AN ABSTRACT OF THE THESIS OF

Abhilash Raj for the degree of Master of Science in Computer Science presented on December 15, 2017.

Title: A Decade of Linux Kernel Vulnerabilities, their Mitigation and Open Problems

Abstract approved: ______________________________

Rakesh Bobba

The aim of this thesis is to study past 10 years of security vulnerabilities reported against Linux Kernel and all existing mitigation techniques that prevent the exploitation of those vulnerabilities. To systematically study the security vulnerabilities, they were categorized into classes and sub-classes based on their type.

This thesis first examines over 1100 Common Vulnerabilities and Exposures (CVEs) reported against Linux Kernel in the period of past 10 years. It then presents a survey of techniques that exist today to prevent exploitation or mitigate impact of these vulnerabilities. Techniques surveyed include those added to Linux kernel in past few years, notable patches and those proposed in research papers but not yet adopted. Finally, based on the above analysis, this thesis discusses the gaps in the security of Linux Kernel that cannot be efficiently mitigated using the existing techniques and explores open problems for future research.

A Decade of Linux Kernel Vulnerabilities, their Mitigation and Open Problems

by

Abhilash Raj

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented December 15, 2017
Commencement June 2018

ACKNOWLEDGEMENTS

I would never have been able to finish my thesis without the guidance of my committee members and support from friends and family.

I would like to express my deepest gratitude to my Advisor, Prof. Rakesh Bobba, for his guidance, care and help. He provide me an excellent research environment, patiently corrected my writing and financially supported my research. It wouldn't have been possible to finish this work without him.

I would like to thank my committee members D. Kevin McGrath, Prof. Mike Rosulek and Prof. Joseph Louis for their invaluable comments and review of this work.

Finally, I would like to thank my parents and elder brother, who always supported me with their best wishes.

TABLE OF CONTENTS

Page

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

Chapter 1: Introduction

GNU/Linux is one of the most popular general purpose operating system used today because of its robust performance, free availability and open nature of development. All discussions on development and direction of the project happens in public mailing lists and open for anyone to participate. Today, there are well over 1 billion devices running GNU/Linux including smartphones (e.g. Android), embedded devices, autonomous cars, robots (e.g. ROS) and most of the servers running the Internet.

There are over 18 million lines of code in last release of Linux, licensed under GNU General Public License, version 2 [2]. According to Distrowatch [3] , a popular website tracking Linux distributions, there are 275 GNU/Linux distributions including special purpose distributions built to work on old hardware, embedded devices, desktops, special servers etc. Henceforth, any mention of the term *Linux* corresponds to the Linux Kernel unless explicitly specified.

1.1 Linux and Security

Security is an important aspect of any operating system and a large number of critical infrastructures today runs on Linux, it is important to asses its current state of security. There are over 1100 CVEs[1] reported against Linux in past 10 years (2007-2016).

While this number is alarmingly high for a project on which a lot of critical infrastructure depends today, by itself, it's a poor metric to measure Linux's security. Vulnerabilities are simply bugs in a software that have security implications. Measuring the security impact for each bug may not be trivial in case of a system software like Linux. Vulnerabilities found via testing techniques like fuzzing, static analysis, regression testing are often not exploitable.

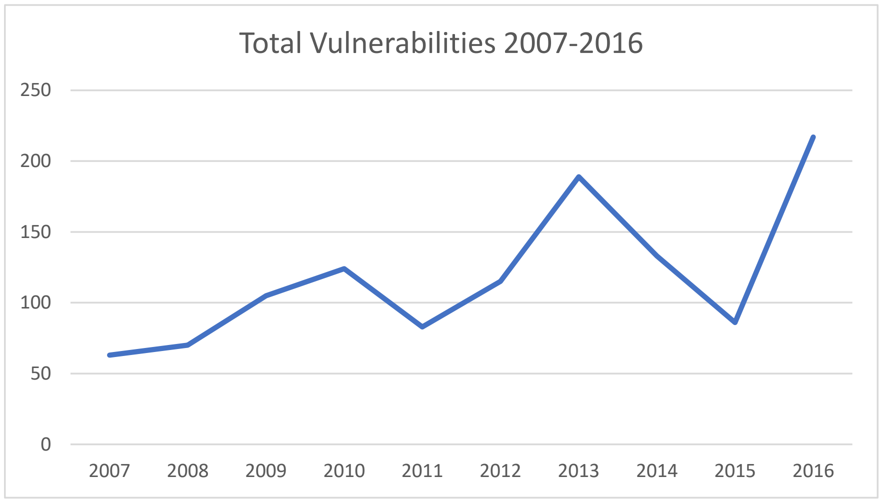


Figure 1.1: Yearly distribution of CVEs for the past decade

1.2 Problem to be addressed by this thesis

Most vulnerabilities, or software bugs in general, often follow a pattern that is repeated. Multiple variants of a vulnerability can be found in the different parts of the code. These types of vulnerabilities have been known to us for a long time, but, they are still show up frequently in Linux. If we look at the frequency of these vulnerabilities over past 10 years, we see that they have always been on rise. It can safely be assumed that unless there is a groundbreaking change in the process of how Linux is developed, these vulnerabilities will continue to show up.

So, what can be done to make Linux more secure, beyond fixing the vulnerabilities as they are found? To answer this question, we take a step back and notice exactly what

[1]CVE database [4] is a common database of all the security vulnerabilities reported against softwares, maintained by a non-profit organization called MITRE Foundation [5].

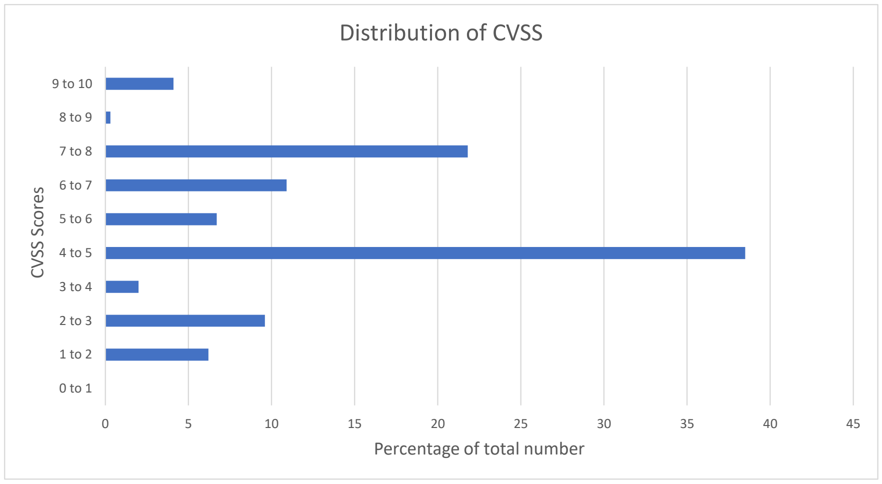


Figure 1.2: Relative distribution of Common Vulnerability Score (v2.0) for past 10 years of Linux kernel CVEs.

a vulnerability is. According to RFC2828 [6], a vulnerability is

> *A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy.*

Being exploitable is what differentiates vulnerabilities from other software bugs. *Proactive defenses* prevent the exploitation of known kind of vulnerabilities, thus rendering them harmless from a security standpoint.

Something important to note here that CVEs are assigned to bugs that may or may not be exploitable. The rationale behind this is based on the fact that whether or not a bug is exploitable (and thus has security related consequences in real world) is dependent on environment and configuration. Given that there are huge number of permutations in which a Linux system can be configured, it is extremely difficult to predict if a bug can be exploited or not.

1.3 Importance of this thesis

A large number of proactive defenses have been proposed both in research community and industry. *The PaX project* [7], first launched in the year 2000, hardens Linux against majority of memory corruption bugs discussed in Chapter 3. *grsecurity* provides defenses against some memory corruption vulnerabilities, a Role Based Access Control (RBAC) system and strict policies for access control in user-space. Over the years, The PaX Team and grsecurity have teamed up to maintain patches for stable version of Linux which includes defenses from both projects. Since April 2017, PaX/gesecurity patches are only available to paying customers of Open Source Inc.

Research community has also produced countless research papers targetted to specific kinds of vulnerabilities and attacks with varied amount of success in terms of industry adoption, performance and security coverage.

This thesis tries to bring together the ideas from research community and industry to present a coherent view of the current state of the art defenses and techniques. It tries to compare security coverage and guarantees along with their performance to understand the practical usability of these defenses.

It also includes a quantitative survey of security vulnerabilities reported against Linux kernel in past decade and categorizes them into various classes and sub-classes based on identifiable patterns in them. Figure 1.3 shows the relative number of vulnerabilities of different types that we identified. Each of these vulnerabilities are discussed in detail in Chapter 3. The final goal of this thesis is to analyze which of these classes do not have efficient and practical proactive defenses in use today.

1.4 Thesis structure

This thesis is organized as follows:

- Chapter 2 provides background for relevant topics covered in this thesis.
- Chapter 3 provides detailed overview of different classes of vulnerabilities, their

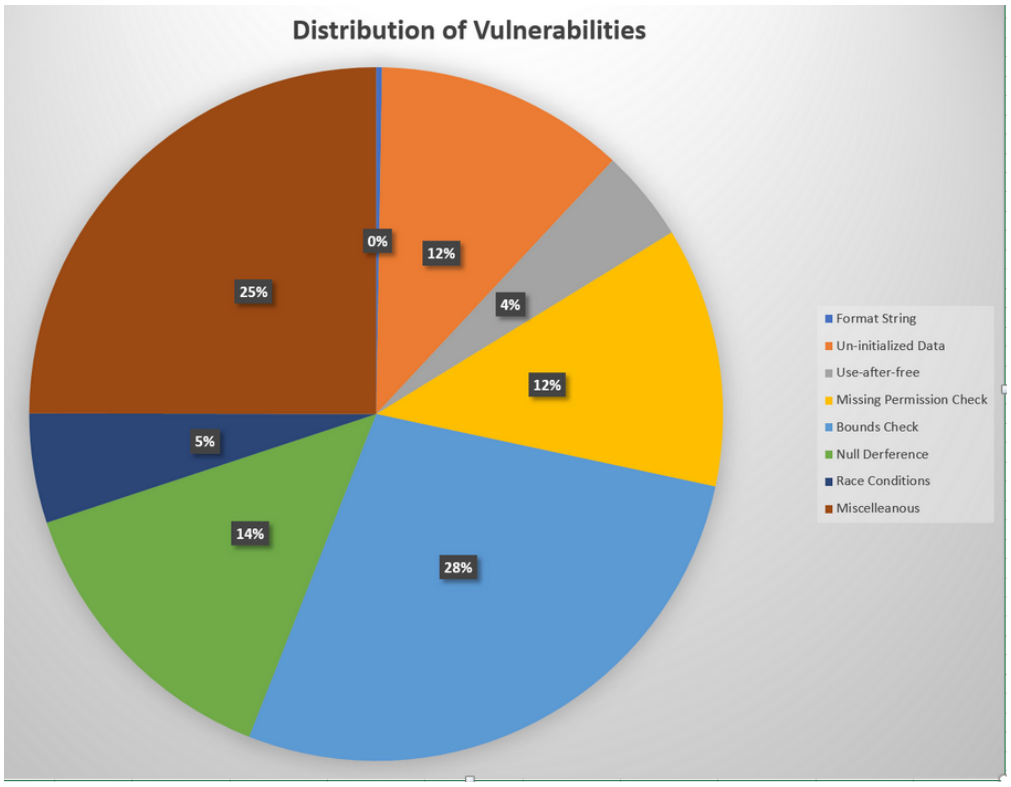


Figure 1.3: Relative occurrences of different types of vulnerabilities

sub-types, how can they be exploited and lists all their defenses.

- Chapter 4 describes attack strategies and defenses of Return-Oriented-Programming attacks, which are based on multiple types of vulnerabilities that are discussed in Chapter 3.
- Chapter 5 concludes this thesis

Chapter 2: Background

2.1 What is a CVE?

Common Vulnerabilities and Exposures (CVE) is a central database of publicly known cyber-security vulnerabilities [4]. Unique identifiers for each vulnerability is assigned by a CVE Numbering Authority (CNA), organizations authorized to identify and assign CVE IDs to vulnerabilities, and published by the primary CNA, The MITRE Corporation. To request a CVE number, one needs to contact one of the CNAs along with the bug report and other relevant information related to fixes. The MITRE Corporation publishes a list of CVEs on the CVE Website [8] after it has been disclosed and fixed. To prevent disclosing a vulnerability before it has been fixed, The MITRE Corporations coordinates with the original projects on the release date.

For Linux kernel, security fixes are often coordinated among the major distributions like RedHat, Debian, OpenSUSE etc. and the vulnerability is published on the CVE Website only after the relevant security patches have been made available.

Common Vulnerability Scoring System (CVSS) is an open industry standard [9] for scoring vulnerabilities based on factors like access complexity, access vector, authentication requirements and impact on confidentiality, availability and integrity.

2.2 What is a Kernel?

A kernel is a system software that manages hardware and facilitate its access to application software. Linux is an operating system kernel written in C. The code for kernel is loaded in a special memory area called *kernel-space* and is mediated through support of privileges from hardware. Loading the kernel in a privileged space prevents it from being overwritten by other programs. A kernel provides an abstraction layer over hardware functionalities through *system calls*.

Linux is a monolithic kernel, which means everything runs in the same address space for performance reasons. Microkernels, run only minimally required services in kernel

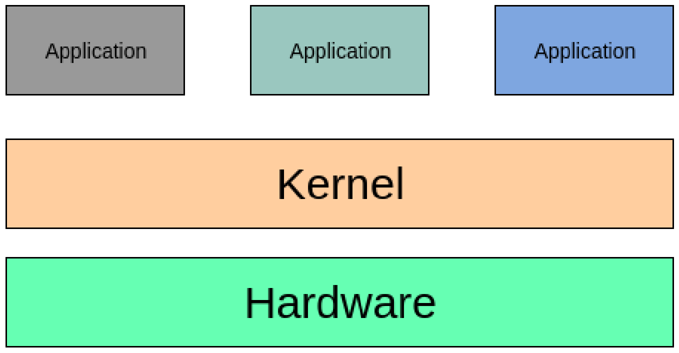


Figure 2.1: An Operating System Kernel

space and the rest of the services run in a separate address space for non-privileged processes. This makes Microkernels lean reducing their startup time and the size of the computing base that needs to be trusted.

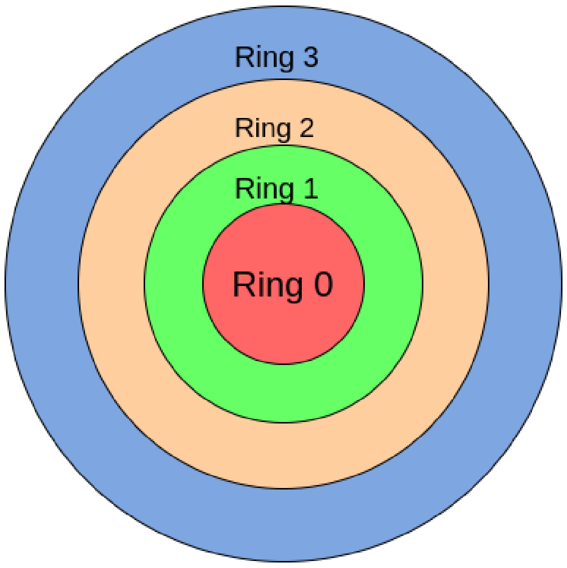


Figure 2.2: Privilege rings in x86 architecture.

Privilege rings are heirarchial In x86 systems, there are four privilege rings from 0-3 with 0 being the most privielged and 3 being the least privileged ring (see fig 2.2). Linux however uses only two of these rings, ring 0 for kernel (also called as supervisor mode) and ring 3 for application software.

2.2.1 Virtual Memory

Virtual Memory is a technique used by modern operating systems for memory management that presents a large contiguous range of (virtual) address space to an application. Modern hardware include an *address translation unit* (a.k.a *Memory Management Unit* (MMU)) which automatically translates virtual addresses (corresponding to location in virtual memory area) to physical addresses (corresponding to actual address in hardware) in the CPU. The range of Virtual Memory Area (VMA) can exceed the capacity of the physical memory, which allows application to address more memory than the physical capacity.

