub fn rust_stack_write(
    _fps: *mut file,
    buf: *const c_char,
    len: usize,
    offset: *mut loff_t,
) -> isize {
    const KBUF_SIZE: usize = 10;
    let flag = 0; // variable to clobber
    let mut kbuf = [0; KBUF_SIZE];
    let bytes_written;

    unsafe {
        bytes_written = std::os::kernel::simple_write_to_buffer(
            kbuf.as_mut_ptr() as *mut c_void,
            len,
            offset,
            buf as *const c_void,
            len,
        );
    }

    bytes_written
}
```

Figure 24. Stack buffer overflow in Rust

The reason for unsafety was either unsafe function call or unsafe memory access. Both could be mitigated with some effort. Unsafe memory access could be mitigated by implementing safe memory management and safe data structures. Normal Rust standard library contains both of these but when used in the kernel settings, the normal standard library cannot be used; instead Rust provides a small core library, which provides only basic pointer types and only stack based memory management. Nothing would prevent a kernel programmer from implementing for instance slab based memory management, although it was deemed to be outside of the scope of this work. The same this true for unsafe C functions, which could be reimplemented in safe Rust. This has been indeed how Rust has been used in some porting efforts. In Linux kernel setting, this seems, however, very unlikely. It would mean that the kernel developers would change their whole infrastructure and learn a new language to use in kernel programming and since the kernel development process seems to work in its current form, it is difficult to see what would drive a change of this magnitude.

Indeed, this social issue of C as traditionally used kernel programming language and the economy of existing, tested codebase may be the biggest hurdles to the widespread use of Rust in the kernel context. The other problems mentioned are of technical nature and they tend to be more easily solved than social or economic problems. It may be that the future successes of Rust and other memory safe languages come from new, greenfield projects.

Finally, it must also be noted that the current Rust implementation may not be optimally safe Rust code, and it would have been possible to eliminate some of the ported errors without using unsafe code. Rust language is also evolving in a rapid pace, and may provide more facilities in the form of code and documentation for reducing the use of unsafe code.

Table 5. Faults prevented in Rust port of C module and reasons for unsafety

Fault type	**Prevented by Rust/Reason for unsafety**
Stack buffer overflow	unsafe function access
Slab buffer overflow	unsafe function access
Unsigned integer overflow	yes (Only in debug-mode)
Signed integer underflow	yes (Only in debug-mode)
Format string vulnerability	unsafe function access
Stack area overflow	unsafe memory access
Potential data race	unsafe memory access
Double free	unsafe function access
Use after free	unsafe function access
Information leak	unsafe memory access

5 Conclusions

This study surveyed various issues caused by memory unsafe programming and their possible solutions. Even though the issues caused by the lack of memory protection have long history and are well known, they still pose a serious threat to the security of the computer systems.

The long list of mitigations surveyed have one thing in common. All the mitigations can be circumvented, if the attacker is clever and persistent enough. The exploitation of the vulnerabilities is considerably more difficult in the modern operating systems than it was twenty years ago, when the first stack based exploits were popularized, nevertheless, it is still possible. This study showed that unfortunately, there is no silver bullet for eliminating the problems caused by the memory unsafety. The static analysis tools are either not powerful enough or easy enough to use. The tools for dynamic analysis gave better results in the testing; however, it is unclear if the dynamic analysis can find deep bugs in contrast of shallow ones available in the test setting. Dynamic analysis also requires a separate testing phase in addition to development phase in contrast with static analysis, which can be easily included in the development process. This can hinder the use of dynamic analysis in practice. Both analysis types have their use, yet, even if their usage will increase, it is likely that they cannot significantly decrease the amount of memory safety bugs.

Rewriting code in memory safe languages, such as, Rust, has often been brought up as a way to eliminate memory safety problems. This may be the case when these languages are used to write new code in new projects; however, this study showed that there are serious challenges when trying to use them in old, established settings, such as, Linux kernel. It is our hope, however, that by identifying the challenges in using new tools in old settings can help to make some progress in overcoming those challenges. Memory safe languages have one advantage compared to static and dynamic code analysis techniques: they can provide guarantees that certain bug classes can be eliminated altogether, or at least confined to unsafe sections, whereas code analysis can only guarantee finding *some* of the bugs present in the code. At the moment, it seems however, that the memory safety problems stay with us for some time.

References

Abadi, M., Budiu, M., Erlingsson, Ú., & Ligatti, J. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, *13*(1), 1–40. doi:10.1145/1609956.1609960

Alglave, J., Maranget, L., McKenney, P. E., Parri, A., & Stern, A. 2017a. A formal kernel memory-ordering model (part 1). *Linux Weekly News*. Accessed on 5 January 2019. Retrieved from https://lwn.net/Articles/718628/

