AN EMPIRICAL STUDY ON USE-AFTER-FREE VULNERABILITIES

by

Benjamin P. Steenkamer

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering

Summer 2019

AN EMPIRICAL STUDY ON USE-AFTER-FREE VULNERABILITIES

by

Benjamin P. Steenkamer

Approved: ______________________________

Haining Wang, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: ______________________________

Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: ______________________________

Levi T. Thompson, Ph.D.
Dean of the College of Engineering

Approved: ______________________________

Douglas J. Doren, Ph.D.
Interim Vice Provost for Graduate and Professional Education and
Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to thank Dr. Haining Wang for being my advisor during graduate school and for providing me guidance on my academic path, the research I pursued, and in writing this thesis. I thank Dr. Daiping Liu for providing his knowledge and suggestions on this thesis and for his help guiding me on the research topics I pursued. I thank Zeyu Chen for providing his input on this thesis and my related research as well as assisting in gathering and analyzing data used in this thesis. I would also like to thank Michael Guerrero, who provided his time and effort to help gather and analyze data used in this thesis.

I thank my whole family for providing their love and support through all my life and in everything I do. I especially want to thank my parents for their love, support, and advice through the most challenging part of my life so far. Finally, I want to thank my close friends who supported me with their own experiences, suggestions, and companionship.

TABLE OF CONTENTS

LIST OF TABLES

ABSTRACT

For many years, use-after-free vulnerabilities have been a persistent security threat to kernels, operating systems, and user-level applications written in programming languages that are not memory safe. When left unchecked, use-after-free (UAF) vulnerabilities can be unintentionally introduced, difficult for developers to discover, and very dangerous when exploited by attackers. To assess the scope of the UAF security threat, this study has analyzed 36 previously discovered UAF vulnerabilities that occurred in the Linux kernel and the Mozilla Firefox web browser. Through this analysis, it was found that UAFs can frequently lead to the creation of severe attacks when exploited. In the most common case, almost 81% of the analyzed UAF vulnerabilities allowed for the possibility of a denial of service attack when exploited. It was also observed that 44% of the vulnerabilities could allow for arbitrary code execution if exploited. The security patches used to remove 13 UAF vulnerabilities in the Linux kernel were also analyzed, and it was found that these patches can require significant code changes to fully remove the vulnerabilities.

Based on the results of this study, it is evident that better tools for detecting UAF vulnerabilities need to be developed for the effective mitigation and prevention of this long-term security threat. It is hoped that the results of this study will be used as the basis for future investigations on the nature of UAF vulnerabilities and for the development of new testing tools for UAF discovery and prevention.

Chapter 1
INTRODUCTION

A use-after-free (UAF) vulnerability is a type of exploitable bug that occurs when previously freed memory is accessed through a "dangling pointer." A dangling pointer is simply a pointer that points to previously freed memory. By accessing freed memory through a dangling pointer, an attacker can manipulate program memory in ways the program or system is not expecting. This can cause data corruption, sensitive data leakage, a program crash (denial of service), or potentially allow an attacker to run arbitrary code. According to the Common Weaknesses Enumeration, a reputable database of software weaknesses, if a vulnerable use-after-free bug is present in a piece of software, there is a high likelihood that an exploit can be developed to take advantage of it [1]. Use-after-free vulnerabilities can also be difficult for developers to detect, as they are sometimes the result of obscure program states or unusual input data that are not handled correctly. Due to the severe security issues use-after-free vulnerabilities create, it is essential that more research be done to identify patterns in how these vulnerabilities are introduced, exploited, and fixed.

To better understand the nature of UAFs, this empirical study examines a total of 36 real-world UAF vulnerabilities that have been previously discovered in the Linux kernel and the Mozilla Firefox web browser. Using the information provided by the software developers and reputable security institutions, these UAF vulnerabilities are analyzed to determine how they could have been potentially or actually exploited by attackers. This is done to determine the common types of attacks that result from exploited UAFs. This study also goes into a deeper analysis on 13 UAF vulnerabilities found in the Linux kernel to determine the ways in which developers were able to remove them. These two types of analysis will help form the basis of a more systematized view of UAFs and hopefully lead to the development of more effective UAF prevention and detection methods.

The rest of this thesis will cover the following topics: background information on use-after-free vulnerabilities, an overview on a selection of previous UAF vulnerability research and related empirical studies (chapter 2), explanations of the methods used to conduct this empirical study (chapter 3), a listing of the use-after-free data collected for this study (chapter 4), and a detailed analysis of this data (chapter 5). Finally, the future work for this study will be discussed (chapter 6), along with a conclusion to summarize the main findings of this study (chapter 7). Appendix A also has tables that list all the vulnerabilities used in this study.

Chapter 2
BACKGROUND

In the following sections, more detail is given on how use-after-free (UAF) vulnerabilities can be introduced into a program, ways in which they are exploited, and an overview of previous research conducted on UAF vulnerabilities. The purpose of this chapter is to set up the base knowledge needed to better understand the scope and severity of UAF vulnerabilities.