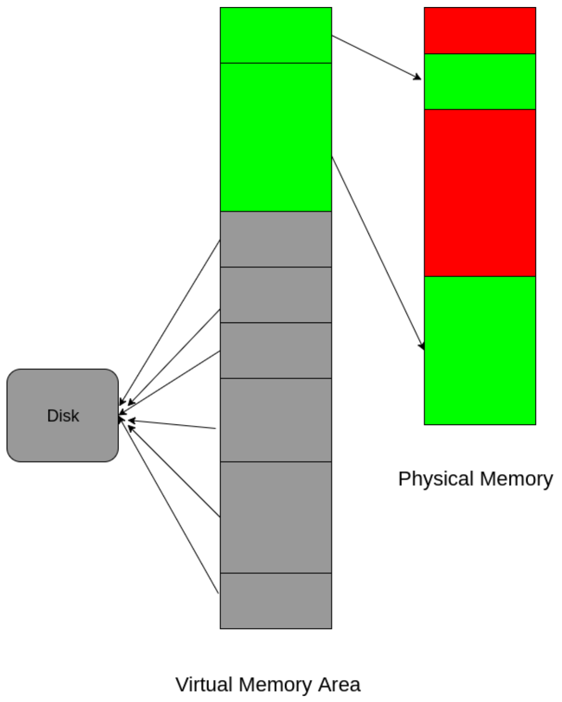


Figure 2.3: Virtual Memory Area. The green regions are portions of VMA that are mapped to physical memory. The red regions denote the physical memory that belong to other processes and are not-accessible to current process. The remaining grey regions in VMA denote the memory regions that are not currently in physical memory and are either present on the disk or empty.

2.2.2 Memory Management and Paging in Linux

Paging is a mechanism that translates a linear memory address to a physical memory address. Entire memory of an operating system is divided into small chunks, which makes it easy to addresses them. These chunks are called *pages.* In Linux, pages are commonly 4Kb in size, however, *huge pages* are bigger in size and can be either 4Mb or 1Gb in size.

Each page has an associated metadata, which includes information about the current status of the page, if the page is currently in the memory or is it swapped to disk, the current permission level etc.

2.2.3 Memory Allocators in Linux

Some portion of the RAM is permanently reserved for kernel to use and stores both the kernel code and the static kernel data structures. The remaining part is called dynamic memory and is managed and allocated by kernel to user-space and kernel-space processes.

Zone Allocators

The *zone page frame allocator* takes care of memory allocation requests for groups of contiguous page frames. *Zones* are specific regions of physical memory allocated for specific purposes, for example `ZONE_DMA` is for direct memory access, `ZONE_HIGHMEM` is used for higher range in memory, `ZONE_NORMAL` is used for other normal requests. Each zone also has a *per-CPU page frame cache* that is a cache of single pages for frequent use.

Buddy Allocators

Inside each zone, *buddy allocator* manages individual pages. Buddy allocator handles requests for contiguous page frames grouped into sizes of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024 contiguous pages. These pages have a contiguous linear address but can possible be fragmented in physical memory. Buddy allocators prevent fragmentation of linear addresses and are efficient as they deal with only page-sized chunks of memory.

Slab Allocators

While buddy allocators are efficient as they deal with only large chunks of memory, it is quite wasteful to request an entire page to store only a few bytes. Slab allocators act as memory caches with reserved pages based on the type of data to be stored, resulting in very fast memory allocations.

Various parts of kernel tend to request a similar sized chunk of memory very frequently. Instead of un-allocating those memory regions, Slab allocator re-uses the free'd memory region for next request, essentially recycling the memory region. This results in much faster memory allocation as memory region is never de-allocated and re-allocated.

Linux includes three Slab allocators:

- **SLOB allocator**: This is the first type of Slab allocator, built with focus on compact memory storage
- **SLAB allocator**: This allocator is confusingly also named as SLAB and is built with cache-friendliness in mind by aligning objects to cache-boundaries
- **SLUB allocators**: This is newest type of Slab allocator and is built with focus on simple instruction counts, better support for debugging and defragmentation

Chapter 3: Vulnerabilities and Defenses

To study the vulnerabilities, a list of all the CVEs reported against Linux Kernel was collected from CVE Details website [10] which includes CVSS score along with each CVE and it's report.

A total of 1109 vulnerabilities were collected and studied as a part of this study. Fig 3.1 presents the categorization of vulnerabilities into classes and sub-classes. Here is a brief introduction of all the classes:

Vulnerability Class	**Type**	**No. of CVEs**	**% of Total**
Un-initialized Data		128	11.54%
Use-after-free		47	4.24%
Bounds Check	Heap Overflow	87	7.84%
	Stack Overflow	30	2.71%
	Buffer Over-read	28	2.52%
	Integer Overflow	57	5.14%
	Refcount Overflow	11	0.99%
	Integer Underflow	10	0.90%
	Integer Signedness	14	1.26%
	Array Index Errors	19	1.71%
Null Derference		149	13.44%
Format String		3	0.27%
Missing Permission Check		133	11.99%
Race Conditions		56	5.05%
Miscelleanous	Infinite Loop	12	1.08%
	Memory Leak	17	1.53%
	Divide-by-zero	10	0.90%
	Cryptography	8	0.72%
	Length Calculation Bugs	19	1.71%
	Other	224	20.19%
Total		1109	

Table 3.1: No. of CVEs reported in each of the categories.

- **Un-initialized data** vulnerabilities are a result of missing initialization of data structures before they are exposed outside of kernel-space memory.
- **Use-after-free** vulnerabilities happen when there exists a reference to a freed

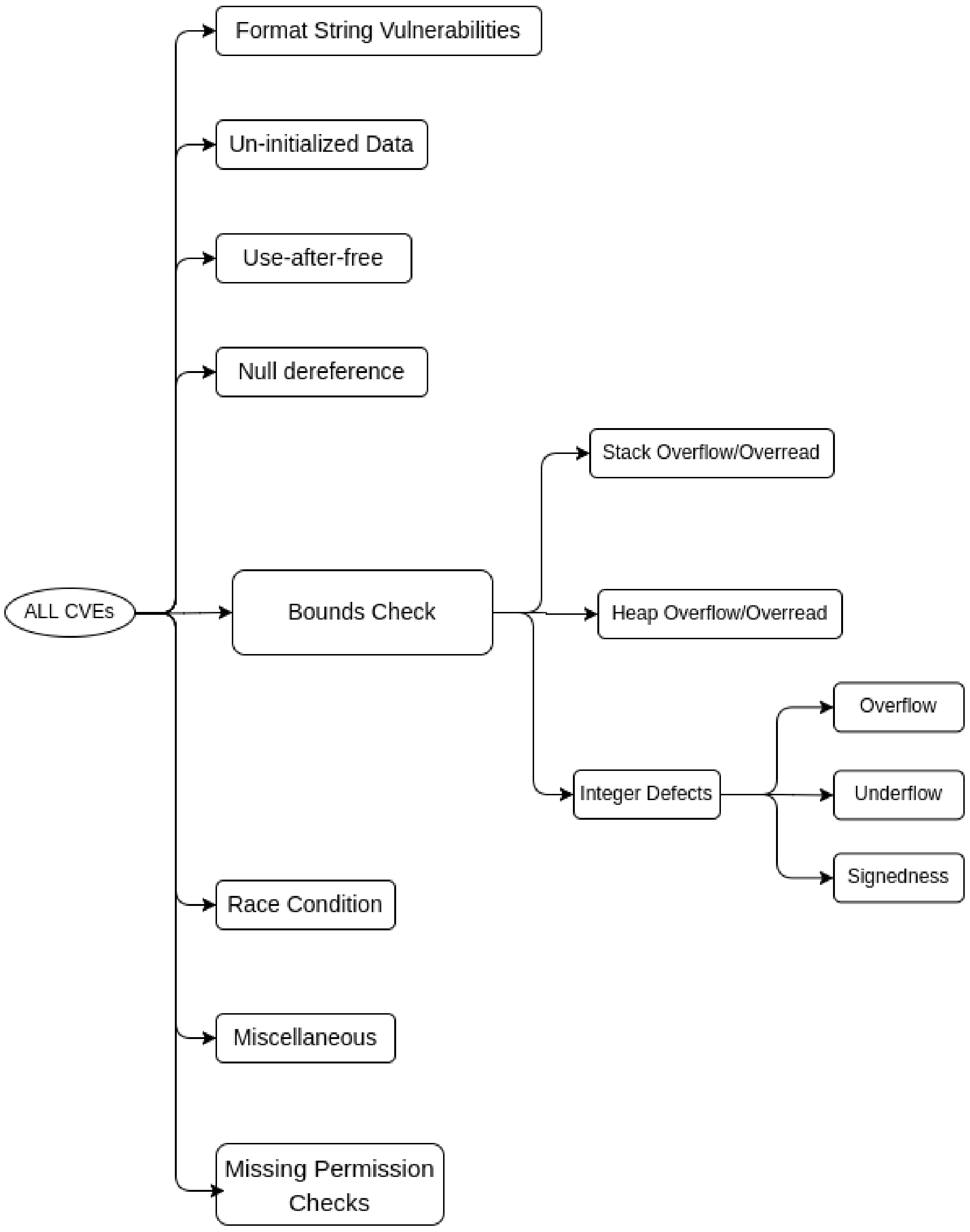


Figure 3.1: Categorization of Vulnerabilities

memory region which can be exploited by placing a valid new object at the same memory region.

- **Bounds** vulnerabilities happen because of missing or wrong bounds checks on data moved between kernel-space and user-space memory. It also includes wrong length calculations and array index errors vulnerabilities.
- **Null derference** vulnerabilities are result of NULL pointers being dereferenced before null-check in kernel.
- **Format String** vulnerabilities happen when un-trusted format strings find their way into string formatting functions leading to information disclosure.
- **Race conditions** occur because of poor handling of locks around critical sections of memory when accessed by multiple threads or processes.
- **Miscellaneous** includes all the other vulnerabilities that do not fall into any of the categories mentioned above. Some of these do have some commonality in their effects or side effects and are sub-grouped in those types.

Following sections explore each of the above mentioned classes, their exploitation methodology and existing defenses against them. Defenses were collected from the last public release of PaX/grsecurity (April 2017), proactive defenses added to Linux in past few years and research papers published in past 6 years.

3.1 Un-Initialized Data

An un-initialized object can leak sensitive information from kernel memory when it's moved across privilege boundaries like user-space memory, network, filesystem etc. This happens when a region of memory allocated to an object contains sensitive information and because of the missing initialization, the sensitive data continues to persist in the object. Such object, when copied to user-space memory for example, leaks information previously stored in kernel memory. Missing initialization of function pointers can cause NULL pointer dereference leading to OOPS[1] (CVE-2011-2184), privilege escala-

[1]OOPS is a Linux terminilogy for deviation from normal behavior of the kernel. It may result in a Kernel panic or crash but can also result in continued operation with reduced reliability. OOPS often results in an error log which can help administrators debug the actual cause of crash.

tion (CVE-2009-2692, CVE-2008-2812) and other potential attacks discussed in section 3.4.

A total 128 vulnerabilities of this kind were reported against Linux in past 10 years, which accounts for 11.54% of total vulnerabilities. Information obtained by exploiting them is useful in building attacks that break other defenses like Address Space Layout Randomization (ASLR). ASLR is a technique used to randomize the base address of various sections in the Kernel memory in order to thwart the attacks that re-use the code segments by indirect call transfers. They are called code-reuse attacks or return-oriented programming attacks and are discussed in section 4.1.

Another source of un-initialized data vulnerabilities is padding added to *word* align [2] the data structures at compile time. This padding is often[3] un-initialized memory and is out of programmer's control since they are added at compile time.

3.1.1 Mitigation Techniques

There are several defense techniques proposed to defend against un-initialized data related attacks. Table 3.2 provides a brief summary of all the defense techniques ordered by the year they were first published in public domain. Each of the defenses are then mentioned in detail along with their defense strategy, performance overheads and coverage.

PaX

PaX patchsets are a collection of security enhancements to Linux that defend against several vulnerability classes that lead to memory corruption.

PAX MEMORY STACKLEAK is a GCC plugin from PaX which clears the kernel stack before a function returns to the user-space. While this can prevent leakages from any previous system call, data from current system call will still leak. According to the analysis performed by Lu. et. al. [13], its performance overhead is high and can rage

[2]A *word* length is a specific property of CPUs and is defined by it's architecture. Majority of the registers in a CPU can hold data of *word* size.

[3]This varies from compiler-to-compiler and depends on data structure and type of initialization(e.g. initialized with constants or variables).

Name	Year	Coverage	Bypassable	Cost
Chow et. al [11]	2005	pages un-used for fixed time	yes	N/A
`PAX_MEMORY_STACKLEAK`	2011	Kernel stack leakages	yes, in current system call	2.6-200%
`PAX_MEMORY_STRUCTLEAK`	2013	structures with `__user` fields	no	¡ 1%
`PAX_MEMORY_SANITIZE`	2013	memory allocated by slab allocators & buddy allocator	no	9-12%
Peiro et. al. [12]	2014	Kernel stack, single function boundary, compiler padding	yes, with 0.8% probability	N/A
Unisan [13]	2016	security sensitive un-initialized data	no	1.5%
Memory Sanitization in Linux	2017	memory allocated by slab allocators	yes, by large memory allocated by buddy allocator	3-20%
SafeInit [14]	2017	security sensitive un-initialized data	no	-3% - 5.9%

Table 3.2: Mitigation Techniques for Un-initialized data errors.

from 2.6% to 200% depending on the workload.

PAX MEMORY STRUCTLEAK is a GCC plugin, also a part of PaX, which zero initializes local structures that can be copied to the user-space in future as a preventive measure for missing initializations. The performance impact of this is less as compared to `PAX_MEMORY_STACKLEAK` but it offers less coverage. The fields to be zero-initialized are determined based of `__user` annotations of the fields. However, `__uesr` doesn't cover all the structures that will be copied to user-space and isn't explicitly used only for the structures that will be copied to user-space in future. It also doesn't recognize all the different types of initializers leading to some false positives.

PAX MEMORY SANITIZE erases memory pages (allocated using *buddy allocator*) and slab objects (allocated using *slab allocator*) as soon as they are freed by writing NULL values to them. This technique is called **memory sanitization**, by erasing the memory pages when they are *freed*, the lifetime of sensitive data can be reduced preventing future leaks. For better performance, one can disable erasing of Slab objects at the cost of lesser coverage.

Protections added to Linux

Memory Sanitization Linux v4.6 gained partial support for memory sanitization, a port of `PAX_MEMORY_SANITIZE`. Sanitization happens at both slab allocator level and page allocator level by writing a magic value to the freed regions. It includes an additional *sanity check* which would verify that nothing was written to the memory by checking for magic value during allocation time. Because of the high performance overhead, this feature is disabled by default.

STRUCTLEAK `PAX_MEMORY_STRUCTLEAK` GCC plugin was ported over to Linux 4.14 which zero-initializes all the local variables which are passed by reference without being initialized. `GCC_PLUGIN_STRUCTLEAK_BYREF_ALL` is the configuration option that enables this feature.

Proposed Defenses in Research

One of the oldest technique to detect un-initialized data at compile-time is static analysis of the source code. Peiro et. al. [12] use taint analysis to track memory regions from source (i.e. allocation) to their sink (e.g. copy to user-space, sent over network) to make sure that no un-initialized data structures leak. Their implementation can only detect allocations on the stack and can only track memory within a single function boundary.

Lu et. al. [13] proposed UniSan to detect unsafe allocations and automatically initialize variables using binary instrumentation techniques. UniSan is built as a tool that takes LLVM IR (intermediate representation i.e. bitcode files) as input and returns instrumented binary. Because initializing all variables would incur a high performance overhead, it uses static data flow analysis to initialize only those variables which can leak to user-space.

Milburn et. al. [**?**] proposed SafeInit, as a plugin to LLVM compiler toolchain, which initializes all variables unconditionally and then optimizes their code to remove initialization for certain variables that are never used in any context that they can be exposed. They claim to fix a wider range of un-initialized data vulnerabilities than UniSan.

Chow et. al. [11] proposed a strategy in year 2005 called *secure de-allocation*, which involves writing *zero* values to the memory pages after they are freed. Kernel pages are marked *dirty* when they are freed and can only be used after they have been cleaned (zeroed) by a long running kernel daemon thread. This thread periodically scans dirty pages and zeroes them if they have been dirty for a certain fixed time. A downside to this scheme is that it doesn't seem to cover slub allocators like SLUB, SLAB or SLOB which can re-use the same pages for different objects without freeing them.

One thing to note here is that all the memory sanitization techniques mentioned above, that sanitize memory region when they are freed, do not provide absolute safety against information disclosure. Memory leaks when the memory is never freed cannot be covered by these methods in long running daemon processes for example.