Alglave, J., Maranget, L., McKenney, P. E., Parri, A., & Stern, A. 2017b. A formal kernel memory-ordering model (part 2). *Linux Weekly News*. Accessed on 5 January 2019. Retrieved from https://lwn.net/Articles/720550/

Andrews, J. 2008. Security bugs and full disclosure. *Kernel Trap*. Accessed on 15 October 2017. Retrieved from https://web.archive.org/web/20080719130436/http://kerneltrap.org/Linux/Security_Bugs_and_Full_Disclosure

Anonymous. 2001. Once upon a free(). *Phrack Magazine*, *11*(57). Accessed on 18 February 2018. Retrieved from http://phrack.org/issues/57/9.html

AppArmor Authors. 2017. *AppArmor*. Accessed on 1 November 2017. Retrieved from http://wiki.apparmor.net/index.php/Main_Page

Argyroudis, P. 2012. *The Linux kernel memory allocators from an exploitation perspective* [Argp]. Accessed on 17 December 2018. Retrieved from https://argp.github.io/2012/01/03/linux-kernel-heap-exploitation/

Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. 2014. *Operating systems: Three easy pieces*. Arpaci-Dusseau Books Wisconsin. Accessed on 7 February 2018. Retrieved from http://pages.cs.wisc.edu/~remzi/OSTEP/

Biesheuvel, A. 2016. *KASLR in the arm64 Linux kernel* [Work of ard]. Accessed on 14 November 2018. Retrieved from http://www.workofard.com/2016/05/kaslr-in-the-arm64-kernel/

Bletsch, T. 2011. *Code-reuse attacks: New frontiers and defenses* (Doctoral dissertation, North Carolina State University).

Bonwick, J., & Sun Microsystems. 1994. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer* (pp. 87–98). Accessed on 16 December 2018. Retrieved from https://www.usenix.org/legacy/publications/library/proceedings/bos94/full_papers/bonwick.ps

Brown, N. 2016a. Better types in C using sparse and smatch. *Linux Weekly News*. Accessed on 29 March 2019. Retrieved from https://lwn.net/Articles/696624/

Brown, N. 2016b. Smatch: Pluggable static analysis for C. Accessed on 29 March 2019. Retrieved from https://lwn.net/Articles/691882/

Brown, N. 2016c, June 8. Sparse: A look under the hood. *Linux Weekly News*. Accessed on 29 March 2019. Retrieved from https://lwn.net/Articles/689907/

Burow, N., Zhang, X., & Payer, M. [Mathias]. 2018. Shining light on shadow stacks. *arXiv:1811.03165 [cs]*. arXiv: 1811.03165. Accessed on 13 November 2018. Retrieved from http://arxiv.org/abs/1811.03165

CEA LIST, & Inria. 2019. *Frama-C*. Accessed on 29 March 2019. Retrieved from https://frama-c.com/

Chisnall, D. 2010. *What is Mac OS X?* Accessed on 6 August 2017. Retrieved from http://www.informit.com/articles/article.aspx?p=1552774

Clang. 2019. *Clang: A C language family frontend for LLVM*. Accessed on 30 March 2019. Retrieved from https://clang.llvm.org/

Cook, K. 2014. *-fstack-protector-strong* [Codeblog]. Accessed on 11 November 2018. Retrieved from https://outflux.net/blog/archives/2014/01/27/fstack-protector-strong/

Cook, K. 2015. Kernel Self Protection Project. *Linux Weekly News*. Accessed on 14 October 2017. Retrieved from https://lwn.net/Articles/663361/

Corbet, J. 2009. Fun with NULL pointers, part 1. *Linux Weekly News*. Accessed on 6 June 2016. Retrieved from https://lwn.net/Articles/342330/

Corbet, J. 2015. Kernel security: Beyond bug fixing [LWN.net]. *Linux Weekly News*. Accessed on 14 October 2017. Retrieved from https://lwn.net/Articles/662219/

Corbet, J. 2014. One year of Coverity work. *Linux Weekly News*. Accessed on 20 April 2019. Retrieved from https://lwn.net/Articles/608992/

Corbet, J. 2017a. Preventing stack guard-page hopping. *Linux Weekly News*. Accessed on 27 November 2018. Retrieved from https://lwn.net/Articles/725832/

Corbet, J. 2017b. Unsafe_put_user() turns out to be unsafe. Accessed on 15 October 2017. Retrieved from https://lwn.net/Articles/736348/

Corbet, J. 2016. Virtually mapped kernel stacks. *Linux Weekly News*. Accessed on 20 November 2018. Retrieved from https://lwn.net/Articles/692208/

Corbet, J., Rubini, A., & Kroah-Hartman, G. 2005. *Linux device drivers: Where the kernel meets the hardware*. " O'Reilly Media, Inc."

COW, D. 2016. Dirty COW. Accessed on 5 January 2019. Retrieved from https://dirtycow.ninja/

Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., & Lokier, J. 2001. FormatGuard: Automatic protection from printf format string vulnerabilities. In *USENIX security symposium* (Vol. 91, p. 10).

Cowan, C., Wagle, P., Pu, C., Beattie, S., & Walpole, J. 1999. *Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade.*

Cppcheck. 2019. *Cppcheck wiki: Found bugs.* Accessed on 20 April 2019. Retrieved from https://web.archive.org/web/20131014013121/http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=Found_bugs#Linux_kernel

CVE-2009-1897. 2009. CVE-2009-1897. Accessed on 6 June 2016. Retrieved from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1897

CVE-2010-2240. 2010. CVE-2010-2240. Accessed on 28 November 2018. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2010-2240

CVE-2010-3848. 2010. CVE-2010-3848. Accessed on 28 November 2018. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2010-3848

CVE-2013-1848. 2013. CVE-2013-1848. Accessed on 20 January 2019. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2013-1848

CVE-2014-0196. 2014. CVE-2014-0196. Accessed on 5 January 2019. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2014-0196

CVE-2016-5195. 2016. CVE-2016-5195. Accessed on 5 January 2019. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2016-5195

CVE-2016-5344. 2016. CVE-2016-5344. Accessed on 20 January 2019. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2016-5344

CVE-2016-8655. 2016. CVE-2016-8655. Accessed on 5 January 2019. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2016-8655