2.1 A Use-after-free Code Example

To clearly demonstrate what a UAF bug is, an example of one is given below as a C code snippet. As it is written, this example highlights just one way in which a UAF bug can be introduced into a program. However, there are an innumerable number of ways that they can be introduced through programming constructs and specific implementations.

```
0: char *ptr = malloc(1);
1: free(ptr);
... // More lines of code here
12: *ptr = 'x';
```

In line 0 of this example, the pointer variable `ptr` is assigned the memory location of a newly allocated byte on the heap. In line 1, the memory location that `ptr` points to is immediately freed, allowing the memory manager to allocate that location again in the future. At this point, `ptr` is considered a dangling pointer because it now points to a freed memory location. Lines 2 through 11 are not shown but would have some additional logic for this example program. Then in line 12, `ptr` is dereferenced and assigned a value. In other words, the memory location pointed to by `ptr` was written to (i.e., used) after it was freed.

This usage of `ptr` is a bug because undefined behavior can occur when writing to a memory location that is not allocated for the pointer through which it was accessed. For example, somewhere in lines 2 through 11 there could be another memory allocation where the memory manager decides to reuse the freed location that `ptr` still points to. Then this memory location has important program data written to it. When line 12 is executed, part of the important data is overwritten or corrupted by the UAF with `ptr`. This corruption of data then leads to undefined program behavior.

It is also possible for a UAF to occur if instead `ptr` in line 12 is used to read from the memory location. This would also be a UAF, but it would not necessarily corrupt the data like the original example could. However, this bug can still be an issue for many reasons, such as a scenario where the data being read contains sensitive information and `ptr` is controlled and viewable by an attacker.

2.2 Use-after-frees and Memory Safety

In more general terms, UAFs are categorized as memory errors because they violate the principles of proper memory usage and lead to undefined behavior [1]. A memory safety policy, which is implemented in many programming languages, enforces proper memory usage and prevents UAFs and other memory errors from occurring. Unfortunately, the C programming language does not have native support for a memory safety policy [2], so the UAF bug shown in chapter 2.1 will pass through all C compiler checks and can execute without throwing any errors. Since C and other memory unsafe languages like C++ remain very popular for application and operating system development, the threat posed by UAFs is pervasive.

2.3 Use-after-frees As Exploitable Vulnerabilities

The example from chapter 2.1 demonstrates one of many ways in which UAF bugs can be introduced and cause issues. Like other programming bugs, UAF bugs become vulnerabilities when there is potential for an attacker to exploit them for malicious purposes. The ways in which an attacker exploits UAF vulnerabilities and other memory errors was previously systematized in the work of Szekeres et al. [2]. In this work, the authors state that two possible ways in which memory errors occur are through the use of an out-of-bounds pointer or a dangling pointer to write, read, or free memory. Szekeres et al. constructed an attack model to show how these memory errors can be exploited to cause a variety of attacks. These attacks were generalized as: "code corruption," "control-flow hijack," "data only," and "information leak." Each of these attacks deteriorates one or more aspect of a program's confidentiality, integrity, or availability.

In the attack model created by Szekeres et al., attackers are shown to have several possible vectors in which they can exploit a memory error and launch a particular attack. However, it should be noted that in practice, it may not be feasible for an attacker to perform a certain exploit for a given UAF or memory error in general. The variety of programs, the ways in which they are implemented, and the source of the memory error all contribute to making a certain exploit more or less difficult to achieve. The authors also note that a dangling pointer in heap memory is more likely to be exploited than a pointer to a location on the stack. However, it is possible for a pointer to a location on the stack to become dangling if assigned to a global variable. Many of the UAFs analyzed in this study are of the heap memory kind. Regardless of the exploit difficulty and dangling pointer location, UAFs are dangerous security vulnerabilities due to the severity of attacks they facilitate.

2.4 Use-after-free Defense Research

Due to the severe security threat that UAF vulnerabilities pose, there have been many previous research studies that have developed programs for UAF mitigation or prevention. The specific goals for each program vary, but in general, they try to statically or dynamically detect and locate the source of UAF vulnerabilities so that developers can fix their code and remove the vulnerabilities. In some cases, these programs try to prevent the exploitation of UAFs during program runtime.

These UAF defense programs have reported varying levels of effectiveness in detecting UAFs or preventing UAFs from being exploited. However, even with the development of this technology, UAFs still continue to pose a significant threat to software written in languages that are not memory safe. This is not to say these programs provide little benefit toward improving software security. Rather, some of the recent developments in UAF defense offer a great benefit over previous technology and certainly offer a better chance of finding UAFs over not using any defense software at all.

Some of these defense programs also detect other types of memory errors along with UAFs, and due to their relative effectiveness, they have become a crucial part of many software projects. One such defense program is called AddressSanitizer (ASan) [3], which is a popular memory error detector developed by a team at Google. ASan is capable of detecting out-of-bounds and UAF memory errors. To achieve this, it requires the target program to be recompiled with ASan's own memory allocator and code instrumentation, which adds runtime overhead to the program. This overhead makes ASan less likely to be included in the release version of software and typically reserves it to use in developer builds and testing.