3.1.2 Analysis

Un-initialized data vulnerabilities make up for 11.54% of all the vulnerabilities found (see Table 3.1). Defense mechanisms from PaX/grsecurity (`PAX_STACKLEAK`, `PAX_STRUCTLEAK`, `PAX_MEMORY_SANITIZE`) have very high overheads as mentioned in Table 3.2. `PAX_STACKLEAK` also doesn't prevent from leaks in the current system call as it clears the stack for all previous system calls. Linux gained support for memory sanitization, a port of `PAX_MEMORY_SANITIZE`, but it also suffers from same problem of high performance overhead and is thus disabled by default.

Research papers like UniSan [13] and SafeInit [14] can achieve acceptable performance and show promising guarantees. They zero-initialize only the data structures that are ever exposed outside of kernel. According to their analysis, they are able prevent all the un-initialized data vulnerabilities reported against Linux from being exploited. Both of them can also detect leaks caused by paddings added by compiler to optimize the size of data structures by aligning them with processor *word length.* These paddings can also reveal sensitive information and is beyond the control of a programmer since they are added during compile time.

3.2 Use-After-Free

Use-after-free (UAF) vulnerability occurs when an object in memory is accessed after it has been destroyed (freed). These are also termed as *temporal memory errors* or *dangling pointers* [4]. Most of the UAF vulnerabilities target heap allocations as they are managed

[4]A pointer pointing to a deleted object is called *dangling pointer.*

manually, but sometimes a local variable can escape from local scope if they are assigned to a global pointer causing a temporal memory error after the function returns and stack frame is cleared. Stack based UAF vulnerabilities haven't been reported for Linux in past 10 years.

A total of 47 UAF vulnerabilities were reported against Linux in past 10 years which accounts for 4.24% of total vulnerabilities reported. *Double free* vulnerabilities are a special case of UAF when the memory region is freed twice leading to panic (denial of service) (CVE-2012-1853, CVE-2010-3080).

Most cases of UAF vulnerabilities end up with a Kernel OOPS or panic as the pointer being used often points to invalid memory. But, in some cases, the pointer could be valid if a new object was allocated on the same memory region the pointer points to. This can be used to point a legitimate pointer to an attacker controlled object. Exploiting a UAF vulnerability commonly involves three steps:

1. An object is freed while a pointer to it exists,
2. Second object is allocated on the memory pointed by the pointer in (1),
3. Kernel de-references the pointer to get attacker controlled data,

Step 1

Executing all these three steps in correct order is important for a successful exploitation. CVE-2016-0728 [15] allowed local users to gain privileges by controlling a kernel keyring object due to use-after-free vulnerability. This particular vulnerability was a result of reference counter overflow, which allowed the object to be freed when its reference counter reached zero. Reference counters are implemented as integers in Linux, which makes them vulnerable to overflows 3.3.3. A total of 11 reference counter related vulnerabilities were reported in our analysis of past 10 years.

A mis-management of reference count causes an object to be freed, even when there are valid references to the object(CVE-2009-3624, CVE-2014-2851). Bugs in the memory management code which allows objects to be freed pre-maturely is the single largest cause of UAF vulnerabilities. Often this happens due to corner cases and un-common errors which are not handled properly.

Step 2

Linux uses *freelist* [5] based memory allocators for objects called *Slab allocators*. There are three Slab allocators in Linux , SLAB, SLUB and SLOB. Each Slab allocator is built upon a different philosophy and use-case, but they are often predictable when allocating objects. By allocating multiple objects of same size, an attacker can reliably control the allocation of one of the objects over the same memory region as the previously freed object [16]. This is **step 2** in the attack.

Step 3

After Step 1 and 2, the setup for UAF vulnerability is complete and the only thing remaining is the final execution. When the kernel de-references the pointer, which now is pointing to an attacker controlled object, it inadvertently gets the attack controlled data. If the object consisted of function pointers, the attacker can replace entire functions to execute arbitrary code.

3.2.1 Mitigation Techniques

To systematize the study of mitigation techniques, they are grouped under the step that they try to prevent in the attack scenario mentioned in previous section. Table 3.3 provides a summary of all the defensive techniques ordered by the year that they were first published.

Name	Year	Step	Coverage	Bypassable	Cost
KASAN	-	3	All memory regions	yes, re-allocated memory	n/a[6]
TaintTrace [17]	2006	3	All memory regions	no	5.53%
DieHard [18]	2007	2	Heap allocations	yes, probabilistic protection only	~6%
CETS [19]	2010	3	All memory regions	yes, re-allocated memory	~48%
`PAX_REFCOUNT`	2015	1	all `atomic_t`	no	~0%
Randomized Freelists	2016	3	Slab Allocators	yes, heap spraying attacks	~0%
`refcount_t`, Linux	2017	1	only objects using it	no	~0%
Oscar [20]	2017	2-3	All memory regions	no	4%

Table 3.3: Mitigation Techniques for Use-after-free vulnerabilities.

[5]Freelists are linked lists of pre-allocated memory regions for object(s) of a particular size for faster allocation.

Prevention of Step 1

To prevent **step 1**, pointers should not be allowed to point to freed objects. In other words, all references to a freed object should invalidated. In Kernel, often refcount[7] (mis)calculations can provide an easy way to free an object which is still in use.

`PAX_REFCOUNT`, a part of PaX, adds checks around increment of `atomic_t` data type in Linux such that its value can never overflow beyond `INT_MAX` [8]. Other uses of `atomic\_t`, which are not related to reference counts, are renamed to `atomic\_unchecked\_t` throughout the source. This feature was ported over to Linux in v4.11 to protect against reference counters overflows, however instead of preventing overflows in all `atomic_t` types, a new type `refcount_t` was added to Linux and all the reference counters will eventually be ported to use this type.

Prevention of Step 2

To prevent **step 2**, an adversary should not be allowed to reliably predict memory allocations. Since Linux v4.8, freelists in the Slab allocators like SLAB, SLUB are randomized[9] making it hard to reliably predict the memory allocations. However, by repeated memory allocations of object of one size, it will ultimately result in the previously freed object memory being reused, making the attack only slightly more difficult.

DieHard by Berger et. al. [18] emulates the semantics of a probabilistic infinite virtual memory[10] using a bitmap based fully-randomized memory manager. It uses oversized heap allocations and randomization to pick free memory regions for new allocations. DieHard was motivated by the idea of preventing memory errors. Novark et. al. extended this idea to provide better security and performance and proposed DieHarder [21], which is also a probabilistic memory allocator designed with security in mind. DieHarder uses sparse pages layout to randomly allocate pages in virtual memory, provides support for ASLR by randomizing the address of small object pages and fills freed memory with

[7]KASAN is a debugging tool.

[7]Short for *reference count*, often used verbatim in the source code

[8]INT MAX is the theoretical maximum value of an integer data type in C, it is determined by the processor architecture

[9]i.e. objects allocated do not follow a simple pattern of allocated-first-free-block

[10]Having an infinite amount of memory would mean a memory region would never have to be reused thus thwarting UAF.

random data to prevent it's misuse.

Prevention of Step 3

To prevent **step 3**, code paths that lead to dangling pointers should be avoided. Kernel Address Sanitizer (KASAN) is GCC plugin that tracks dynamic memory allocations in Kernel using compiler instrumentation techniques to prevent pointers to invalid memory regions. KASAN uses *shadow memory* [11] to track the usage of each byte and can detect its unsafe usage. It incurs a significant performance and memory overhead. However, it can be used to find use-after-free and out-of-bounds read[12] bugs during testing, which can then be fixed.

Shadow Memory has also been used in previous works like TaintTrace by Cheng et. al. [17] and Compiler Enforced Temporal Safety for C (CETS) by Nagarkatte et. al. [19] to track memory allocations in a tree, hash table or trie. A downside of tracking memory regions is that they can only detect un-safe usage of freed memory and not the memory that has been re-allocated.

CETS by Nagarkatte et. al. [19] uses shadow memory to track information about the validity of each pointer. By associating this information with pointer instead of the memory regions, a pointer can be invalidated when the memory is re-allocated by changing the value of a flag. This method is called lock-and-key mechanism where the pointer is a given a key to open the lock in the memory region. When the memory is re-allocated, the lock changes and the pointer can no longer access it. CETS can provide complete temporal safety, but only if complete spatial safety[13] is also guaranteed by some other mechanism.

Oscar by Dang et. al. [20] uses permissions on shadow virtual pages to prevent danging pointer attacks. By using a shadow virtual page, each object is allocated it's own virtual page which is then destroyed when the object is freed. Oscar presents an

[11]A seperate managed region of memory that is protected with other mechanisms to prevent un-authorized write.

[12]Out-of-bounds bugs happen when the length of buffer being read-from doesn't match the length of the data being-written to.

[13]Spatial memory errors happen when pointers become invalid by pointing to out-of-bounds objects due to array index errors or are overwritten using buffer-overflow techniques

improved model with lower performance and memory overheads compared to the works previous to it.

3.2.2 Analysis

Use-after-free vulnerabilities constitute for 4.24% of all the vulnerabilities along with around 1% of refcount overflow/mis-management related vulnerabilities which enable the first step in exploitation of a use-after-free vulnerability. By preventing reference counter overflows using `refcount_t` or `PAX_REFCOUNT_`, it is possible to prevent about 11 (about 1% of all vulnerabilities) of the use-after-free vulnerabilities. Randomized Slab freelists do provide probabilistic defense against them by making it hard to predict the memory allocations from allocators, theoretically, it is possible to bypass them using heap-spraying attacks [22].

Work done by Dang et. al in Oscar [20] and Cheng et. al. in TaintTrace [17] shows possible methods to thwart all use-after-free vulnerabilities by tracking each byte of memory in shadow space for un-safe usage but their performance overheads (table 3.3) are too high to be used in a production environment. Formalized fuzz testing (using a tool like syzkaller from Google [23]) of Linux with these features before release or during development cycles would be the most ideal use-case of these techniques.

3.3 Bounds Check

Any input that a software takes from outside its trusted domain should be validated. Except for data type, the simplest validation technique is length check – input data should not be larger than the size of memory region it copied to. This is also true for any data that is read i.e. only the amount of memory required should be copied to the destination, anything extra could reveal sensitive information outside of trusted domains. The former vulnerability is called **buffer overflow** and the latter is called **buffer over-read**. Except for the source and sink of the data, these two vulnerabilities are similar in every aspect and are collectively addressed here as **bounds check** vulnerabilities.

Depending on the region of memory the vulnerability corresponds to, like heap or stack, the effects of the overflow/over-read can vary and hence the mitigation technique. Hence, bounds check vulnerabilities are further categorized into three sub-categories:

- Stack Overflow/Over-read
- Heap Overflow/Over-read
- Integer Defects

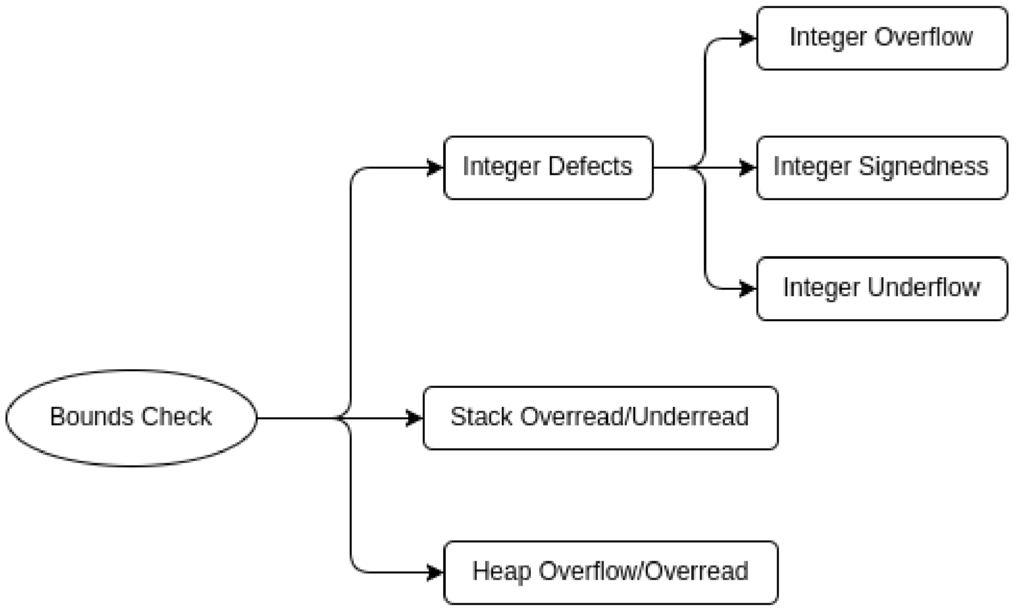


Figure 3.2: Bounds Check vulnerabilities

3.3.1 Stack Overflow

Stack overflow happens when a variable on the stack is written beyond it's bounds to neighboring memory region. Often, this is used to overwrite the return address at the bottom of the stack frame, so when the the function returns, control jumps to an attacker controlled location pointed by the value of the overwritten return address. These attacks are also called as *control flow hijacking* attacks as it changes the usual flow of the program by re-directing it to a different location in memory. Data or variables which can change the control flow of a program are termed as *control data*. Apart from compromising the control flow, a stack overflow can also be used to inject code in the Kernel and then execute them.

Stack overflow bugs can sometime overwrite beyond the current stack region to neighboring page. So, when an attacker is able to overflow a stack buffer, it can potentially corrupt the entire stack or even overwrite heap memory, which is generally allocated just below stack region.

Mitigation Techniques

Name	Coverage	Bypassable	Cost
Stack Canaries by GCC	Return address on stack	yes, direct write using a pointer	˜0%
`PAX_PAGEEXEC` / `PAX_SEGMEXEC`	code injection	no	
`GRKERNSEC_KSTACKOVERFLOW`	cross-page overflow	no	
`VMAP_STACK`, Linux	cross-page overflow	no	
`PAX_USERCOPY` / Hardened User-copy	all overflows	yes, to corrupt stack or current frame	

Table 3.4: Mitigation Techniques for Stack Overflows

There are no known ways to completely prevent a buffer overflow at hardware level, or at programming language level. The best that can be done is to detect an overflow. Most widely used protection mechanism to detect buffer overflow is adding a random value called *stack cookie* or *stack canary* after the buffer that can potentially overflow and checking it's value when function returns. Since it increases the amount of memory used and incurs some performance overhead on every function return, it is often not added un-conditionally for every buffer in the stack frame. Generally, a stack canary is added only before the return address to prevent control flow hijacking attacks by overwriting the return address.

To manage the tradeoffs between memory overheads and security, most common compilers (GCC and LLVM) have options to enable overflow protection. GCC includes three options [24] :

- `-fstack-protector-all`: This adds a stack canary to all functions and thus has high memory overheads
- `-fstack-protector`: This adds a stack canary to all the functions that call `alloca` [14] and functions with buffers larger than 8 bytes.

[14]`alloca` is used to allocate memory on stack which is automatically freed

- `-fstack-protector-strong`: This adds a stack canary to all the functions covered by `-fstack-protector`, all functions that use a local variable or local register references, and all the functions that have local array definitions.

Given that enough entropy [15] is used for canaries that they cannot be de-anonymized [16], they can detect stack overflows. It is still possible to overwrite the return address by exploiting a dangling array pointer. Dangling array pointers are arrays on stack that can be made to point to any location by controlling it's index variable.

Data Execution Prevention (DEP) or W $\oplus$ X (write xor execute) is a policy used to prevent any memory region from being marked writable and executable at the same time to prevent code injection attacks. A data region which contains attacker controlled data, if allowed to execute, could be used for code-injection attacks. Recent processors allow marking memory pages with a No eXecute (NX) bit which is used to implement W $\oplus$ X policy by removing execute permissions from all data pages. `PAX_PAGEEXEC` and `PAX_SEGMEXEC` can emulate W $\oplus$ X and NX respectively in older architectures that don't have native hardware support for it. Because they are implemented in software, they have higher performance overheads than DEP policy using NX support from hardware.

Guard Pages are commonly used to make sure any writes that cross the page boundary are trapped preventing overwrite of any neighboring data structure. Until recently, the stack region in kernel used to be directly mapped, i.e., the memory was virtually and physically contiguous. This meant that the guard pages would have to be mapped in the physical memory too making them very expensive[17]. In Linux v4.9, `CONFIG_VMAP_STACK` option was added to allow stack to be virtually mapped without any need for it to be physically contiguous, which allowed the use of guard pages to prevent overflows across page boundaries. This feature has been a part of grsecurity patches with a configuration option called `GRSECURITY_KSTACKOVERFLOW`.

[15]Entropy is a measure of randomness and depends on the source and length of data.