CVE-2017-1000251. 2017. CVE-2017-1000251. Accessed on 2 November 2018. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2017-1000251

CVE-2017-2636. 2017. CVE-cve-2017-2636. Accessed on 5 January 2019. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-CVE-2017-2636

CVE-2018-0825. 2018. CVE-2018-0825. Accessed on 26 March 2018. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2018-0825

CVE-2018-1000004. 2018. CVE-2018-1000004. Accessed on 16 January 2018. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2018-1000004

CVE-2018-6789. 2018. CVE-2018-6789. Accessed on 26 March 2018. Retrieved from https://nvd.nist.gov/vuln/detail/CVE-2018-6789

CWE-415. 2019. CWE-415. Accessed on 1 May 2019. Retrieved from https://cwe.mitre.org/data/definitions/415.html

CWE-416. 2019. CWE-416. Accessed on 1 May 2019. Retrieved from https://cwe.mitre.org/data/definitions/416.html

de Raadt, T. 2004. *Exploit mitigation techniques.* Accessed on 28 October 2018. Retrieved from http://www.openbsd.org/papers/ven05-deraadt/index.html

Delalleau, G. 2005. *Large memory management vulnerabilities.* Accessed on 27 November 2018. Retrieved from https://cansecwest.com/core05/memory_vulns_delalleau.pdf

Dietz, W., Li, P., Regehr, J., & Adve, V. 2012. Understanding integer overflow in C/C++. In *Proceedings of the 34th international conference on software engineering (ICSE), Zurich, Switzerland* (p. 11). Zurich, Switzerland: IEEE Press. Accessed on 19 July 2018. Retrieved from http://www.cs.utah.edu/~regehr/papers/overflow12.pdf

Dlang. 2019. *Areas of D usage.* Accessed on 24 April 2019. Retrieved from https://dlang.org/areas-of-d-usage.html

Duarte, G. 2009. *How The Kernel Manages Your Memory.* Accessed on 12 December 2018. Retrieved from https://manybutfinite.com/post/how-the-kernel-manages-your-memory/

Eckert, M., Bianchi, A., Wang, R., Shoshitaishvili, Y., Kruegel, C., & Vigna, G. 2018. HeapHopper: Bringing bounded model checking to heap implementation security. In *Proceedings of the 27th USENIX security symposium* (p. 19). 27th USENIX security symposium. Baltimore, MD, USA.

Edge, J. 2014. "Strong" stack protection for GCC. *Linux Weekly News.* Accessed on 26 September 2018. Retrieved from https://lwn.net/Articles/584225/

Edge, J. 2016. Hardened usercopy. *Linux Weekly News.* Accessed on 28 November 2018. Retrieved from https://lwn.net/Articles/695991/

Edge, J. 2013a. Kernel address space layout randomization. *Linux Weekly News.* Accessed on 11 November 2018. Retrieved from https://lwn.net/Articles/569635/

Edge, J. 2013b. Randomizing the kernel [Linux weekly news]. *Linux Weekly News.* Accessed on 12 November 2018. Retrieved from https://lwn.net/Articles/546686/

Edge, J. 2009. The future for grsecurity. *Linux Weekly News.* Accessed on 25 October 2017. Retrieved from https://lwn.net/Articles/313621/

Edgecombe, R. 2018. *Kernel-hardening - [PATCH v9 0/4] KASLR feature to randomize each loadable module.* Accessed on 14 November 2018. Retrieved from https://www.openwall.com/lists/kernel-hardening/2018/11/10/1

Efremov, D., Mandrykin, M., & Khoroshilov, A. 2018. Deductive verification of unmodified Linux kernel library functions. In T. Margaria & B. Steffen (Eds.), *Leveraging applications of formal methods, verification and validation. verification* (pp. 216–234). Lecture Notes in Computer Science. Springer International Publishing.

Evans, C. 2017. *Ode to the use-after-free: One vulnerable function, a thousand possibilities.* Accessed on 8 May 2017. Retrieved from https://scarybeastsecurity.blogspot.com/2017/05/ode-to-use-after-free-one-vulnerable.html

Facebook. 2019. *Infer*. Accessed on 29 March 2019. Retrieved from https://fbinfer.com/

Fog, A. 2018. *Calling conventions*. Accessed on Retrieved from https://www.agner.org/optimize/calling_conventions.pdf

Gaurav, S. K., Keromytis, D., Angelos, & Prevelakis, V. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on computer and communications security* (p. 10). ACM.

GCC. 2019. *GCC, the gnu compiler collection*. Accessed on 29 March 2019. Retrieved from https://gcc.gnu.org/

Google. 2019. *Syzkaller*. Accessed on 21 April 2019. Retrieved from https://github.com/google/syzkaller

Gorman, M. 2004. *Understanding the Linux Virtual Memory Manager*. Bruce Perens' Open Source series. Upper Saddle River, NJ: Prentice Hall.

Groß, S. 2014. *Exploiting CVE-2014-0196 a walk-through of the Linux pty race condition PoC* [Include security]. Accessed on 5 January 2019. Retrieved from http://blog.includesecurity.com/2014/06/exploit-walkthrough-cve-2014-0196-pty-kernel-race-condition.html

Grsecurity. 2017. *Grsecurity.net*. Accessed on 15 October 2017. Retrieved from https://www.grsecurity.net/

Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., & Mangard, S. 2017. KASLR is dead: Long live KASLR. In E. Bodden, M. Payer, & E. Athanasopoulos (Eds.), *Engineering secure software and systems* (Vol. 10379, pp. 161–176). doi:10.1007/978-3-319-62105-0_11

Hall, J., Méndez, R. C., & Lich, B. 2017, April 5. *Mitigate threats by using Windows 10 security features*. Accessed on 23 July 2017. Retrieved from https://docs.microsoft.com/en-us/windows/threat-protection/overview-of-threat-mitigations-in-windows-10

Horn, J. 2016. *Project Zero: Exploiting Recursion in the Linux Kernel*. Accessed on 20 November 2018. Retrieved from https://googleprojectzero.blogspot.com/2016/06/exploiting-recursion-in-linux-kernel_20.html

Howard, M. 2006. *Address space layout randomization in Windows Vista* [Michael howard's web log]. Accessed on 23 July 2017. Retrieved from https://blogs.msdn.microsoft.com/michael_howard/2006/05/26/address-space-layout-randomization-in-windows-vista/