[16]Small values can be *guessed* using brute force techniques in reasonable time-frames.

[17]Memory is scarce in kernel-space and guard page would waste one page of physical memory per stack page if the stack is directly mapped..

PaX USERCOPY is another feature from PaX which adds bounds checks to any data that is copied to and from the user-space. It modifies all the functions that copy data from user-space to verify that the data doesn't write past the stack to prevent overflows to heap. Depending on the architecture support for stack frame pointers, it can also prevent the writes past the current stack frame to prevent stack corruption attacks.

Hardware Based Techniques like *Memory Protection eXtensions* (MPX) [25] , like recently introduced by Intel, would allow software to specify pointer bounds with each memory allocation. Each de-reference of the pointer is then verified to be within these bounds by the hardware. An initial study by Oleksenko et. al [26] shows promising results as compared to software based bounds checking mechanism. However, the current iteration of the implementation has poor support and buggy implementation. Since the software needs to provide bounds information with each pointer, it requires considerable amount of invasive change in the source code to support MPX. The performance impact, though better than software based solutions, is about 50% as per the studies done by Oleksenko et. al.[26].

3.3.2 Heap Overflow

Heap based buffer overflows are similar to stack overflows in mechanism, however, due to manual memory management in heap region, there exist different strategies to prevent heap overflows. Heap overflow can overwrite important data structures like function pointers, object metadata or other objects on the heap.

Most objects are allocated on heap by Slab allocators like SLAB, SLUB and SLOB. These allocators store the metadata along with the objects in the kernel memory, placement of which can vary depending on the allocator. In most of the slab allocators, the metadata is stored along with the objects in adjacent memory regions for fast access and cleaner memory layout.

A common technique to exploit buffer overflows is by providing incorrect length values to a function that copies data from user-space to kernel-space memory (CVE-2013-6381, CVE-2012-2119, CVE-2012-2136). This can also happen if validation code in-correctly determines the size of data due to wrong calculations, poor assumptions or other means.

Similar to stack, heap overflows can also be used to inject code and be exploited by forcing the control flow to the address of injected code in the heap.

Mitigation Techniques

Name	Coverage	Bypassable	Cost
Stack Canaries by GCC	Return address on stack	yes, direct write using a pointer	˜0%
PaX strict `mprotect`	W $\oplus$ X	no	
`PAX_PAGEEXEC` / `PAX_SEGMEXEC`	code injection	no	
`GRKERNSEC_KSTACKOVERFLOW`	cross-page overflow	no	
`PAX_USERCOPY` / Hardened User-copy	all overflows	yes, non-slab allocations	

Table 3.5: Mitigation Techniques for Heap Overflow

PaX Code injection attacks can be prevented by W $\oplus$ X policy over the heap region. Strict `mprotect` from PaX prevents any region of the memory which has been written to be marked executable in future by an `mprotect` system call.

`PAX_USERCOPY` changes the behavior of functions that copy data to or from user-space to kernel space to check for object bounds on every copy operation. It adds checks for validity of kernel pointers to make sure that it points with-in address range of kernel memory, that it is not NULL, and that it doesn't point to a zero-length memory allocation. If the pointer points to a memory region managed by one of the Slab allocators, it also checks if the size of the data being copied is same the size of the object. If the pointer doesn't point to a slab allocated memory region, it checks that the data doesn't cross page or *compound page* [18] boundaries and doesn't span across pages that are a part of two separate allocations.

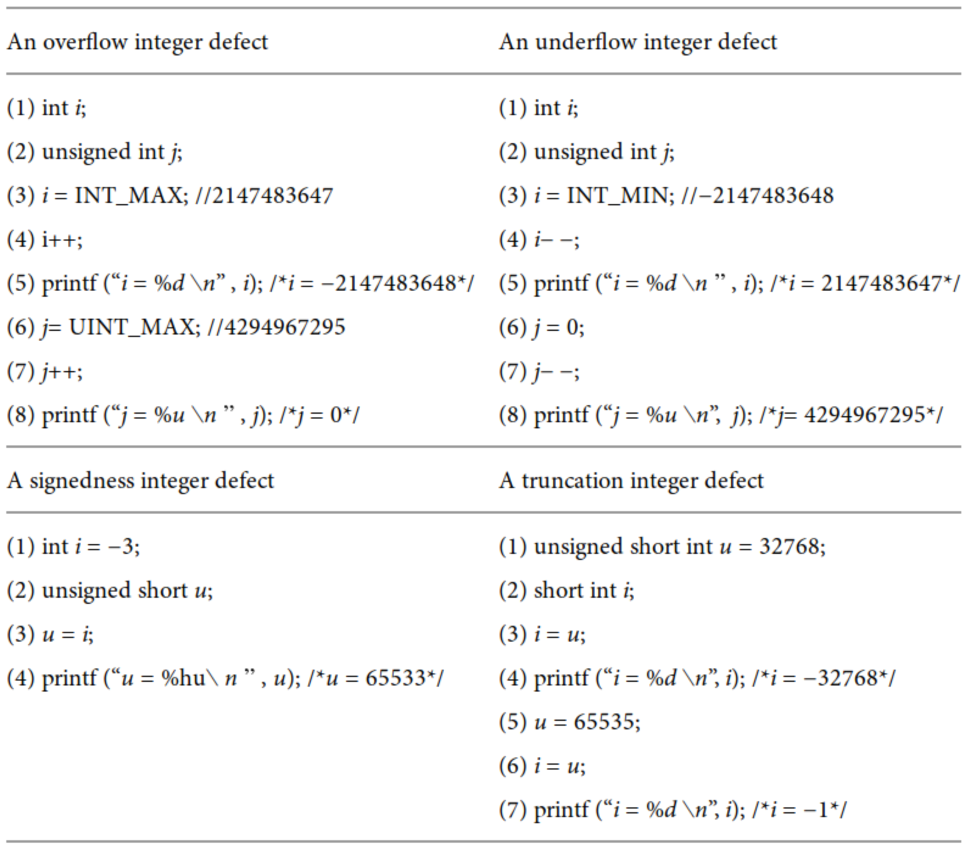
3.3.3 Integer defects

Integer overflows happen as a result of arithmetic operations that result in values larger than the allocated region. Often, this happens because registers in processors are fixed-length. They often lead to undefined behaviors in programs if not handled properly.

[18] Compound pages are a combination of one or more page that can be used as a single buffer.

According to Hui et. al. [1] there are four types of integer overflow related bugs that can happen, as explained in Figure 3.3:

- Integer overflow defect
- Integer underflow defect
- Integer signedness defect
- Integer truncation defect



An overflow integer defect	An underflow integer defect
(1) int *i*;	(1) int *i*;
(2) unsigned int *j*;	(2) unsigned int *j*;
(3) *i* = INT_MAX; //2147483647	(3) *i* = INT_MIN; //−2147483648
(4) i++;	(4) *i*− −;
(5) printf ("*i* = %*d* *n*" , *i*); /**i* = −2147483648*/	(5) printf ("*i* = %*d* *n* " , *i*); /**i* = 2147483647*/
(6) *j*= UINT_MAX; //4294967295	(6) *j* = 0;
(7) *j*++;	(7) *j*− −;
(8) printf ("*j* = %*u* *n* " , *j*); /**j* = 0*/	(8) printf ("*j* = %*u* *n*", *j*); /**j*= 4294967295*/
A signedness integer defect	A truncation integer defect
(1) int *i* = −3;	(1) unsigned short int *u* = 32768;
(2) unsigned short *u*;	(2) short int *i*;
(3) *u* = *i*;	(3) *i* = *u*;
(4) printf ("*u* = %hu\\ *n* " , *u*); /**u* = 65533*/	(4) printf ("*i* = %*d* *n*", *i*); /**i* = −32768*/
	(5) *u* = 65535;
	(6) *i* = *u*;
	(7) printf ("*i* = %*d* *n*", *i*); /**i* = −1*/

Figure 3.3: Types of Integer defects. Source: Metamorphic Testing Integer Overflow Faults of Mission Critical Program: A Case Study [1]

One common source of integer related defects is the compatibility layer for 32bit binaries to work on 64bit architectures. Integer is represented in hardware using fixed width registers which can store maximum up to `INT_MAX` in a signed integer and `INT_UMAX` in an unsigned integer. Integer overflows are undefined behavior according to C11 standard [Section 3.4.3, C11 Standard], because of which compilers are free to handle it in any way they seem fit.

There are 68 integer overflow vulnerabilities (including 11 reference counter overflows, which can lead to Use-After-Free bugs 3.2), 10 integer underflow vulnerabilities and 14 integer signedness vulnerabilities that were reported against Linux in past 10 years. Integer defects can lead to various different types of vulnerabilities like privilege escalation due to logic error (CVE-2011-2022). Overflow of size variables can lead to small buffers being allocated, causing an overflow later when the data is copied to the buffer (CVE-2014-9904, CVE-2012-6703). Overflow of variables during size calculation of data can lead to information disclosure (CVE-2011-2209, CVE-2011-2208).

Mitigation Techniques

Name	Coverage	Bypassable	Cost
`PAX_SIZE_OVERFLOW`	size of memory allocations	no	

Table 3.6: Mitigation Techniques for Integer defects

Because these vulnerabilities result due to the behavior of the hardware, it is hard to prevent all occurrences of these vulnerabilities. However, if it is known beforehand that a variable can overflow, GCC includes primitives [27] to perform addition, subtraction, multiplication operations with overflow checks.

PaX `PAX_SIZE_OVERFLOW` from the PaX patches is a GCC plugin which can detect overflows. It does so by using a double sized data structure to compute the output of expressions and compares that to the actual output to detect an overflow. However, it is not possible to put this check *everywhere* in the source, so, it detects *interesting* size variables in functions which can have security implications like buffer overflow due to wrong sized memory allocations. Variables that are intentionally allowed to overflow can be marked so that they are not checked for overflow.

Testing Oracles Integer defects and their effects on missions critical programs in C programming language was studied by Hui et. al [1]. The lack of *testing oracles* [19] makes the testing of integer defects hard. They use a technique called *mutation testing* [20] to replace (1) *int* with *char*, (2) *int* to *short int* and (3) *int* to *long int* along with replacing *signed* with *unsigned* data types to introduce mutation.

People have proposed solutions to detect integer defects in softwares also by training on signatures of previously known vulnerabilities. These solutions inference type information of the variables in binaries to pre-calculate the values that can be then compared against actual output, use the *status register* to check for overflows and *carry flag* or use some other test oracle to compare the results against. Wang et. al. [28] proposed *SoupInt*, which uses this information to check for memory allocations that are smaller than they should be and would probably result in a buffer overflow. They use static analysis techniques to track the variables that can potentially overflow to functions that perform memory allocations. They also go one step ahead of detection and generate patch by binary instrumentation to fix these flaws using existing error handlers.

3.3.4 Analysis

Bounds Check related vulnerabilities account for a total of 20% of all the vulnerabilities that are reported. Because C doesn't have any built-in support for bounds checking, it is hard to prevent these from happening altogether. Integer overflows can be detected (using options in GCC) but it would cost a lot of performance to enable it un-conditionally for all integer types.

Stack Overflow Detecting stack based overflows is possible using stack canaries but only after they have already happened and, in the best case, would result in a Denial of Service or OOPS. `PAX_USERCOPY` prevents overflows past the current stack frame preventing compromise of control flow but would still result in loosing return address at the bottom of the stack frame causing a crash. By virtually mapping stack (`CONFIG_VMAP_STACK`) without any need for it to be physically contiguous, it is now possible to detect cross-page

[19]Testing oracle is a theoretical machine that can predict the *correct* output of a program or expression for testing.

[20]Testing of programs which do not have any test oracle by mutating data types in pre-defined fashion to detect anomalous behavior and thus determine the presence of an integer defect.

overflows in Linux by adding guard pages. This feature was a part of grsecurity/PaX as `PAX_KSTACKOVERFLOW` before it was implemented in Linux. By exploiting an array indexing bug (19 of which were reported in past 10 years), it is possible to overwrite the return address bypassing canaries and all other protections surveyed.

Heap Overflow Heap based overflows are harder to protect because of the manual memory management. Often un-trusted length values result in more amount data to be copied from user-space resulting in corruption or overwrite of neighboring data structures or metadata. `PAX_USERCOPY` validates the length of all objects managed by Slab allocators preventing overflows of these objects. Features from `PAX_USERCOPY` are being ported to Linux to prevent these attacks and can be assumed to be a solved problem. Buffers not allocated through Slab allocators are still susceptible to overflows.

Integer Defects Integer defects occur due to fixed width of integers in hardware registers which allows values past a certain maximum (`INT_MAX` or `INT_UMAX`) to overflow and wrap around. `PAX_SIZE_OVERFLOW` can detect security sensitive integers, which define the length of memory allocations. Static analysis tools like SoupInt can detect integer overflow bugs by comparing output from computation in hardware with emulated calculation with double length data types in software.

3.4 Null Dereference

Pointers are variables that store the address of another variable or memory region. According to C11 standard, de-referencing a NULL pointer is undefined behavior. This means, there is no standard for what happens when a NULL pointer is dereferenced, which leaves this decision to compilers. Often compilers use this undefined behaviors to optimize the code and sometimes even remove checks for NULL pointers after they have been de-referenced (since they can't be NULL after being de-referenced.). Wang et. al. [29] studied the effect of undefined behavior in C and argue that they lead to tricky real world issues which can lead to problems in real world.

Usually, NULL (zero) pointers are considered invalid, in which case any code that de-references a NULL pointer in user-space causes a memory error, and the kernel will kill the process. But in kernel-space, zero is a technically valid pointer and it would point to the *zero page* in virtual memory. This page is mapped with no permission bits set, so any access to it will trap into kernel which will decide if the process should be allowed the access or not. Only processes with `CAP_SYS_RAWIO` are allowed to map to page zero.

`vm.mmap_min_addr_` is a `sysctl` [21] knob to set the minimum address that any process can `mmap` to, which is set to a non-zero value by default in most GNU/Linux distributions. To be able to exploit a NULL pointer de-reference, the first step would involve bypassing this protection to map adversary controlled code or data in the zero page.

Once, an adversary can add code or data to zero page, they can use a NULL pointer de-reference bug to use that. There were a total of 149 NULL pointer de-reference vulnerabilities that were found, which account for 13.4% of total. Without enough permissions, a NULL pointer de-reference results in an OOPS and hence denial of service.

3.4.1 Mitigation Techniques

Name	Coverage	Bypassable	Cost
`KERNEXEC`	complete	no	
`SMEP` / `PXN`	only function pointers	no	
`SMAP`	complete	yes, can be disabled by writing to a register	
`UDEREF`	complete	no	
Smatch	detect NULL de-reference bugs in source	n/a [22]	

Table 3.7: Mitigation Techniques for NULL pointer Dereferences.

To prevent a NULL pointer de-reference vulnerability from being exploited, one can go about two different ways:

1. Prevent any user-space application to map anything to page zero

2. Prevent any NULL pointer from being de-referenced in kernel

[21] `sysctl` is tool used to examine and configure parameters for Kernel. Configuration parameters are usually exposed through `procfs` or `configfs` in the virtual file system.

[23] Smatch is a static analysis tool and cannot guarantee detecting all vulnerabilities of any kind.

Page zero mapping

While `vm.mmap_min_addr` prevents any process from mapping to zero page by default unless the process has `CAP_SYS_RAWIO`, in past it could be bypassed by compromising a *setuid* [23] binary with required capability. Since it is allowed to map to page zero with certain capabilities, a bug in user-space program can be exploited to map contents in page zero. `PER_CLEAR_ON_SETID` is a list of security critical flags that are cleared when a setuid binary is executed. Since Linux v2.6.31, this list of flags includes `MMAP_PAGE_ZERO`, which would make it impossible to use this specific technique to map attacker controlled code to page zero by exploiting a setuid process.

Page zero access

It is hard to prevent Kernel from de-referencing a NULL pointer as they are valid pointers, but it is possible to prevent Kernel from de-referencing a user-space pointer[24] when it is expecting a Kernel space pointer. Supervisor Mode Exec Prevention (SMEP), added to *Ivy Bridge* [25] microarchitecture by Intel, prevents Kernel from fetching instructions to execute from user-space (Privilege Level 0 in Linux). In recent ARM architectures, a similar feature called Privileged eXecute Never (PXN) was added to prevent kernel mode access of user-space memory.. PaX's `KERNEXEC` uses memory segmentation in older architectures which do not support SMEP or PXN to implement similar semantics. This can prevent de-referencing dangling pointers controlled by attackers.

Supervisor Mode Access Prevention (SMAP), added to *Broadwell* [26] microarchitecture by Intel, prevents Kernel from any access to user-space memory. However, unlike SMEP, SMAP cannot be enabled un-conditionally since there are legitimate use cases when kernel needs access to user-space memory. For this reason, SMAP can be enabled and disabled by writing to `CR4` register. PaX's `UDEREF` , similar to `KERNEXEC`, uses memory segmentation in older architectures to prevent Kernel from accessing user-space memory. It patches all user-space accessing functions to change segmentation related

[23]setuid is a mechanism in Linux where any user executing a binary with setuid bit set will execute the process with permissions of the owner.

[24]NULL pointer would point to user-space when Kernel is executing in the context of a user-space process like in system calls.

[25]Introduced with 3rd generation of Intel Core i CPUs

[26]Introduced with 5th generation of Intel Core i CPUs

registers to temporarily allow the access.

Static Analysis

Static analysis tools like Smatch [30] and Coccinelle [31] can be used to find potential NULL pointer vulnerabilities in the source code by testing at development time.

3.4.2 Analysis

NULL Dereference vulnerabilities account for 13.44% of the total vulnerabilities reported, highest of all other types. As discussed before, NULL dereference vulnerabilities can be exploited if an attacker can map to page zero. `PAX_KERNEXEC` and `PAX_UDEREF` can potentially prevent these attacks by emulating `SMAP`, `SMEP` / `PXN` in software, but their reliance on memory segmentation and availability to only paying customers makes them less-likely to be adopted in future. Static analysis tools like Smatch and Coccinelle however can be used to find NULL dereference bugs by testing at development time. Since `SMAP` can still be disabled by writing to `CR4` register (possibly using a ROP attack), NULL pointer de-references can still be a security threat, even with modern hardware support.

3.5 Format String Vulnerability

String formatting is a common technique in programming languages to use template strings with placeholders, which are later replaced by values to generate desired strings. In C, functions `printf` and `fprint` are common examples which accept these template strings and variables that represent the value to placeholders and return a *formatted string.*

```
char *screen = "screen";
int number = 10;
printf("Print the number %d to the %s.", number, screen);
```

In the above example , `printf` is a *format function*, `%d` is *format string parameter* which defines the *type* that the variable `number` is, and string "`Print the number %d to the %s.`"

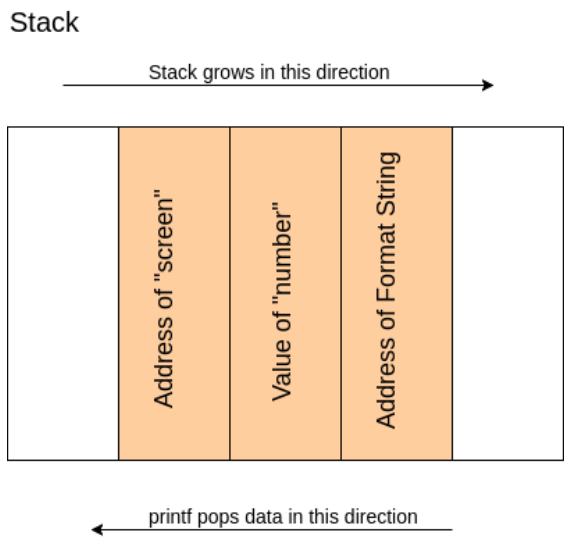


Figure 3.4: Stack frame for a format string function.

is *format string*. Fig 3.4 shows their arrangement in the stack. `printf` pops the values from the stack depending on the number of format string parameters in the format string. A malicious input for the value of variable `screen` in the above example like "`screen %x`" will make `printf` pop the next value from the stack which could potentially be some sensitive information.

Format string parameter defines the type of input variable, like `%d` expects and `int` or integer type. Format string parameters can be used to read from or write to data in the stack, for example:

- `%x`: To read bytes from memory
- `%s`: To read character strings from memory
- `%n`: To write an integer in the memory
- `%p`: To print the address a pointer points to

A bug where an adversary can control the format string parameter in a format string, can leak arbitrary memory regions (using `%x`) or write arbitrary memory regions (using `%n`). In CVE-2013-2852 [32] an input parameter to the Broadcom driver module (b43), which is controlled by user-space, is used in a error message without proper validations. When the error message is printed or logged, the format string vulnerability in the error message can cause arbitrary memory read/write. A total of only 3 format-string vulnerabilities were found in our analysis.

3.5.1 Mitigation Techniques

Compiler Instrumentation

GNU Compiler Collection or GCC can detect calls to `printf` which can potentially lead to a format string vulnerability. When option `-Wformat-security` is enabled, GCC warns about calls to `printf` without a literal string (i.e. input is a variable pointing to format string) and there are no arguments to format string. If an adversary can control this variable, they can read data from memory. Due to it's security implications, all uses of `%n` were removed from Linux source in the year 2014 [33] and any future use will be ignored during compile time.

Strict use of format strings

Information leak from the format string parameters like `%p`, which can reveal kernel pointers, can only be prevented with careful use of these parameters. A new format string specifier `%pK` was introduced in 2011 [34] to hide the kernel pointers from being leaked in the logs or `/proc` filesystem. Depending on the value of `/proc/sys/kernel/kptr_restrict` sysctl's value, `%pK` will have the following behavior:

- `kptr_restrict = 0` : No deviation from the standard `%p` behavior
- `kptr_restrict = 1` : If the current user doesn't have `CAP_SYSLOG` capability, all kernel pointers will be printed as all 0's.
- `kptr_restrict = 2`: All the kernel pointers are printed as 0's, regarless of the privileges.

3.5.2 Analysis

While format string vulnerabilities are not very common, there is no defense mechanism that can prevent their exploitation if they somehow find their way into the kernel. Removal of `%n` would make sure that format strings can't be used to write to memory. While `kptr_restrict` can prevent information leaks, it is an opt-in method, which means that it works only if all uses of `%p` are carefully removed from Linux. For now, it is a matter of convention to not use potentially dangerous format string parameters, which can lead to information disclosure. There is nothing preventing the use of `%x`, which bypasses any protection provided by `kptr_restrict`.

3.6 Missing permission check

Missing or wrong permission check are another class of programming errors which is very commonly seen in Linux. There are over 133 vulnerabilities reported in Linux which were caused due to missing or wrong permission checks. Depending on operation and context, a missing or wrong permission check can lead to variety of attacks.

Poor handling or namespaces can lead to privilege escalations in containers which are often isolated using namespaces in Linux (CVE-2016-1576). Bugs in network stack can allow remote attackers to bypass firewall or other network restrictions (CVE-2012-4444). Not clearing permissions when spawning off a low privilege process can lead to it having un-intended permissions which it may not be prepared to handle (CVE-2009-1895). Missing checks for file permissions can allow arbitrary changes to append-only files (CVE-2010-2066).

3.6.1 Mitigation Techniques

Missing authorization step in the workflow makes it nearly impossible to control the access of resource from users that aren't authorized to access it *by design*. This makes missing permission checks vulnerability impossible to mitigate.

3.7 Race conditions

Race conditions are vulnerabilities related to poor handling of critical sections in multi-threaded or multi-process software. Usually, this occurs when two processes running in parallel change a shared data structure without proper co-ordination among themselves. Consider the example below:

```
if (x==10) {  // Time of check
    y = x*x;  // Time of use
}
```

Here, if `x` is a shared variable among more than one process, it is possible that the value of x could be different when its value is checked and when it is used in next line. This is commonly called TOCTTOU (Time Of Check To Time of Use) vulnerabilities, and this is a simple example of a possible race condition. The part of code operating on a shared variable is termed as *critical section* and it's use is often coordinated using locks to prevent a race condition.

There are 56 race condition vulnerabilities that were reported against Linux in past 10 years. Linux gained support for Simultaneous Multi-Processing (SMP) in v2.0, which was released in May, 1996 [35] . However, a preemptive system [27] can suspend a thread when it is executing in a critical section and can cause race conditions even on systems with no support for SMP. In Linux v.2.6, support for preemption for processes running kernel code was added.

Race conditions can result in various types of attacks depending on the shared state that was left un-checked. In CVE-2009-1527, a wrong type of lock was used in a `ptrace` system call when accessing the state of a process, which could lead to local privilege escalation when tracing a *setuid* application. CVE-2014-9710 allowed users to gain privileges because of a race condition which left access control fields empty for some time which could be leveraged to bypass intended privilege checks in Btrfs filesystem.

[27]In a preemptive system, operating system can take control back from processes without their will. This is done to make sure a single process doesn't starve other processes.

3.7.1 Mitigation Techniques

Race conditions are hard to detect and mitigate. Because of their nature, it is impossible to detect them using usual testing methods. No known mitigations exist today that can prevent exploitation of an existing race condition in Linux.

3.8 Denial of Service Vulnerabilities

The three basic pillars on which security of any system is defined are *confidentiality*, *integrity* and *availability*. There are several reasons as to why perfect availability is impossible to achieve. We focus only on factors that are controlled by software and ignore the failures caused by external factors like hardware or external infrastructure. Availability is a property of a software system, that depending on the system being observed, could mean different things. For example, in a web service, availability implies that the endpoint always accepts requests and returns a valid response. This availability is representation of the service as a whole, there could be multiple failures of individual services that comprise the web service, but due to smart replication and request routing, the web service endpoint is always functional.

In the context of Linux kernel, availability is defined with the responsiveness of processes. Software bugs and crashes affect the availability of the system. These crashes are often due to internal bugs, that can be triggered from an un-trusted outside input. Different inputs invoke different code paths inside kernel and a bug in any of those code paths could lead to a software crash.

3.8.1 Mitigation Techniqes

Most denial of system vulnerabilities are simple software bugs that are fixed quite easily when found, however, the process of finding these bugs might be more challenging. It is hard to find a pattern in denial of service vulnerabilities, simply due to the large number and types. *Fuzzing* is a software testing methodology where instead of manually curating testing data, which could potentially be a large amount given the combinations of environments, configuration and inputs, a tool is made to generate testing data. These tools are generally called as *fuzzers*.

Fuzzers

Fuzzers are provided with basic templates of the API to be tested and are allowed to run against a given system. The response for each request is recorded and validated and any inconsistencies in the expected output are reported as bugs. Linux expose a vast variety of system calls, it's API, which can be tested using these fuzzers.

Trinity [36] and **Syzkaller** [37] are Linux system call fuzzers. Both of them use system call templates for argument domain specification. Syzkaller also uses code coverage information for guiding the fuzzing.

While fuzzers can be helpful to find out some of the bugs, their coverage is very limited. They do not provide any guarantee to find all bugs and can often miss complicated bugs.

3.9 Miscellaneous

While the majority of vulnerabilities are based on repeating patterns, some aren't. However, these vulnerabilities can still be grouped by the result of those vulnerability. For example, the vulnerabilities that all result in an infinitely running loop, thus affecting the availability of a system, are grouped together. These vulnerabilities can be simple programming errors that cause un-expected side effects. Some common ones include faulty cryptographic implementations (CVE-2009-3238, CVE-2007-2451, CVE-2014-7284), Information leaks due to various reasons that aren't a part of any of the previously mentioned classes (CVE-2011-0710, CVE-2010-4565), memory leaks etc.

A large part of these vulnerabilities result in DoS i.e. affect the availability of the system. Linux has several essential processes running in kernel mode, which when fail, push the kernel in an un-reliable state. Several of the bugs that can generally be classified into more specific classes of bugs, like NULL Dereference 3.4, Buffer Overflow 3.3 etc, can also cause services to crash. Even simple bugs like a missing error handler can cause kernel crashes. Compute intensive code paths, which can be invoked from an un-trusted input, can make the system un-available to other requests, even without a failure.

CVE-2011-1768 allowed remote attackers to cause an OOPS by sending a packet while the `ip_gre` module is being loaded. CVE-2009-0747 allowed local users to cause CPU consumption and error-message flood by trying to mount a crafted EXT4 filesystem.

CVE-2015-5307 allowed guest OS users to hang host OS by raising many Alignment Check exceptions.

CVE-2016-4440 allowed guest OS users to access APIC MSR on host OS due to poor handling of the APICv on/off state. CVE-2013-0311, caused by a bug in translation of cross-region descriptor, can cause guest OS to attain host OS privileges by leveraging KVM guest OS privileges. CVE-2015-0274, caused by a bug in XFS filesystem implementation, uses a wrong size value during attribute replacement allowing local users to cause a denial of service or gain privileges.

Some of these vulnerabilities can be sorted into logical groups, though, these groups don't particularly have any good proactive defenses known yet. These groups include:

- **Infinite Loop**: These vulnerabilities can cause loops to run indefinitely making a system un-responsive and un-available to perform actual work.
- **Memory Leaks**: Memory leaks can often result in huge memory consumption and un-availability of memory, thus starving them out.
- **Divide-by-zero**: Dividing any number by zero is an undefined operation and depending on the situation, it can result in a system crash. These vulnerabilities often happen when the variable in the denominator becomes zero unexpectedly.
- **Cryptography**: Several bugs in cryptographic protocol implementations can result in exploits as cryptography is often the basis of security in a lot of protocols.
- **Length Calculation Bugs**: These bugs are often caused by mis-calculating the size or length of a data or wrong arithmetic. It can often result in either too big or too small memory region being allocated and sometimes may or may-not result in buffer-overflow or buffer-overread.

3.9.1 Mitigation Techniques

Since these vulnerabilities don't show any common pattern in their type, it is hard to actually prevent them from being exploited. Proactive defenses are based on the idea of predictable behavior of vulnerabilities, which can't be found in these vulnerabilities.

Most vulnerabilities that result in denial of service are simple software bugs that are fixed quite easily when found, however, the process of finding these bugs is more challenging. *Fuzzing* is a software testing methodology where instead of manually curating testing data, which could potentially be a large amount given the combinations of environments, configuration and inputs, a tool is made to generate testing data. These tools are generally called as *fuzzers.*

Fuzzers

Fuzzers are provided with basic templates of the API to be tested and are allowed to run against a given system. The response for each request is recorded and validated and any inconsistencies in the expected output are reported as bugs. Linux expose a vast variety of system calls, it's API, which can be tested using these fuzzers.

Trinity [36] and **Syzkaller** [37] are Linux system call fuzzers. Both of them use system call templates for argument domain specification. Syzkaller also uses code coverage information for guiding the fuzzing.

While fuzzers can be helpful to find out some of the bugs, their coverage is very limited. They do not provide any guarantee to find all bugs and can often miss complicated bugs.

Chapter 4: ROP Attacks and Protections

4.1 Return-Oriented Programming attacks

Return-oriented-programming (ROP) or code-reuse attacks are based on a combination of vulnerabilities and have been shown to bypass modern defenses for various types of vulnerabilities discussed in Chapter 3. Given that these are attacks and not vulnerabilities, they aren't assigned CVE numbers to quantify their frequency. Hence, they are studied separately from other types of vulnerabilities.

ROP attacks hijack the control flow of a program by exploiting a memory corruption vulnerability and then using existing code in memory to perform un-intended operations [38, 39, 40] . Previously called return-to-libc [41], these attacks use the existing code fragments in shared libraries or program binary to implement arbitrary program logic. As these attacks got more sophisticated, their dependence on shared libraries reduced and code fragments could be generated using JIT compiler and within the program binary. Code fragments, also called *gadgets*, which end in a `ret` statement can be chained together to perform turing-commplete [1] [40] set of operations. There are typically three steps in a successful ROP attack:

1. Exploit a memory corruption vulnerability to change a return address or function pointer
2. Jump to the user-controlled code using the above vulnerability and to the next gadget from their in a chain
3. Return to the correct location that was supposed to run in step 1

The first and most important requirement to launch a ROP attack is a memory corruption vulnerability. If there is one thing that analysis of past 10 years of vulnerabilities has revealed, it is that they are available in plenty. By exploiting a spatial

[1]Turing-completeness implies that any arbitrary operation can be performed using any random combination of system calls.

memory vulnerability, as discussed in Bounds Check vulnerabilities, section 3.3, it is possible to overwrite the return address of a function to jump to an arbitrary location. A NULL pointer deference, discussed in section 3.4, can be used to redirect program flow to page zero. Use-after-free vulnerabilities, discussed in section 3.2, allow hijacking control flow by compromising function pointer tables. To redirect the control flow, it is also important to know the correct address of gadgets in memory which may not always be trivial as Linux randomizes the base address of various memory segments (ASLR).

In second step, the control jumps from the location of memory corruption bug to a user controlled location. It is important that this location be mapped in virtual memory area and be accessible at the call site. If this condition is not met, any access to un-mapped region would crash the kernel or cause an OOPS leading to an un-successful attack.