Howard, M., & Lipner, S. 2003. Inside the Windows security push. *IEEE Security Privacy*, *1*(1), 57–61. doi:10.1109/MSECP.2003.1176996

Ionescu, A. 2017. *Universally bypassing CFG through mutability abuse*. EuskalHack 2017, Donostia. Accessed on 15 September 2018. Retrieved from http://alex-ionescu.com/publications/euskalhack/euskalhack2017-cfg.pdf

Information technology - Programming languages - C. 2011. International Organization for Standardization. Geneva, Switzerland.

Iwai, T. 2018. *Kernel/git/torvalds/linux.git - Linux kernel source tree*. Accessed on 5 January 2019. Retrieved from https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b3defb791b26ea0683a93a4f49c77ec45ec96f10

Jang, Y., Lee, S., & Kim, T. 2016. Breaking kernel address space layout randomization with Intel TSX. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security - CCS'16* (pp. 380–392). The 2016 ACM SIGSAC conference. doi:10.1145/2976749.2978321

Jones, D. 2013. *Trinity: A Linux kernel fuzz tester*.

Kaempf, M. 2001. Vudo - an object superstitiously believed to embody magical powers. *Phrack Magazine*, *11*(57). Accessed on 7 June 2018. Retrieved from http://phrack.org/issues/57/8.html#article

Kernighan, B. W., & Ritchie, D. M. 1978. *The C Programming Language*. Prentice Hall.

Khoroshilov, A., & Mandrykin, M. 2017. *Specifying and proving correctness of Linux kernel components with ACSL*. Accessed on 20 April 2019. Retrieved from http://linuxtesting.org/downloads/20170530-ispras-astraver.pdf

Kim, J. 2012. *How does the SLUB allocator work*. Accessed on 27 December 2018. Retrieved from https://events.static.linuxfound.org/images/stories/pdf/klf2012_kim.pdf

Kim, T. 2017. *Rust.ko*. Accessed on 24 April 2019. Retrieved from https://github.com/tsgates/rust.ko

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S., NICTA, UNSW, Open Kernel Labs, & ANU. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on operating systems principles* (pp. 207–220). ACM. Accessed on 6 August 2017. Retrieved from http://dl.acm.org/citation.cfm?id=1629596

Klein, T. 2009. *RELRO - A (not so well known) Memory Corruption Mitigation Technique*. Accessed on 9 January 2019. Retrieved from http://tk-blog.blogspot.com/2009/02/relro-not-so-well-known-memory.html

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., & Yarom, Y. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

Konovalov, A. 2019. *Coverage-guided USB fuzzing with Syzkaller*. OffensiveCon 2019. Accessed on 22 April 2019. Retrieved from https://docs.google.com/presentation/d/1z-giB9kom17Lk21YEjmceiNUVYeI6yIaG5_gZ3vKC-M

Kuperman, B., Brodley, C., Ozdoganoglu, H., Vijaykumar, T., & Jalote, A. 2005. Detection and prevention of stack buffer overflow attacks. *Communications of the ACM*, *48*(11), 50–56. Accessed on 26 September 2018. Retrieved from https://courses.cs.washington.edu/courses/csep590/05au/readings/p50-kuperman.pdf

Lameter, C. 2007. SLUB: The unqueued slab allocator v6. *Linux Weekly News*. Accessed on 16 December 2018. Retrieved from https://lwn.net/Articles/229096/

Larabel, M. 2018. The Linux Kernel Is Now VLA-Free: A Win For Security, Less Overhead & Better For Clang. *Phoronix*. Accessed on 28 November 2018. Retrieved from https://phoronix.com/scan.php?page=news_item&px=Linux-Kills-The-VLA

Lattner, C. 2011a. *LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #1/3*. Accessed on 17 May 2016. Retrieved from http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html

Lattner, C. 2011b. *LLVM Project Blog: What Every C Programmer Should Know About Undefined Behavior #2/3*. Accessed on 6 June 2016. Retrieved from http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html

Lea, D., & Gloger, W. 1996. *A memory allocator*. Accessed on 9 May 2018. Retrieved from http://gee.cs.oswego.edu/dl/html/malloc.html

Levy, E. 1996. Smashing the stack for fun and profit. *Phrack Magazine*, *7*(49).

Linux Information Project. 2017. *Kernel definition*. Accessed on 10 October 2017. Retrieved from http://www.linfo.org/kernel.html

Linux Kernel API. 2018. *The Linux Kernel API*. Accessed on 6 November 2018. Retrieved from https://www.kernel.org/doc/htmldocs/kernel-api/

Linux Kernel Developers. 2017. *The kernel address sanitizer (KASAN) — the Linux kernel documentation*. Accessed on 1 November 2017. Retrieved from https://www.kernel.org/doc/html/latest/dev-tools/kasan.html

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., & Hamburg, M. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

LLVM. 2019. *The llvm compiler infrastructure*. Accessed on 30 March 2019. Retrieved from https://www.llvm.org/

Löhner, C. 2017. *Funny statistics for the Linux kernel - lines of code, bad words, good words - the Linux counter project - statistics about Linux, its users and more*. Accessed on 10 October 2017. Retrieved from https://www.linuxcounter.net/

Loosemore, S., McGrath, R., Oram, A., & Stallman, R. M. 2001. *The GNU C library reference manual.* Free software foundation Boston.

Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., & Vigna, G. 2017. DR. CHECKER: A soundy analysis for linux kernel drivers. In *Proceedings of the 26th USENIX security symposioum*, Vancouver, BC, Canada. Accessed on Retrieved from https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-machiry.pdf

Mackall, M. 2005. Slob: Introduce the SLOB allocator. *Linux Weekly News.* Accessed on 16 December 2018. Retrieved from https://lwn.net/Articles/157944/

Madhavapeddy, A., & Scott, D. J. 2014, January 1. Unikernels: The rise of the virtual library operating system. *Communications of the ACM*, *57*(1), 61–69. doi:10.1145/2541883.2541895

Marco-Gisbert, H., & Ripoll, I. 2014. *On the effectiveness of full-ASLR on 64-bit Linux.* Accessed on Retrieved from http://cybersecurity.upv.es/attacks/offset2lib/offset2lib-paper.pdf

Marjamäki, D. 2019. *Cppcheck: A tool for static C/C++ code analysis.* Accessed on 30 March 2019. Retrieved from http://cppcheck.sourceforge.net/

Meer, H. et al. 2010. Memory corruption attacks the (almost) complete history. *Blackhat USA.(Jul. 2010).* Accessed on 4 March 2018. Retrieved from https://media.blackhat.com/bh-us-10/whitepapers/Meer/BlackHat-USA-2010-Meer-History-of-Memory-Corruption-Attacks-wp.pdf

Miller, M. 2009. *Preventing the exploitation of user mode heap corruption vulnerabilities – security research & defense* [Microsoft tech net]. Accessed on 28 October 2018. Retrieved from https://blogs.technet.microsoft.com/srd/2009/08/04/preventing-the-exploitation-of-user-mode-heap-corruption-vulnerabilities/

Miller, M. 2013. *Software defense: Mitigating heap corruption vulnerabilities – security research & defense* [Microsoft tech net]. Accessed on 28 October 2018. Retrieved from https://blogs.technet.microsoft.com/srd/2013/10/29/software-defense-mitigating-heap-corruption-vulnerabilities/

MirageOS. 2019. *MirageOS: A programming framework for building type-safe, modular systems.* Accessed on 24 April 2019. Retrieved from https://mirage.io/

Moreira, J., Rigo, S., Polychronakis, M., & Kemerlis, V. P. 2017. *DROP THE ROP fine-grained control-flow integrity for the Linux kernel.*

Mozilla Developer Network. 2019. *Building Firefox with Rust code.* Accessed on 1 May 2019. Retrieved from https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Building_Firefox_with_Rust_code

Netzer, R. H. B., & Miller, B. P. 1992. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst. 1*(1), 74–88. doi:10.1145/130616.130623

Newsham, T. 2001. *Exploiting format string vulnerabilities.* Accessed on 19 June 2018. Retrieved from https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., & Kirda, E. 2010. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th annual computer security applications conference on - ACSAC '10* (p. 49). The 26th annual computer security applications conference. doi:10.1145/1920261.1920269

OpenBSD Developers. 2017. *Openbsd.* Accessed on 6 August 2017. Retrieved from https://www.openbsd.org/

Packard, K. 2010. *Kernel/git/torvalds/linux.git - Linux kernel source tree.* Accessed on 28 November 2018. Retrieved from https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=320b2b8de12698082609ebbc1a1716572

PaX Team. 2001. *Address space layout randomization.* Accessed on 8 October 2018. Retrieved from https://pax.grsecurity.net/docs/aslr.txt

Penninckx, W., Jacobs, B., Smans, J., & Muhlberg, J. T. 2019. *Verification of Linux kernel modules: Experience report.* Accessed on Retrieved from https://www.researchgate.net/publication/267206117_Verification_of_Linux_Kernel_Modules_Experience_Report

Perla, E., & Oldani, M. 2011. *A guide to kernel exploitation: Attacking the core.* Amsterdam; Boston: Syngress.

Peslyak, A. 1997. *Getting around non-executable stack (and fix).* Accessed on 3 August 2018. Retrieved from http://seclists.org/bugtraq/1997/Aug/63

Peslyak, A. 2000. *Jpeg com marker processing vulnerability.* Accessed on 18 February 2018. Retrieved from http://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability

Pettersson, P. 2016. *CVE-2016-8655 Linux af_packet.c race condition (local root)* [Oss-sec]. Accessed on 5 January 2019. Retrieved from https://seclists.org/oss-sec/2016/q4/607

Phantasmal Phantasmagoria. 2005. *The malloc maleficarum.* bugtraq@securityfocus.com. Accessed on 18 February 2018. Retrieved from https://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt

Pierce, C. 2016. *ROP is dying and your exploit mitigations are on life support* [Endgame]. Accessed on 15 September 2018. Retrieved from https://www.endgame.com/blog/technical-blog/rop-dying-and-your-exploit-mitigations-are-life-support

Pingios, A. 2010. *CVE-2010-3848: Linux kernel econet_sendmsg() Stack Overflow.* Accessed on 28 November 2018. Retrieved from https://xorl.wordpress.com/2010/12/01/cve-2010-3848-linux-kernel-econet_sendmsg-stack-overflow/

Popov, A. 2017. *CVE-2017-2636: Exploit the race condition in the n_hdlc linux kernel driver bypassing SMEP*. Accessed on 5 January 2019. Retrieved from https://a13xp0p0v.github.io/2017/03/24/CVE-2017-2636.html

Popov, A. 2018. *How STACKLEAK improves Linux kernel security*. Accessed on 27 November 2018. Retrieved from http://blog.ptsecurity.com/2018/10/how-stackleak-improves-linux-kernel.html

Qualys. 2018. *Oss-security - integer overflow in Linux's create_elf_tables() (CVE-2018-14634)*. Accessed on 20 January 2019. Retrieved from https://www.openwall.com/lists/oss-security/2018/09/25/4

Qualys. 2017a. *Qualys Security Advisory*. Accessed on 19 June 2017. Retrieved from https://www.qualys.com/2017/06/19/stack-clash/stack-clash.txt

Qualys. 2017b. *The Stack Clash*. Accessed on 19 June 2017. Retrieved from https://blog.qualys.com/securitylabs/2017/06/19/the-stack-clash

Quan, L. 2018. *Oss-sec: Sound driver conditional competition*. Accessed on 5 January 2019. Retrieved from https://seclists.org/oss-sec/2018/q1/51

Reece, A. 2013. *Introduction to return oriented programming (ROP)* [Code arcana]. Accessed on 9 September 2018. Retrieved from http://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html

Regehr, J. 2011. *Race condition vs. data race* [Embedded in academia]. Accessed on 27 July 2018. Retrieved from https://blog.regehr.org/archives/490

Remacs developers. 2019. *Remacs: A community-driven port of emacs to rust*. Accessed on 1 May 2019. Retrieved from https://github.com/remacs/remacs