Finally, to successfully evade detection, the attack should be able to return the control back to the program for it proceed under compromised conditions.

Information Leakage Research community has also shown that some defenses like ASLR Address Space Layout Randomization, mentioned later in defenses, can be broken using information obtained from hardware side-channel attacks which are based on micro-architectural features. Gruss et. al. [42] use software pre-fetching instructions to obtain sensitive information from various caches in x86 system, Hund et. al. [43] used timing channel attacks against double page faults to discover valid memory locations, Jang et. al. [44] use Intel Transactional Synchronization Extension (TSX) from recent Intel processors to perform timing channel using similar methods. Information revealed from vulnerabilities in Linux can also be used to weaken the guarantees offered by ASLR (CVE-2013-0914, CVE-2016-3672, 2015-8575, 2015-8569, 2014-9585, 2014-9419). Oikonomopoulos et. al. [45] later showed that it is possible to determine the location of code fragments without complicated side-channel attacks and instead relying on allocation oracles, which repeatedly allocated chunks of memory to determine the holes in address-space where the possible code-targets could be.

4.1.1 Mitigation Techniques

To systematize the study of all the mitigation techniques, they are grouped based on the step that they prevent in above mentioned ROP attack. We skip over mitigation techniques for memory corruption vulnerabilities as they have been discussed in detail in sections 3.2, 3.3, and 3.4.

Prevention of Step 1

In order to execute **step 1**, an attacker requires the address of gadgets in memory to jump to. Memory address for a particular instruction in memory can be calculated by adding it's offset from the start of the program to the base address of the memory segment where the program is mapped. Address Space Layout Randomization (ASLR) is a technique to randomize the base address of different section in virtual memory area. Depending on the implementation, each chosen section could have some entropy in it's base address making it hard to guess its value reliably. There are 256 and 512 random positions possible for 32bit and 64bit x86 Linux in current implementation of ASLR.

GR Security `GRKERNSEC_RANDSTRUCT` is a GCC plugin from grsecurity which randomizes the layout of all the structures comprised of function pointers (a.k.a *ops* structures) to make it harder to overwrite them using spatial memory errors. Because it has a high performance overhead, another option `GRKERNSEC_RANDSTRUCT_PERFORMANCE` is provided which takes into account the size of cache-line to randomize structures with reduced security guarantees. This plugin was ported over to Linux in v4.13 but only randomizes structures that are explicitly marked with `__randomize_layout` [46].

Address Space Layout Randomization 32bit implementations of ASLR are vulnerable to de-anonymization attacks by means of brute force ([47], [48]). Randomizing `mmap` , each shared library, code and data segments separately can increase the difficulty of such attacks. Kil et. al. [49] proposed Address Space Layout Permutation (*ASLP*) to randomize base address of stack, heap, shared libraries and executable to make it harder to guess the address of code in memory, such techniques are called fine grained ASLR. Bigelow et. al. [50] proposed re-randomization of memory space as soon as the program gives an output to render the data revealed from information leaks useless. Lu et. al.

[51] encode the code-pointers when they are treated as data so that they are of no use when leaked to determine the address of code region.

Isolation A recent work by Gruss et. al. [52], *KAISER*, makes Kernel Address Space Layout Randomization (*KASLR*) more promising by proposing stronger isolation between kernel-space and user-space memory in order to defend against side-channel attacks. They propose removing mapping of kernel in userspace, with exception of some portions which would allow context switch to kernel-space memory. ARM processors have two separate page tables for mapping user-space and kernel-space which allows stronger isolation of user-space and kernel-space.

Return-less kernel Li et. al. [39] proposed removing all the `ret` instructions using binary instrumentation techniques as an effort to remove all the gadgets that can be used in a ROP attack, but Checkoway et. al. [40] and Bletsch et. al. [53] showed that it is possible to perform ROP attacks even without `ret` instructions by using `jmp` instructions which also allow jumping to a region in memory like `ret`. They termed it as jump-oriented programming (JOP) attacks.

Protected Pointers Cowan et. al. proposed *PointGuard* [54] which prevents pointers from leaking by encrypting them in the memory and decrypting them only before dereferencing. One single encryption key is used to encrypt all addresses and is stored in a register. It suffered from various problems like use of single key (which can be exposed with an information leak) for all encryption and incompatibility with existing source and binaries. Bhatkar et. al. [55] introduced *Data Space Randomization* which would extend the idea of PointGuard to encrypt all data, pointers as well as other variables, using separate keys and instrument binary to support it. While it promises better security than PointGuard due to use of different encryption keys, it suffers from other problems like binary and source incompatibility, poor performance (15% overhead), incompatibility with unmodified libraries etc.

Prevention of Step 2

To prevent jumping to an attacker controlled address, **step2** of attack, a technique called Control Flow Integrity (CFI) was proposed by by Abadi et. al [56] in 2005. It involves static analysis of the source code to generate a call flow graph (CFG) and binary instrumentation to enforce program flow to strictly adhere to CFG. An indirect jump from the point of memory corruption bug to attacker controlled code is called *forward edge* attack since it creates a forward edge in CFG. Ligatti et. al. [57], Tice et. al. [58] and many others further improved forward edge CFI to achieve better performance without compromising the security. A new class of CFI techniques called coarse-grained CFI relaxed strict restrictions of original (fine grained) CFI to allow for more valid control flow jumps like ROPecker by Cheng et. al. [59] and kBouncer [60] by Pappas et. al. But they were soon shown to be in-effective [61, 62] against slight modifications in attack strategy.

GRSecurity `RAP` from grsecurity is a GCC plugin which implements fine-grained CFI in Linux. It infers type information from functions and function pointers and checks a hash value of the function type on pointer dereference to make sure that the pointer points to the correct place. However, in existing Linux code, several pointers have different types than the functions that they point to which makes it a lot of work to change and then maintain(enforce it in future).

Hardware Techniques `SMEP`, discussed in section 3.4, can prevent certain classes of ROP attacks called *ret2user* attacks which redirect the control flow of Linux program to user-space for gadgets. Gruss et. al. [42] recently showed that protections that prevent access to user-space like `SMAP`, `SMEP`, `PXN`, `PAX_SEGMEXEC`, `PAX_PAGEEXEC` can be bypassed by using `pysmap`, a mapping of entire user-space memory in kernel-space in Linux, to bypass all the protections that prevent user-space access from the Kernel.

Prevention of Step 3

To execute **step3** in the ROP attack chain, an attacker should be able to return back to the site of memory corruption bug in order resume the normal flow of the program (in the compromised state.) *Stack Shield* [63] is a tool which uses *shadow stacks* to prevent indirect control transfers from stack overflow attacks that overwrite return addresses. Return addresses are pushed on to a different stack (called shadow stacks), which is protected using other mechanisms, and compared against the return address when the function returns to make sure it wasn't modified by an attacker. Studies by Carlini et. al. [64] show that shadow stacks are effective in preventing arbitrary code execution. Dang et. al. [65] studied the performance overheads of different shadow stack implementations to see if they are practical and propose lightweight techniques to reduce performance impact of existing shadow stack implementations. Intel recently introduced Control-flow Enforcement Technology (CET) [66] which will be a part of future generation of Intel processors and will provide hardware support for shadow stacks in hardware which will increase the performance of such solutions.

GRSecurity `RAP`, from grsecurity, uses a technique similar to what Cowan et. al. [54] and Bhatkar et. al. [55] proposed previously to prevent step 1 of ROP attack. It encrypts the return address on function call and stores the encrypted return address and the encryption key in two registers, when the function returns, it re-encrypts the address being returned-to and halts execution if both of them don't match. The choice of key is not limited to a single value for every process, long running threads in Linux like the scheduler can use a new key on each iteration.

4.1.2 Analysis

Code-reuse attacks or **ROP** attacks are one of the biggest challenges today. ASLR based protection schemes can be bypassed with data obtained from information leak vulnerabilities, over 200 (includes un-initialized data, buffer over-reads, other information leaks) of the which were reported in past 10 years. `RAP` from grsecurity/PaX encrypts return addresses and enforces forward-edge CFI by allowing pointers to point to only valid functions that are identified by their unique hash. However, since it uses encryption to prevent return-addresses from being corrupted, it is susceptible to information leaks which can reveal the encryption keys. Forward-edge CFI in `RAP` also depends on the fact that function pointers can only point to specific function types which is not enforced in C. This would require a lot of change in Linux source code to function correctly.

Research papers that prevent pointer corruption by encrypting them in memory suffer from a high overhead and binary-incompatibility. Coarse grained CFI techniques improved the performance but can be easily bypassed with slight modification in the attack strategy. Intel CET will provide support for shadow stacks in hardware for better performance of CFI techniques in future, but would leave others without latest hardware susceptible to ROP attacks. If these techniques will be able to offer complete remediation against ROP, even with support from hardware, still remains a question. ASLR and related techniques, based on randomization of addresses have shown to be weak, but research community hasn't given up on it yet. Newer works promise better implementation and ideas that can possible render information leaks useless.

One point to note here is that some solutions to prevent ROP by leveraging virtualization techniques have also been proposed. They however haven't been covered in this survey, which studies the current security capabilities of Linux kernel itself, as it requires an outside agent to prevent these attacks.

Chapter 5: Discussion and Conclusion

5.1 Discussions

Vulnerability Class	**Defenses**	**Research**
Un-initialized Data	- Defense exists for different memory allocators but not general case - Limited coverage, does not prevent against un-marked fields - Poor performance	- Better coverage - Better performance available - Solved with acceptable performance
Use-after-free	- Defenses available for special cases like reference counters - Randomization techniques makes attacks harder	- Full coverage - Solved with acceptable performance
Bounds Check/ Stack Overflow	- Detection tools available to cover most cases except direct memory overwrite	- Studies based on Hardware extensions show promising results but poor performance
Bounds Check/ Heap Overflow	- Code injection attacks in heap are solved - Overflows crossing page boundaries can be prevented - Unsolved for non-slab allocators	-
Bounds Check/ Integer Defects	- Size overflow detection possible with limited coverage and explicit overflow check	- Testing theories to detect overflows based on signatures exist
Null Derference	- Solved with proper configuration - Hardware features (SMAP, SMEP) prevent exploitation	-
Format String	- Pointer leaks can be prevented with configuration in some cases - Compile time checks detection possible with limited coverage	- Not available
Missing Perm. Check	- Impossible to enforce permissions without authorization step	- Not available
Race Conditions	- Not available	- Not available
Miscelleanous	- Fuzzing tools can find bugs leading to crashes - No proactive defenses available as these lack patterns	- Not available

Table 5.1: Summary of Current State of Defense Techniques

In Chapter 3 and Chapter 4, we discussed different types of vulnerabilities, their exploitation methodologies and finally their defenses. Apart from some classes like denial of service, missing permission checks etc. our survey reveals that there exists solutions for most range of vulnerabilities that are commonly found in Linux with varying amount of coverage and performance. Table 5.1 provides a summary of all the defenses discussed in Chapter 3.

However, it is very difficult to discretely distinguish these vulnerabilities into solved and unsolved groups due to several reasons. The most ideal definition of a solved problem would be a defense mechanism that exists in Mainline Linux kernel and which is enabled by default. Even though the defenses may exist in Linux, their widespread use is seen only if they are enabled by default. There are several reasons for why defenses are not enabled by default, stability, compatibility with legacy software and performance are the top three reasons.

A general feature in Linux is initially disabled by default when introduced in the Mainline. Various distributions then configure their kernels to a desired state and sometimes enable these features depending on the requirements of their users. Slowly and gradually after these features have been tested with a subset of users, these features are then enabled by default in Mainline kernel with feedback from major distributions. This entire cycle takes a few years to complete. Proactive defenses are often considered as features, as compared to fixes for a specific CVE.

From here, there are several less-than-ideal states, that these defense mechanisms can be in. Depending on the preference, stability and performance can be traded off for better security. Various solutions exist that can be improved in terms of performance overhead, stability and support for existing applications, etc.

In our survey, we found that most vulnerabilities have a defense mechanism that works. Some of them are implemented by PaX and GRSecurity patches, others have been added to Linux over the past decade. However, the performance impact of several of the defenses offered by PaX and GRSecurity patches leave a lot more to be desired.

The quantitative study of the vulnerabilities draws a picture about the trends in vulnerabilities over the years. According to Figure 1.1, the total number of vulnerabilities have been on rise in past decade, with occasional lows. Figure 1.2 shows the distribution of CVSS for vulnerabilities that we analyzed. It shows that high severity issues, while not un-common, are rare.

5.2 Conclusions

In conclusion, the number of vulnerabilities in Linux has been on rise effectively in past 10 years. However, the survey of defenses added to Linux and other options that exist outside of Mainline Linux, points to an improved overall state of security. Table `table:summar-of-all-defenses` consists of an summary of solutions for various categories and open gaps for future research work and areas requiring performance improvements. It is impossible to make claims about the net-improvement in security of Linux *quantitatively*, but theoretically, there are solutions for most of the classes of vulnerabilities in Linux. Some of these defenses come at a cost of poor performance, but considering the features that have already been added to Linux, it can be safely assumed that with significant engineering efforts, these can be incorporated in Linux to make it even more resistant.

5.3 Future Work

Now that we know about the relative occurrences of different classes of vulnerabilities, a logical extension of this work is to map the vulnerabilities to different parts of Linux. Linux has a huge code base and it would be interesting to see which parts of Linux have higher frequency of security vulnerabilities. In a typical Linux installation, not all the parts are useful since Linux is comprised of a large number of drivers for specific devices. By stripping off less or un-used parts of code in Linux, which have high number of vulnerabilities, Linux can be made more secure.

Bibliography

[1] Zhanwei Hui, Song Huang, Zhengping Ren, and Yi Yao. Metamorphic Testing Integer Overflow Faults of Mission Critical Program : A Case Study. 2013, 2013.

[2] Gnu general public license v2.0 - gnu project - free software foundation. `https://www.gnu.org/licenses/gpl-2.0.html`. (Accessed on 09/02/2017).

[3] Distrowatch.com: Put the fun back into computing. use linux, bsd. `http://distrowatch.com/search.php?ostype=Linux#simple`. (Accessed on 09/02/2017).

[4] Cve - common vulnerabilities and exposures (cve). `https://cve.mitre.org/`. (Accessed on 09/02/2017).

[5] The mitre corporation. `https://www.mitre.org/`. (Accessed on 09/02/2017).

[6] Rfc 2828 - internet security glossary. `https://tools.ietf.org/html/rfc2828`. (Accessed on 12/10/2017).

[7] Homepage of pax. `https://pax.grsecurity.net/`. (Accessed on 09/02/2017).

[8] Cve - cve list master copy. `https://cve.mitre.org/cve/cve.html`. (Accessed on 09/12/2017).

[9] Nvd - cvss. `https://nvd.nist.gov/vuln-metrics/cvss`. (Accessed on 09/09/2017).

[10] Cve security vulnerability database. security vulnerabilities, exploits, references and more. `https://www.cvedetails.com/`. (Accessed on 09/02/2017).

[11] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding Your Garbage : Reducing Data Lifetime Through Secure Deallocation. pages 331–346, 2005.

[12] S. Peiró, M. Muñoz, M. Masmano, and A. Crespo. Detecting stack Based Kernel Information Leaks. *Advances in Intelligent Systems and Computing*, 299:321–331, 2014.

[13] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. UniSan : Proactive Kernel Memory Initialization to Eliminate Data Leakages. *Ccs*, pages 920–932, 2016.

[14] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. SafeInit : Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. (March), 2017.

[15] Cve - cve-2016-0728. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2016-0728`. (Accessed on 09/03/2017).

[16] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 414–425, New York, NY, USA, 2015. ACM.

[17] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. TaintTrace : Efficient Flow Tracing with Dynamic Binary Rewriting Singapore-MIT Alliance. pages 7–12, 2006.

[18] Emery D Berger and Benjamin G Zorn. DieHard : Efficient Probabilistic Memory Safety. 2007.

[19] Santosh Nagarakatte and Milo M K Martin. CETS : Compiler-Enforced Temporal Safety for C. pages 1–10, 2010.

[20] Thurston H Y Dang, Petros Maniatis, Google Brain, David Wagner, and David Wagner. Oscar : A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers This paper is included in the Proceedings of the. 2017.

[21] Gene Novark and Emery D Berger. DieHarder : Securing the Heap. 2010.

[22] Yu Ding, Tao Wei, Tielei Wang, Zhenkai Liang, and Wei Zou. Heap Taichi: Exploiting Memory Allocation Granularity in Heap-Spraying Attacks.