Reshetova, E., Liljestrand, H., Paverd, A., & Asokan, N. 2017. Towards Linux kernel memory safety. *arXiv:1710.06175 [cs]*. arXiv: 1710.06175. Accessed on 29 October 2017. Retrieved from http://arxiv.org/abs/1710.06175

Ritchie, D. M. 1993. The development of the C language. *SIGPLAN Not. 28*(3), 201–208. doi:10.1145/155360.155580

Ritchie, D. M., & Thompson, K. 1974. The UNIX time-sharing system. *Commun. ACM*, *17*(7), 365–375. doi:10.1145/361011.361061

Rocha, L. 2016. *Evolution of stack based buffer overflows* [Count upon security]. Accessed on 29 August 2018. Retrieved from https://countuponsecurity.com/tag/return-to-libc/

Rodrigues, R. E., Campos, V. H. S., & Pereira, F. M. Q. 2013. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 IEEE/ACM international symposium on code generation and optimization (CGO)* (pp. 1–11). Proceedings of the 2013 IEEE/ACM international symposium on code generation and optimization (CGO). doi:10.1109/CGO.2013.6494996

Roemer, R., Buchanan, E., Shacham, H., & Savage, S. 2012. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security*, *15*(1), 1–34. doi:10.1145/2133375.2133377

Rosenberg, D. 2016. *A heap of trouble: Breaking the Linux kernel SLOB allocator.* Accessed on 18 December 2018. Retrieved from https://www.vsecurity.com/download/papers/slob-exploitation.pdf

Rust Developers. 2019. *Rust programming language.* Accessed on 21 April 2019. Retrieved from https://www.rust-lang.org/

Serebryany, K., & Iskhodzhanov, T. 2009. ThreadSanitizer: Data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications* (pp. 62–71). WBIA '09. doi:10.1145/1791194.1791203

Seri, B. 2017. *Kernel/git/torvalds/linux.git - Linux kernel source tree.* Accessed on 7 November 2018. Retrieved from https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e860d2c904d1a9f38a24eb44c9f34b8f915a6ea

Seward, J. 2019. Valgrind. Accessed on 21 April 2019. Retrieved from http://www.valgrind.org/

sgrakkyu, & twiz. 2007. Attacking the Core: Kernel Exploiting Notes. *Phrack Magazine*, (64). Accessed on 29 November 2018. Retrieved from http://phrack.org/issues/64/6.html

Shacham, H. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on computer and communications security* (pp. 552–561). CCS '07. doi:10.1145/1315245.1315313

Smalley, S., Vance, C., & Salamon, W. 2001. Implementing SELinux as a Linux security module. *NAI Labs Report*, *1*(43), 139.

Smatch. 2019. *Smatch.* Accessed on 29 March 2019. Retrieved from http://smatch.sourceforge.net/

Spafford, E. H. 1989. The internet worm program: An analysis. *ACM SIGCOMM Computer Communication Review*, *19*(1), 17–57.

Sparse. 2019. *Sparse.* Accessed on 29 March 2019. Retrieved from https://sparse.wiki.kernel.org/index.php/Main_Page

Spender, B. 2013. *KASLR: An exercise in cargo cult security* [Grsecurity forums]. Accessed on 15 November 2018. Retrieved from https://forums.grsecurity.net/viewtopic.php?f=7&t=3367

Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., & Walter, T. 2009. Breaking the memory secrecy assumption. In *Proceedings of the second european workshop on system security - EUROSEC '09* (pp. 1–8). The second european workshop. doi:10.1145/1519144.1519145

Synopsys. 2019a. *Coverity.* Accessed on 29 March 2019. Retrieved from https://scan.coverity.com/

Synopsys. 2019b. *Coverity Linux scan.* Accessed on 20 April 2019. Retrieved from https://scan.coverity.com/projects/linux

Szekeres, L., Payer, M. [M.], Wei, T., & Song, D. 2013. SoK: Eternal war in memory. In *2013 IEEE symposium on security and privacy* (pp. 48–62). 2013 IEEE symposium on security and privacy. doi:10.1109/SP.2013.13

Taylor, I. L. 2008. *Airs – signed overflow.* Accessed on 27 July 2018. Retrieved from https://www.airs.com/blog/archives/120

The Computer Security Group at UC Santa Barbara. 2019. *Dr. CHECKER: A soundy vulnerability detection tool for linux kernel drivers.* Accessed on 20 April 2019. Retrieved from http://cppcheck.sourceforge.net/

Timberg, C. 2015. The kernel of the argument. *The Washington Post.* Accessed on 8 May 2017. Retrieved from http://www.washingtonpost.com/sf/business/2015/11/05/net-of-insecurity-the-kernel-of-the-argument

van de Ven, A. 2004. *New security enhancements in Red Hat Enterprise Linux v.3, update 3.* Red Hat. Accessed on 28 October 2018. Retrieved from https://web.archive.org/web/20050512030425/http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf

Vanegue, J. 2010. Zero-sized heap allocations vulnerability analysis. In *WOOT.* WOOT'10 proceedings of the 4th USENIX conference on offensive technologies, Washington, DC: USENIX.

Vermeulen, S. 2011. *High level explanation on some binary executable security.* Accessed on 9 January 2019. Retrieved from http://blog.siphos.be/2011/07/high-level-explanation-on-some-binary-executable-security/

Wang, X., Chen, H., Cheung, A., Jia, Z., Zeldovich, N., & Kaashoek, M. F. 2012. Undefined behavior: What happened to my code? In *Proceedings of the asia-pacific workshop on systems* (p. 9). ACM. Accessed on 17 May 2016. Retrieved from http://dl.acm.org/citation.cfm?id=2349905

Wang, X., Zeldovich, N., Kaashoek, M. F., & Solar-Lezama, A. 2013. Towards optimization-safe systems: Analyzing the impact of undefined behavior. (pp. 260–275). doi:10.1145/2517349.2522728

Wojtczuk, R. 2001. The advanced return-into-lib(c) exploits: Pax case study. *Phrack Magazine*, *11*(58). Accessed on 29 August 2018. Retrieved from http://phrack.org/issues/58/4.html

Xi, H. 2019. *The ATS programming language.* Accessed on 24 April 2019. Retrieved from http://www.ats-lang.org/

Xie, T., Zhang, Y., Li, J., Liu, H., & Gu, D. 2016. New exploit methods against ptmalloc of GLIBC. In *2016 IEEE trustcom/BigDataSE/ISPA* (pp. 646–653). 2016 IEEE trustcom/BigDataSE/ISPA. doi:10.1109/TrustCom.2016.0121

Appendices

Appendix 1: Makefile source code

```
ifneq ($(KERNELRELEASE),)
obj-m += faulty.o
faulty-y := faulty_main.o

ccflags-y := -DDEBUG -Wall

else

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

endif
```

Appendix 2: faulty.c source code

```
/*
 * Faulty: A kernel module with intentional (and unintentional?) bugs
 */

#include <linux/debugfs.h>
#include <linux/delay.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/uaccess.h>
#include <linux/slab.h>

#define BUF_SIZE 256

static struct dentry *dir;
static const char *root = "ffaulty";

static int init_endpoint(struct dentry *dir, const char *fn,
                        const struct file_operations *fops);
static ssize_t sbo_read(struct file *fps, __user char *buf, size_t len,
                        loff_t *offset);
static ssize_t sbo_write(struct file *fps, const char __user *buf,
                        size_t len, loff_t *offset);
static ssize_t slab_read(struct file *fps, char __user *buf, size_t len,
                        loff_t *offset);
static ssize_t slab_write(struct file *fps, const char __user *buf,
                        size_t len, loff_t *offset);
static void slab_operate_with_other_data(void);
static ssize_t unsigned_overflow_read(struct file *fps, char __user *buf,
                                size_t len, loff_t *offset);
static ssize_t signed_underflow_read(struct file *fps, char __user *buf,
                                size_t len, loff_t *offset);
static ssize_t format_read(struct file *fps, char __user *buf, size_t len,
                        loff_t *offset);
static ssize_t format_write(struct file *fps, const char __user *buf,
                        size_t len, loff_t *offset);
static ssize_t race_read(struct file *fps, char __user *buf, size_t len,
                        loff_t *offset);
static ssize_t race_write(struct file *fps, const char __user *buf,
                        size_t len, loff_t *offset);
static ssize_t df_alloc(struct file *fps, char __user *buf, size_t len,
                        loff_t *offset);
static ssize_t df_free(struct file *fps, const char __user *buf,
                size_t len, loff_t *offset);
static ssize_t use_after_free_read(struct file *fps, char __user *buf,
                                size_t len, loff_t *offset);
static ssize_t infoleak_read(struct file *fps, char __user *buf,
                        size_t len, loff_t *offset);
static void non_reachable_function(void);

// stack buffer overflow
static char *buffer = "Write more than 10 bytes here to cause "
        "stack buffer overflow.\n";
```

```
static const struct file_operations fops_sbo = {
        .owner = THIS_MODULE,
        .open = simple_open,
        .read = sbo_read,
        .write = sbo_write,
};

// slab corruption
static const struct file_operations fops_slab = {
        .owner = THIS_MODULE,
        .open = simple_open,
        .read = slab_read,
        .write = slab_write,
};

struct some_data {
        char data[10];
        bool flag_which_is_never_set;
};

static struct some_data *user_controlled;
static struct some_data *other_data;
static bool toggle;

// under/overflow
static u8 unsigned_counter = 250;
static s8 signed_counter = -124;

static const struct file_operations fops_overflow = {
        .owner = THIS_MODULE,
        .open = simple_open,
        .read = unsigned_overflow_read,
};

static const struct file_operations fops_underflow = {
        .owner = THIS_MODULE,
        .open = simple_open,
        .read = signed_underflow_read,
};

// format string bug
static char *some_string = "A write to this endpoint will get copied "
        "to kernel message buffer\n";

static const struct file_operations fops_format = {
        .owner = THIS_MODULE,
        .open = simple_open,
        .read = format_read,
        .write = format_write,
};

// data race
static char *race1;
static char *race2;

static const struct file_operations fops_race = {
        .owner = THIS_MODULE,
        .open = simple_open,
        .read = race_read,
        .write = race_write,
};

// double free
static char *double_free;

static const struct file_operations fops_double_free = {
        .owner = THIS_MODULE,
        .open = simple_open,
        .read = df_alloc,
        .write = df_free,
};

// use after free
static const struct file_operations fops_use_after_free = {
        .owner = THIS_MODULE,
        .open = simple_open,
        .read = use_after_free_read,
};

static const struct file_operations fops_infoleak = {
        .owner = THIS_MODULE,
        .open = simple_open,
        .read = infoleak_read,
};

// FAULT: infoleak
#define DATA_LEN 4096
struct a_struct {
        char data[DATA_LEN];
} *uninitialized;

static int __init mod_init(void)
{
        pr_debug("Faulty: creating debugfs-endpoints\n");

        dir = debugfs_create_dir(root, NULL);

        if (dir == ERR_PTR(-ENODEV)) {
                pr_err
                    ("Faulty: Debugfs doesn't seem to be compiled "
                            "into the kernel\n");
                return -ENODEV;
        }

        if (dir == NULL) {
                pr_err
                    ("Faulty: Cannot create debugfs-entry '%s'", root);
                return -ENOENT;
        }

        if (!init_endpoint(dir, "sbo", &fops_sbo))
                pr_debug
```

```
                    ("Faulty: Stack buffer overflow at debugfs '%s/sbo'\n",
                           root);

        if (!init_endpoint(dir, "slab", &fops_slab))
                pr_debug("Faulty: Slab buffer overflow at debugfs '%s/slab'\n",
                        root);

        if (!init_endpoint(dir, "overflow", &fops_overflow))
                pr_debug("Faulty: Unsigned integer overflow at debugfs "
                        "'%s/overflow'\n", root);

        if (!init_endpoint(dir, "underflow", &fops_underflow))
                pr_debug("Faulty: Signed integer underflow at debugfs "
                        "'%s/underflow'\n", root);

        if (!init_endpoint(dir, "format", &fops_format))
                pr_debug("Faulty: Format string bug at debugfs "
                        "'%s/format'\n", root);

        if (!init_endpoint(dir, "data-race", &fops_race)) {
                race1 = kzalloc(PAGE_SIZE, GFP_KERNEL);
                race2 = kzalloc(PAGE_SIZE, GFP_KERNEL);
                pr_debug("Faulty: Data race at debugfs "
                        "'%s/data-race'\n", root);
        }

        if (!init_endpoint(dir, "double-free", &fops_double_free))
                pr_debug("Faulty: Double free bug at debugfs "
                        "'%s/double-free'\n", root);

        if (!init_endpoint(dir, "use-after-free", &fops_use_after_free))
                pr_debug("Faulty: Use-after-free bug at debugfs "
                        "'%s/use-after-free'\n", root);

        uninitialized = kmalloc(sizeof(struct a_struct), GFP_KERNEL);

        if (!init_endpoint(dir, "infoleak", &fops_infoleak))
                pr_debug("Faulty: Infoleak at debugfs '%s/infoleak'\n", root);

        pr_debug("Faulty: module loaded\n");
        return 0;

}

static void __exit mod_exit(void)
{
        debugfs_remove_recursive(dir);
        kfree(race1);
        kfree(race2);
        kfree(uninitialized);

        pr_debug("Faulty: Unloaded faulty kernel module\n");
}

static int init_endpoint(struct dentry *dir, const char *fn,
                        const struct file_operations *fops)
```

```
{
        struct dentry *fil = debugfs_create_file(fn, 0644, dir, NULL, fops);

        if (fil == NULL) {
                pr_err("Faulty: Cannot create endpoint %s\n", fn);
                return -ENOENT;
        }

        return 0;
}

static ssize_t sbo_read(struct file *fps, char __user *buf, size_t len,
                        loff_t *offset)
{
        return simple_read_from_buffer(buf, len, offset, buffer,
                                       strlen(buffer));
}

static ssize_t sbo_write(struct file *fps, const char __user *buf, size_t len,
                         loff_t *offset)
{
        int kbuf_size = 10;
        int flag = 0; // variable to clobber
        char kbuf[kbuf_size];
        int bytes_written = 0;