[23] Github - google/syzkaller: syzkaller is an unsupervised, coverage-guided linux system call fuzzer. `https://github.com/google/syzkaller`. (Accessed on 09/12/2017).

[24] Gcc documentation. `https://gcc.gnu.org/onlinedocs/gcc-7.2.0/gcc/Instrumentation-Options.html#index-fstack-protector`. (Accessed on 09/05/2017).

[25] Introduction to intel memory protection extensions — intel software. `https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions`. (Accessed on 09/05/2017).

[26] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: An empirical study of intel MPX and software-based bounds checking approaches. *CoRR*, abs/1702.00719, 2017.

[27] Using the gnu compiler collection (gcc): Integer overflow builtins. `https://gcc.gnu.org/onlinedocs/gcc/Integer-Overflow-Builtins.html`. (Accessed on 09/06/2017).

[28] Tielei Wang, Chengyu Song, and Wenke Lee. Diagnosis and Emergency Patch Generation for Integer Overflow Exploits. pages 255–275, 2014.

[29] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M Frans Kaashoek. Undefined Behavior : What Happened to My Code ? . pages 4–9, 2012.

[30] Smatch the source matcher. `http://smatch.sourceforge.net/`. (Accessed on 09/04/2017).

[31] Coccinelle: A program matching and transformation tool for systems code. `http://coccinelle.lip6.fr/`. (Accessed on 09/04/2017).

[32] Cve - cve-2013-2852. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2852`. (Accessed on 09/02/2017).

[33] Linux kernel source tree. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=708d96fd060bd1e729fc93048cea8901f8bacb7c`. (Accessed on 09/02/2017).

[34] Linux kernel source tree - kptr_restrict. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=455cd5ab305c90ffc422dd2e0fb634730942b257`. (Accessed on 10/09/2017).

[35] Linux-kernel archive: Linux 2.0 really is released. `http://lkml.iu.edu/hypermail/linux/kernel/9606.1/0056.html`. (Accessed on 11/14/2017).

[36] Github - kernelslacker/trinity: Linux system call fuzzer. `https://github.com/kernelslacker/trinity`. (Accessed on 11/14/2017).

[37] Github - google/syzkaller: syzkaller is an unsupervised, coverage-guided kernel fuzzer. `https://github.com/google/syzkaller`. (Accessed on 11/14/2017).

[38] R Skowyra, K Casteel, H Okhravi, N Zeldovich, and W Streilein. Systematic Analysis of Defenses Against Return Oriented Programming. pages 82–102, 2013.

[39] Jinku Li, Zhi Wang, Xuxian Jiang, and Mike Grace. Defeating Return-Oriented Rootkits With Return-less Kernels. 2010.

[40] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, and Ahmad-reza Sadeghi. Return-Oriented Programming without Returns. 2010.

[41] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[42] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch SideChannel Attacks: Bypassing SMAP and Kernel ASLR. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 368–379, 2016.

[43] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR.

[44] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX.

[45] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, Cristiano Giuffrida, and Herbert Bos. Poking Holes in Information Hiding This paper is included in the Proceedings of the Poking Holes in Information Hiding. 2016.

[46] Linux kernel source tree - randstruct plugin. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=313dd1b629219db50cad532dba6a3b3b22ffe622`. (Accessed on 09/06/2017).

[47] Hovav Shacham, Matthew Page, Ben Pfaff, and Dan Boneh. On the Effectiveness of Address-Space Randomization. 2014.

[48] Router exploitation cisco. `https://www.blackhat.com/presentations/bh-usa-09/LINDNER/BHUSA09-Lindner-RouterExploit-PAPER.pdf`. (Accessed on 09/11/2017).

[49] Chongkyung Kil and Jinsuk Jun. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software.

[50] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. 2015.

[51] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. Stopping Address Space Leakage for Code Reuse Attacks Categories and Subject Descriptors.

[52] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, and Stefan Mangard. KASLR is Dead : Long Live KASLR.

[53] Tyler Bletsch, Xuxian Jiang, and Vince W Freeh. Jump-Oriented Programming : A New Class of Code-Reuse Attack.

[54] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard TM : Protecting Pointers From Buffer Overflow Vulnerabilities. 2003.

[55] Sandeep Bhatkar and R Sekar. Data Space Randomization .

[56] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[57] J A Y Ligatti. Control-Flow Integrity Principles , Implementations , and Applications. V(February), 2007.

[58] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, Geoff Pike, Caroline Tice, Tom Roeder, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. 2014.

[59] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H Deng. ROPecker : A Generic and Practical Approach for Defending Against ROP Attacks. (February):23–26, 2014.

[60] Vasilis Pappas. kBouncer : Efficient and Transparent ROP Mitigation. pages 1–8, 2012.

[61] Lucas Davi, Ahmad-reza Sadeghi, Intel Cri-sc, Technische Universität, Daniel Lehmann, Technische Universität Darmstadt, Fabian Monrose, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets : On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection Stitching the Gadgets :. 2014.

[62] Nicholas Carlini, David Wagner, and Nicholas Carlini. ROP is Still Dangerous : Breaking Modern Defenses ROP is Still Dangerous : Breaking Modern Defenses. 2014.

[63] Stack shield. `http://www.angelfire.com/sk/stackshield/`. (Accessed on 09/06/2017).

[64] Nicolas Carlini, Antonio Barresi, David Wagner, and Thomas R Gross. Control-Flow Bending : On the Effectiveness of Control-Flow Integrity.

[65] Thurston H Y Dang and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. pages 1–12.

[66] Control-flow enforcement technology preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`. (Accessed on 09/07/2017).

APPENDICES

Appendix A: List of ALL CVEs

A.1 Un-initialized data

CVE-2016-9178	CVE-2013-7271	CVE-2013-3223
CVE-2016-5244	CVE-2013-7270	CVE-2013-3222
CVE-2016-5244	CVE-2013-7269	CVE-2013-3076
CVE-2016-4580	CVE-2013-7268	CVE-2013-2636
CVE-2016-4580	CVE-2013-7267	CVE-2013-2635
CVE-2016-4578	CVE-2013-7265	CVE-2013-2634
CVE-2016-4569	CVE-2013-7264	CVE-2013-2547
CVE-2016-4486	CVE-2013-7263	CVE-2013-2237
CVE-2016-4485	CVE-2013-4516	CVE-2013-2234
CVE-2016-4485	CVE-2013-4515	CVE-2013-2148
CVE-2016-4482	CVE-2013-4470	CVE-2013-2147
CVE-2016-4470	CVE-2013-3237	CVE-2013-2141
CVE-2016-0821	CVE-2013-3236	CVE-2012-6549
CVE-2015-8950	CVE-2013-3235	CVE-2012-6548
CVE-2015-8539	CVE-2013-3234	CVE-2012-6547
CVE-2015-7885	CVE-2013-3233	CVE-2012-6546
CVE-2015-7884	CVE-2013-3232	CVE-2012-6545
CVE-2015-5697	CVE-2013-3231	CVE-2012-6544
CVE-2014-9900	CVE-2013-3230	CVE-2012-6543
CVE-2014-9895	CVE-2013-3229	CVE-2012-6542
CVE-2014-9892	CVE-2013-3228	CVE-2012-6541
CVE-2014-1446	CVE-2013-3227	CVE-2012-6540
CVE-2014-1445	CVE-2013-3226	CVE-2012-6539
CVE-2014-1444	CVE-2013-3225	CVE-2012-6538
CVE-2013-7281	CVE-2013-3224	CVE-2012-6537

CVE-2012-3430	CVE-2010-4083	CVE-2010-3298
CVE-2011-4087	CVE-2010-4082	CVE-2010-3297
CVE-2011-2492	CVE-2010-4081	CVE-2010-3296
CVE-2011-2184	CVE-2010-4080	CVE-2010-2955
CVE-2011-1173	CVE-2010-4079	CVE-2010-2942
CVE-2011-1173	CVE-2010-4078	CVE-2009-3612
CVE-2011-1172	CVE-2010-4077	CVE-2009-3228
CVE-2011-1171	CVE-2010-4076	CVE-2009-3002
CVE-2011-1170	CVE-2010-4075	CVE-2009-3001
CVE-2011-1163	CVE-2010-4074	CVE-2009-2847
CVE-2011-1160	CVE-2010-4073	CVE-2009-2692
CVE-2011-1080	CVE-2010-4072	CVE-2009-1914
CVE-2011-1078	CVE-2010-3881	CVE-2009-1192
CVE-2011-1044	CVE-2010-3877	CVE-2009-0676
CVE-2011-0711	CVE-2010-3876	CVE-2008-2812
CVE-2010-4655	CVE-2010-3875	CVE-2008-0598
CVE-2010-4525	CVE-2010-3861	CVE-2005-4881
CVE-2010-4158	CVE-2010-3477	

A.2 Use-After-Free

CVE-2016-6828	CVE-2016-2184	CVE-2014-1737
CVE-2016-4805	CVE-2016-0723	CVE-2014-0131
CVE-2016-4794	CVE-2015-7312	CVE-2013-7446
CVE-2016-4557	CVE-2015-5706	CVE-2013-7348
CVE-2016-3951	CVE-2015-3636	CVE-2013-7026
CVE-2016-3841	CVE-2015-0568	CVE-2013-4343
CVE-2016-2547	CVE-2014-6417	CVE-2013-4127
CVE-2016-2546	CVE-2014-5332	CVE-2013-2017
CVE-2016-2546	CVE-2014-4653	CVE-2013-1797
CVE-2016-2544	CVE-2014-3153	CVE-2013-1767
CVE-2016-2384	CVE-2014-2568	CVE-2012-3511

CVE-2012-3510	CVE-2011-1479	CVE-2010-1188
CVE-2012-2745	CVE-2011-0714	CVE-2009-4141
CVE-2012-2133	CVE-2010-4526	CVE-2007-1592
CVE-2012-1583	CVE-2010-4169	CVE-2007-0772
CVE-2012-1583	CVE-2010-3080	

A.3 Bounds Check

A.3.1 Stack Overflow

CVE-2016-8658	CVE-2014-0049	CVE-2010-1451
CVE-2016-8650	CVE-2013-4588	CVE-2009-4020
CVE-2016-7042	CVE-2013-1928	CVE-2009-2584
CVE-2016-6187	CVE-2012-4530	CVE-2009-2406
CVE-2015-2666	CVE-2012-3364	CVE-2008-5025
CVE-2014-9410	CVE-2011-4913	CVE-2008-3911
CVE-2014-8884	CVE-2011-4330	CVE-2007-3105
CVE-2014-6184	CVE-2011-1180	
CVE-2014-5471	CVE-2010-3858	
CVE-2014-3181	CVE-2010-3848	

A.3.2 Heap Overflow

CVE-2016-7425	CVE-2014-4323	CVE-2013-4514
CVE-2016-6516	CVE-2014-4322	CVE-2013-4513
CVE-2016-5829	CVE-2014-3186	CVE-2013-4512
CVE-2016-5343	CVE-2014-3185	CVE-2013-4387
CVE-2016-5342	CVE-2014-3184	CVE-2013-2894
CVE-2016-4568	CVE-2014-3183	CVE-2013-2893
CVE-2016-3134	CVE-2013-6382	CVE-2013-2892
CVE-2015-5156	CVE-2013-6381	CVE-2013-2891
CVE-2014-6416	CVE-2013-4591	CVE-2013-2890

CVE-2013-2889	CVE-2011-1759	CVE-2009-1389
CVE-2013-2850	CVE-2011-1577	CVE-2009-0065
CVE-2013-1929	CVE-2011-1017	CVE-2008-5702
CVE-2013-1860	CVE-2011-1010	CVE-2008-5134
CVE-2013-1796	CVE-2011-0712	CVE-2008-4933
CVE-2013-1773	CVE-2010-4656	CVE-2008-4395
CVE-2013-1772	CVE-2010-4650	CVE-2008-3915
CVE-2013-0913	CVE-2010-4527	CVE-2008-3496
CVE-2012-3400	CVE-2010-3874	CVE-2008-3247
CVE-2012-2319	CVE-2010-3873	CVE-2008-2750
CVE-2012-2137	CVE-2010-3084	CVE-2008-0352
CVE-2012-2136	CVE-2010-2492	CVE-2007-6151
CVE-2012-2119	CVE-2010-1451	CVE-2007-6063
CVE-2011-4077	CVE-2010-1436	CVE-2007-5904
CVE-2011-3359	CVE-2010-1084	CVE-2007-1217
CVE-2011-3353	CVE-2009-4005	CVE-2016-4997
CVE-2011-2700	CVE-2009-4004	CVE-2013-4387
CVE-2011-2534	CVE-2009-3234	CVE-2010-1084
CVE-2011-2517	CVE-2009-2407	
CVE-2011-2182	CVE-2009-1633	
CVE-2011-1776	CVE-2009-1439	

A.3.3 Buffer Overread

CVE-2016-5696	CVE-2010-0003	CVE-2011-0710
CVE-2013-4299	CVE-2016-2117	CVE-2010-4565
CVE-2013-4350	CVE-2009-2910	CVE-2010-4563
CVE-2013-1798	CVE-2008-2729	CVE-2010-1636
CVE-2012-4467	CVE-2007-6206	CVE-2010-0415
CVE-2011-1079	CVE-2012-0957	CVE-2014-2038
CVE-2010-4563	CVE-2011-1162	CVE-2014-1690
CVE-2010-2943	CVE-2011-1020	CVE-2013-4579

CVE-2013-4350	CVE-2014-9731	CVE-2016-7915
CVE-2013-2898	CVE-2014-9585	CVE-2014-7825
CVE-2013-2546	CVE-2014-9419	CVE-2014-3985
CVE-2016-4913	CVE-2014-8709	CVE-2016-7917
CVE-2014-8709	CVE-2014-8133	CVE-2016-4998
CVE-2013-2164	CVE-2016-6480	CVE-2015-8575
CVE-2013-0349	CVE-2016-6156	CVE-2015-8569
CVE-2013-0343	CVE-2013-7027	CVE-2013-2548
CVE-2013-0160	CVE-2016-2064	CVE-2013-7266
CVE-2011-2909	CVE-2014-9728	CVE-2012-6536
CVE-2016-5243	CVE-2007-2172	CVE-2014-9584
CVE-2016-2383	CVE-2014-3985	CVE-2013-1828
CVE-2016-2117	CVE-2007-1734	CVE-2007-4571
CVE-2016-0823	CVE-2009-0787	CVE-2014-9903
CVE-2015-8374	CVE-2008-4445	CVE-2015-1593
CVE-2015-2042	CVE-2011-0463	CVE-2007-4571
CVE-2015-2041	CVE-2016-7917	

A.3.4 Integer Overflow

CVE-2016-9084	CVE-2014-4655	CVE-2012-0038
CVE-2016-3135	CVE-2014-4611	CVE-2011-4611
CVE-2016-0758	CVE-2014-4611	CVE-2011-4097
CVE-2015-8830	CVE-2014-4611	CVE-2011-2496
CVE-2015-5707	CVE-2014-4608	CVE-2011-2022
CVE-2015-4167	CVE-2014-4608	CVE-2011-1759
CVE-2015-4167	CVE-2014-2889	CVE-2011-1746
CVE-2014-9904	CVE-2013-4511	CVE-2011-1745
CVE-2014-7975	CVE-2013-2596	CVE-2011-1593
CVE-2014-4656	CVE-2012-6703	CVE-2011-1494
CVE-2014-4656	CVE-2012-2383	CVE-2010-4649
CVE-2014-4655	CVE-2012-0044	CVE-2010-4175

CVE-2010-4162	CVE-2010-3015	CVE-2008-3526
CVE-2010-4160	CVE-2010-2959	CVE-2008-3276
CVE-2010-4157	CVE-2010-2538	CVE-2008-2826
CVE-2010-3865	CVE-2010-2478	CVE-2008-2358
CVE-2010-3442	CVE-2009-3638	CVE-2007-5966
CVE-2010-3081	CVE-2009-1265	
CVE-2010-3067	CVE-2009-1265	

A.3.5 Integer Underflow

CVE-2011-1770	CVE-2007-4997	CVE-2010-4529
CVE-2010-4164	CVE-2011-4913	CVE-2014-3144
CVE-2009-2846	CVE-2007-2875	
CVE-2009-1385	CVE-2011-1476	

A.3.6 Integer Signedness

CVE-2011-1013	CVE-2009-3280	CVE-2011-2209
CVE-2011-0521	CVE-2007-1730	CVE-2011-2208
CVE-2010-3859	CVE-2007-4573	CVE-2010-3310
CVE-2010-3437	CVE-2009-2909	CVE-2009-0029
CVE-2010-3301	CVE-2011-2906	

A.3.7 Refcount Overflow

CVE-2016-4558	CVE-2014-2851	CVE-2012-2127
CVE-2014-0205	CVE-2012-2127	CVE-2010-0623
CVE-2014-5045	CVE-2009-3624	CVE-2013-4483
CVE-2016-0728	CVE-2008-3077	

A.3.8 Array Index Errors

CVE-2009-3080	CVE-2013-1763	CVE-2013-4247
CVE-2013-4587	CVE-2011-1169	CVE-2008-1673
CVE-2013-2888	CVE-2014-3182	CVE-2009-1046
CVE-2011-1477	CVE-2008-5701	CVE-2014-9683
CVE-2011-1493	CVE-2015-4036	CVE-2015-3214
CVE-2016-0774	CVE-2011-2695	
CVE-2016-3713	CVE-2008-3535	

A.4 NULL Dereference

CVE-2015-8746	CVE-2013-4254	CVE-2009-2698
CVE-2003-1604	CVE-2013-1059	CVE-2009-2695
CVE-2015-8543	CVE-2013-2206	CVE-2009-1897
CVE-2015-7990	CVE-2011-2482	CVE-2009-1360
CVE-2015-6937	CVE-2011-2942	CVE-2009-1298
CVE-2014-8173	CVE-2013-3301	CVE-2008-5033
CVE-2014-7841	CVE-2013-1827	CVE-2008-3792
CVE-2014-7145	CVE-2013-1826	CVE-2007-6694
CVE-2011-2519	CVE-2013-0313	CVE-2007-5501
CVE-2011-0695	CVE-2013-0310	CVE-2007-4567
CVE-2011-0709	CVE-2012-2744	CVE-2007-3642
CVE-2010-4263	CVE-2012-1097	CVE-2007-2876
CVE-2010-4342	CVE-2011-2525	CVE-2007-1000
CVE-2010-2960	CVE-2011-1478	CVE-2011-1927
CVE-2010-2798	CVE-2011-1076	CVE-2015-8955
CVE-2010-1643	CVE-2011-1093	CVE-2016-4951
CVE-2010-0437	CVE-2009-4308	CVE-2013-2895
CVE-2010-0006	CVE-2009-3726	CVE-2009-4138
CVE-2014-3631	CVE-2009-3623	CVE-2009-4021
CVE-2014-3535	CVE-2009-2844	CVE-2009-3640
CVE-2014-5077	CVE-2009-2768	CVE-2009-3620
CVE-2014-0101	CVE-2009-2767	CVE-2009-3043

CVE-2009-2908	CVE-2010-3066	CVE-2016-3138
CVE-2009-2849	CVE-2010-2954	CVE-2016-3137
CVE-2009-2287	CVE-2010-1488	CVE-2016-3136
CVE-2009-0748	CVE-2010-1187	CVE-2016-3070
CVE-2008-3686	CVE-2010-1148	CVE-2016-2782
CVE-2008-1514	CVE-2010-0622	CVE-2016-2543
CVE-2007-3731	CVE-2009-4895	CVE-2016-2188
CVE-2007-3104	CVE-2008-7256	CVE-2016-2187
CVE-2007-1496	CVE-2014-2739	CVE-2016-2186
CVE-2007-1388	CVE-2014-2678	CVE-2016-2185
CVE-2007-0822	CVE-2014-1874	CVE-2015-8970
CVE-2007-0006	CVE-2013-7339	CVE-2015-8956
CVE-2006-7203	CVE-2013-6432	CVE-2015-8746
CVE-2012-5517	CVE-2013-6431	CVE-2015-8551
CVE-2012-1601	CVE-2013-6380	CVE-2015-8324
CVE-2011-4594	CVE-2013-5634	CVE-2015-5257
CVE-2011-4325	CVE-2013-3302	CVE-2015-4692
CVE-2011-4110	CVE-2013-2899	CVE-2014-9715
CVE-2011-4081	CVE-2013-2897	CVE-2014-8481
CVE-2011-2928	CVE-2013-2896	CVE-2014-8480
CVE-2011-2518	CVE-2009-3288	CVE-2014-7841
CVE-2011-2203	CVE-2009-3288	CVE-2014-7207
CVE-2011-2183	CVE-2013-1819	CVE-2011-5321
CVE-2011-1927	CVE-2013-1774	CVE-2015-7799
CVE-2011-1771	CVE-2012-6647	CVE-2015-7566
CVE-2011-1748	CVE-2011-3619	CVE-2015-7550
CVE-2011-1598	CVE-2016-6327	CVE-2015-7515
CVE-2010-4242	CVE-2016-4581	CVE-2015-3288
CVE-2010-3849	CVE-2016-3140	
CVE-2010-3079	CVE-2016-3139	

A.5 Format String

CVE-2013-2851	CVE-2013-1848
CVE-2013-2852	

A.6 Missing Permission Check

CVE-2011-2905	CVE-2013-2094	CVE-2009-0835
CVE-2012-2123	CVE-2013-1979	CVE-2009-0834
CVE-2009-4536	CVE-2013-1858	CVE-2009-0675
CVE-2009-4131	CVE-2013-0268	CVE-2009-0028
CVE-2009-3725	CVE-2012-4444	CVE-2008-4554
CVE-2009-3722	CVE-2012-0028	CVE-2008-4210
CVE-2009-3290	CVE-2012-0056	CVE-2008-4113
CVE-2008-3525	CVE-2010-4347	CVE-2008-3833
CVE-2008-2931	CVE-2010-2963	CVE-2008-3527
CVE-2007-4998	CVE-2010-2962	CVE-2008-1294
CVE-2007-3851	CVE-2010-2537	CVE-2008-0163
CVE-2007-1497	CVE-2010-1146	CVE-2008-0001
CVE-2016-3699	CVE-2010-0298	CVE-2007-6434
CVE-2016-5340	CVE-2015-1593	CVE-2007-3850
CVE-2016-1576	CVE-2011-1016	CVE-2007-3848
CVE-2012-6689	CVE-2010-4258	CVE-2007-3843
CVE-2016-1575	CVE-2009-1895	CVE-2007-3740
CVE-2015-8660	CVE-2013-1943	CVE-2007-2480
CVE-2015-2925	CVE-2013-0228	CVE-2007-1497
CVE-2014-8160	CVE-2009-3286	CVE-2007-0958
CVE-2014-5207	CVE-2009-2848	CVE-2012-5532
CVE-2014-5206	CVE-2009-1883	CVE-2012-4444
CVE-2014-3534	CVE-2009-1630	CVE-2012-3520
CVE-2014-4943	CVE-2009-1338	CVE-2012-2669
CVE-2014-4014	CVE-2009-1337	CVE-2012-2313
CVE-2013-6383	CVE-2009-1184	CVE-2011-4127
CVE-2013-4300	CVE-2009-1072	CVE-2011-4112

CVE-2011-4080	CVE-2013-4270	CVE-2016-0821
CVE-2011-2495	CVE-2013-2930	CVE-2015-8944
CVE-2011-2494	CVE-2013-2929	CVE-2015-4176
CVE-2011-2484	CVE-2013-1959	CVE-2015-2830
CVE-2011-0726	CVE-2013-1958	CVE-2015-0239
CVE-2011-0006	CVE-2013-1957	CVE-2014-9717
CVE-2010-4648	CVE-2013-1956	CVE-2014-8989
CVE-2010-4346	CVE-2012-4542	CVE-2014-8160
CVE-2010-3850	CVE-2011-4347	CVE-2014-7975
CVE-2010-3448	CVE-2011-1585	CVE-2014-4654
CVE-2010-2946	CVE-2011-1182	CVE-2014-3690
CVE-2010-2524	CVE-2011-1019	CVE-2013-7421
CVE-2010-2226	CVE-2016-7097	CVE-2013-6335
CVE-2010-2071	CVE-2016-3672	CVE-2013-4312
CVE-2010-2066	CVE-2016-2854	CVE-2010-1446
CVE-2010-1641	CVE-2016-2853	CVE-2011-1021
CVE-2014-1738	CVE-2016-2550	
CVE-2014-0181	CVE-2016-1237	

A.7 Race Conditions

CVE-2009-1527	CVE-2009-3547	CVE-2009-1388
CVE-2008-5182	CVE-2014-9529	CVE-2009-1243
CVE-2008-1669	CVE-2015-0572	CVE-2009-0935
CVE-2008-1375	CVE-2014-2672	CVE-2008-4307
CVE-2007-0997	CVE-2011-4348	CVE-2008-4302
CVE-2014-0196	CVE-2012-3552	CVE-2008-2365
CVE-2015-3339	CVE-2010-2653	CVE-2012-4508
CVE-2014-4699	CVE-2009-4027	CVE-2012-2373
CVE-2013-0871	CVE-2014-9710	CVE-2011-1833
CVE-2016-5728	CVE-2009-2691	CVE-2011-1082
CVE-2016-2059	CVE-2009-1961	CVE-2010-4248

CVE-2010-4161	CVE-2016-2549	CVE-2014-8559
CVE-2010-1437	CVE-2016-2545	CVE-2014-8086
CVE-2014-2706	CVE-2016-2069	CVE-2014-7842
CVE-2015-7613	CVE-2015-8839	CVE-2014-4652
CVE-2013-1792	CVE-2015-8767	CVE-2014-3611
CVE-2016-7916	CVE-2015-4170	CVE-2010-5313
CVE-2016-6136	CVE-2015-3212	CVE-2014-3122
CVE-2016-6130	CVE-2015-1420	

A.8 Miscellaneous

A.8.1 Infinite Loop

CVE-2008-5079	CVE-2013-4348	CVE-2014-6410
CVE-2007-6712	CVE-2013-0290	CVE-2008-7316
CVE-2007-1861	CVE-2015-8785	
CVE-2011-2213	CVE-2015-6526	
CVE-2010-3880	CVE-2014-9420	

A.8.2 Memory Leak

CVE-2009-2903	CVE-2009-0031	CVE-2013-4592
CVE-2008-2136	CVE-2007-2525	CVE-2013-4205
CVE-2007-6417	CVE-2008-2372	CVE-2016-5400
CVE-2013-0217	CVE-2012-2390	CVE-2015-1339
CVE-2014-3688	CVE-2012-2121	CVE-2015-1333
CVE-2014-2309	CVE-2010-4250	

A.8.3 Divide-by-zero

CVE-2009-4307	CVE-2012-0207	CVE-2010-4165
CVE-2016-2070	CVE-2010-1085	CVE-2015-7513
CVE-2015-4003	CVE-2012-4565	
CVE-2013-6367	CVE-2011-1012	

A.8.4 Cryptographic Bugs

CVE-2013-4345	CVE-2007-2451	CVE-2007-2451
CVE-2009-3238	CVE-2014-7284	CVE-2016-2085
CVE-2007-4311	CVE-2007-2453	

A.8.5 Length Calculation bugs

CVE-2014-0077	CVE-2015-1465	CVE-2008-0007
CVE-2014-0069	CVE-2014-9428	CVE-2009-0269
CVE-2013-6763	CVE-2011-1573	CVE-2011-1573
CVE-2011-4604	CVE-2009-4537	CVE-2015-8019
CVE-2015-2686	CVE-2008-4618	CVE-2013-2128
CVE-2011-4914	CVE-2008-3272	
CVE-2011-1495	CVE-2008-1675	

A.8.6 Other

CVE-2016-8666	CVE-2014-0155	CVE-2011-1768
CVE-2016-7039	CVE-2014-0102	CVE-2011-1767
CVE-2016-5828	CVE-2012-6638	CVE-2012-1179
CVE-2016-2053	CVE-2013-6376	CVE-2011-4326
CVE-2014-3687	CVE-2013-4563	CVE-2011-2723
CVE-2014-3673	CVE-2013-4125	CVE-2010-4805
CVE-2014-6418	CVE-2013-0216	CVE-2010-4251
CVE-2014-4667	CVE-2012-3412	CVE-2010-3432

CVE-2010-2248	CVE-2008-3831	CVE-2011-3209
CVE-2010-1173	CVE-2008-3534	CVE-2011-2918
CVE-2010-1086	CVE-2008-3528	CVE-2011-2689
CVE-2010-0008	CVE-2008-3275	CVE-2011-2521
CVE-2005-4886	CVE-2008-2148	CVE-2011-2493
CVE-2009-4272	CVE-2008-2137	CVE-2011-1747
CVE-2009-4026	CVE-2007-6716	CVE-2011-1581
CVE-2009-0778	CVE-2007-5500	CVE-2011-1090
CVE-2008-4609	CVE-2007-5498	CVE-2011-1083
CVE-2008-4576	CVE-2007-5093	CVE-2011-1023
CVE-2007-3380	CVE-2007-5087	CVE-2011-0716
CVE-2007-2764	CVE-2007-4133	CVE-2010-4668
CVE-2006-6535	CVE-2007-3720	CVE-2010-4343
CVE-2009-4031	CVE-2007-3719	CVE-2010-4256
CVE-2009-3613	CVE-2007-3513	CVE-2010-4249
CVE-2008-4934	CVE-2007-3380	CVE-2010-4163
CVE-2015-8215	CVE-2007-3107	CVE-2010-3698
CVE-2009-4410	CVE-2007-2878	CVE-2010-3086
CVE-2009-3888	CVE-2007-0771	CVE-2010-2938
CVE-2009-3621	CVE-2006-7051	CVE-2010-0727
CVE-2009-1242	CVE-2006-6921	CVE-2010-0410
CVE-2009-0859	CVE-2012-3375	CVE-2010-0307
CVE-2009-0747	CVE-2012-2375	CVE-2009-4271
CVE-2009-0746	CVE-2012-1090	CVE-2007-6733
CVE-2009-0745	CVE-2012-0879	CVE-2014-3145
CVE-2009-0322	CVE-2012-0058	CVE-2014-2673
CVE-2008-6107	CVE-2012-0045	CVE-2014-2039
CVE-2008-5713	CVE-2011-4621	CVE-2014-1438
CVE-2008-5700	CVE-2011-4324	CVE-2013-6378
CVE-2008-5395	CVE-2011-4132	CVE-2013-4220
CVE-2008-5300	CVE-2011-4131	CVE-2013-4163
CVE-2008-5029	CVE-2011-4086	CVE-2013-4162
CVE-2008-4410	CVE-2011-3637	CVE-2013-4129

CVE-2013-2232	CVE-2015-8844	CVE-2014-3646
CVE-2013-2146	CVE-2015-8215	CVE-2014-3645
CVE-2013-2140	CVE-2015-8104	CVE-2014-3610
CVE-2013-2058	CVE-2015-7872	CVE-2014-3601
CVE-2013-2015	CVE-2015-7509	CVE-2012-6657
CVE-2013-0309	CVE-2015-6252	CVE-2011-0999
CVE-2013-0231	CVE-2015-5366	CVE-2013-6368
CVE-2013-0190	CVE-2015-5307	CVE-2016-4440
CVE-2012-5375	CVE-2015-5283	CVE-2016-2143
CVE-2012-5374	CVE-2015-4700	CVE-2015-8816
CVE-2012-4398	CVE-2015-4178	CVE-2015-8767
CVE-2011-4098	CVE-2015-4177	CVE-2015-5366
CVE-2011-3638	CVE-2015-3332	CVE-2015-5364
CVE-2011-2491	CVE-2015-3291	CVE-2013-0311
CVE-2011-2479	CVE-2015-2672	CVE-2011-3188
CVE-2016-9191	CVE-2015-2150	CVE-2011-2699
CVE-2016-8660	CVE-2015-1573	CVE-2011-2189
CVE-2016-8646	CVE-2015-1350	CVE-2010-1162
CVE-2016-8645	CVE-2015-0275	CVE-2010-0741
CVE-2016-8630	CVE-2014-9730	CVE-2010-1087
CVE-2016-6198	CVE-2014-9729	CVE-2011-3363
CVE-2016-6197	CVE-2014-9090	CVE-2015-0274
CVE-2016-6162	CVE-2014-8369	CVE-2009-1336
CVE-2016-5412	CVE-2014-8172	CVE-2014-9888
CVE-2016-3689	CVE-2014-7970	CVE-2015-2922
CVE-2016-3156	CVE-2014-7843	CVE-2014-9644
CVE-2016-2847	CVE-2014-7283	CVE-2009-0024
CVE-2016-2548	CVE-2014-5472	CVE-2010-0291
CVE-2015-8953	CVE-2014-4667	
CVE-2015-8952	CVE-2014-3688	
CVE-2015-8845	CVE-2014-3647	