        // FAULT: stack buffer overflow
        // length of the incoming data is used instead of
        // target buffer length (kbuf_size)
        bytes_written = simple_write_to_buffer(kbuf, len, offset,
                                               buf, len);

        // TODO: another fault here?
        //strncpy(buffer, kbuf, len);

        // we'll bypass stack canary evasion at this time
        if (flag != 0)
                non_reachable_function();

        return bytes_written;
}


static ssize_t slab_read(struct file *fps, char __user *buf, size_t len,
                        loff_t *offset)
{
        slab_operate_with_other_data();

        if (!user_controlled) {
                pr_debug("Faulty: Slab - Read, no data\n");
                return 0;
        }

        pr_info("Faulty: Slab - Read, there is data\n");
        return simple_read_from_buffer(buf, len, offset,
                                user_controlled->data,
                                strlen(user_controlled->data));

}

static ssize_t slab_write(struct file *fps, const char __user *buf,
                        size_t len, loff_t *offset)
{
        slab_operate_with_other_data();

        if (!user_controlled) {
                pr_debug("Faulty: Slab - Write, No data\n");
        } else {
                pr_debug("Faulty: Slab - Write, Free old data\n");
                kfree(user_controlled);
        }
        user_controlled = kmalloc(sizeof(struct some_data), GFP_KERNEL);

        // TODO test conditions
        if (other_data->flag_which_is_never_set)
                non_reachable_function();

        // FAULT: heap buffer overflow
        return simple_write_to_buffer(user_controlled->data, len, offset,
                                buf, len);

}

// TODO: make this double freeable
static void slab_operate_with_other_data(void)
{
        if (!toggle) {
                toggle = true;
                pr_debug("Faulty: Slab - allocating other data");
                other_data = kzalloc(sizeof(struct some_data), GFP_KERNEL);
        } else {
                pr_debug("Faulty: Slab - freeing other data");
                kfree(other_data);
                toggle = false;
        }
}

static ssize_t unsigned_overflow_read(struct file *fps, char __user *buf,
                                size_t len, loff_t *offset)
{
        char *buffer = kmalloc(BUF_SIZE, GFP_KERNEL);
        ssize_t n = 0;

        // FAULT: unsigned overflow
        snprintf(buffer, BUF_SIZE, "Faulty: Overflow - Counter value :%d\n",
                unsigned_counter++); // note the behaviour of counter

        if (unsigned_counter == 1)
                non_reachable_function();

        n =  simple_read_from_buffer(buf, len, offset, buffer,
                                       strlen(buffer));
        kfree(buffer);
        return n;
}

static ssize_t signed_underflow_read(struct file *fps, char __user *buf,
                                size_t len, loff_t *offset)
{
        char *buffer = kmalloc(BUF_SIZE, GFP_KERNEL);
        ssize_t n = 0;

        // FAULT: signed underflow
        snprintf(buffer, BUF_SIZE, "Faulty: Underflow - Counter value :%d\n",
                signed_counter--); // note the behaviour of counter

        if (signed_counter == 126)
                non_reachable_function();

        n =  simple_read_from_buffer(buf, len, offset, buffer,
                                       strlen(buffer));
        kfree(buffer);
        return n;
}

static ssize_t format_read(struct file *fps, char __user *buf, size_t len,
                        loff_t *offset)
{
        return simple_read_from_buffer(buf, len, offset, some_string,
                                       strlen(some_string));
}

static ssize_t format_write(struct file *fps, const char __user *buf,
                        size_t len, loff_t *offset)
{
        char buffer[BUF_SIZE];
        ssize_t n;

        n = simple_write_to_buffer(&buffer, BUF_SIZE, offset, buf, len);
        buffer[n] = '\0';
        pr_info("Faulty: %s\n", buffer);
        // pr_info(buffer); // this would generate a compile-time error
        // FAULT: format-string
        printk(buffer);
        return n;
}

static ssize_t race_read(struct file *fps, char __user *buf, size_t len,
                        loff_t *offset)
{
        if (strcmp(race1, race2))
                non_reachable_function();

        return simple_read_from_buffer(buf, len, offset, race1,
                                strlen(race1));
}

static ssize_t race_write(struct file *fps, const char __user *buf,
                        size_t len, loff_t *offset)
{
        // FAULT: stack overflow
        char buffer[PAGE_SIZE];
        ssize_t n;

        n = simple_write_to_buffer(&buffer, PAGE_SIZE, offset, buf, len);
        buffer[n] = '\0';

        // FAULT: race
        // slow write is racy
        memcpy(race1, buffer, len);
        udelay(1000);
        memcpy(race2, buffer, len);

        return n;
}

static ssize_t df_alloc(struct file *fps, char __user *buf,
                        size_t len, loff_t *offset)
{
        pr_info("Faulty: double-free allocation\n");
        double_free = kmalloc(len, GFP_KERNEL);
        return len;
}
static ssize_t df_free(struct file *fps, const char __user *buf,
                size_t len, loff_t *offset)
{
        // FAULT: double free
        pr_info("Faulty: double-free deallocation\n");
        kfree(double_free);
        return len;
}

static ssize_t use_after_free_read(struct file *fps, char __user *buf,
                                size_t len, loff_t *offset)
{
        char *tmp = kmalloc(len, GFP_KERNEL);

        strncpy(tmp, buffer, len);
        // FAULT: use after free
        kfree(tmp);
        copy_to_user(buf, tmp, len);
        return len;
}


static ssize_t infoleak_read(struct file *fps, char __user *buf,
                        size_t len, loff_t *offset)
{

        ssize_t l = len < DATA_LEN ? len : DATA_LEN;

        return simple_read_from_buffer(buf, len, offset,
                                uninitialized->data, l);
```

```
}

static void non_reachable_function(void)
{
        pr_info("Faulty: This function should not be reachable.\n");
}

module_init(mod_init);
module_exit(mod_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("A Kernel Module with Faults");
MODULE_AUTHOR("Ilja Sidoroff");
```