During the collection of UAFs for this empirical study, it was found that ASan was being used in the development and testing cycle for a variety of programs. With ASan, developers were able to identify and fix many of the UAFs analyzed in this study. However, even with the use of a memory error detector like ASan, the complex nature of UAFs means they can slip past the initial testing process and be present in released software. This is evident by the fact that many UAFs examined in this study were found in release versions of software that use ASan for testing.

2.4.1 Targeted Use-after-free Defenses

Beyond general memory error detectors like ASan, there exist programs that specifically target and defend against UAFs. These programs achieve this through a variety of methods during a program's runtime. The following programs are a selection of recent developments in this area: DangNull [4] prevents UAFs by nullifying all related pointers to a memory location once it is freed, Oscar [5] implements UAF protection by using the principle of page permissions, and pSweeper [6] protects against UAFs by using "concurrent pointer sweeping" and "object origin tracking."

While these programs can be effective at stopping the exploitation of UAFs, they have other trade-offs. A common trait among these defenses is that they add overhead to the base program in the form of decreased runtime performance and increased memory usage. This can make them less appealing to developers and users who desire fast and lightweight software. However, the recent developments in this area have yielded increased effectiveness in UAF protection and decreased runtime overhead compared to previous defense programs. This trend of improvement will hopefully continue into the future and allow runtime UAF defenses to become more viable for production software.

2.4.2 Program Fuzzing

Program fuzzing is a way to test programs by feeding in purposely malformed or randomly generated input data. These unusual inputs can reveal overlooked edge cases, unintended functionality that leads to instability (e.g., data corruption or program crashes), or security vulnerabilities like UAFs. In short, program fuzzing can find bugs in a program by generating and using random inputs. The program used to generate these inputs is called a "fuzzer." Many modern fuzzers can be configured to run tests automatically, making them appealing for use in production testing environments. An important note about fuzzers is that due to the nature of randomly generated inputs, it can sometimes take hours, days, or longer for a fuzzer to generate the exact input sequence needed to trigger a latent bug. The random behavior of a fuzzer must always be kept in mind when determining its effectiveness or using one to find bugs or vulnerabilities [7].

Fuzzers can discover many common types of bugs, including memory errors like array out-of-bounds accesses and UAFs. An empirical study by Miller et al. [8] is an early example of using random data inputs to test the reliability of software. In this study, it was found that a simple fuzzing program was "able to crash more than 24%" of the "almost 90 different utility programs on seven versions of UNIX." Out of all the causes of these crashes, incorrect pointer and array accesses where the most common. These memory errors included array out-of-bounds accesses, accesses through null pointers, and accesses through pointers with incorrect addresses. This last type of error underlies the same issue of UAFs, where dangling pointers have "incorrect" addresses to memory no longer allocated to them. In this empirical study published almost 30 years ago, it was shown that fuzzers are an effective means of testing for memory errors in software.

American fuzzy lop (AFL) [9] is an example of a popular fuzzer that is used in many software projects for automated bug testing. AFL has seen great success in finding many bugs across a variety of programs. Because of programs like AFL, fuzzers are now a common way of testing for bugs and vulnerabilities. When coupled with memory error detection programs like ASan, fuzzers can be effective automated testing tools for finding UAFs in software.

2.5 Relevant Empirical Studies

There is a large body of previous empirical studies on software bugs and the effectiveness of programs used to detect them. When preparing this thesis, the following three empirical studies had an influence on the ways in which UAF vulnerabilities and patch data were collected and how they were analyzed. These empirical studies are also considered related work as they cover similar issues of bug analysis or are relevant to the future work of developing a better UAF fuzzer (see chapter 6.2). The three studies and a brief description of each are now given. Lu et al. [10] performed an empirical study on 105 concurrency bugs from four popular programs, and through their findings and analysis, created a set of guidelines “for concurrency bug detection, testing, and concurrent programming language design.” Jin et al. [11] studied 109 performance bugs from five popular programs and used their findings and analysis to also create “guidance for future work to avoid, expose, detect, and fix performance bugs.” Jin et al. also applied their findings to discover 332 previously undiscovered issues related to runtime performance within a set of programs they studied. Lastly, Klees et al. [7] looked through the findings of 32 papers on evaluating program fuzzing and determined each paper had issues in the way it conducted its evaluations. Similar to the other two studies, the authors used their findings to create a guide for improving and systematizing the ways in which fuzzers can be evaluated.

Chapter 3
DATA COLLECTION METHODS

In order to conduct an empirical study on UAF vulnerabilities, a sizable selection of known, real-world vulnerabilities need to be gathered and analyzed. These UAFs also need to be found in programs that are representative of commonly used software. The following sections detail the process of selecting the programs, the ways in which UAFs were searched for in the programs, and the specific documentation requirements for UAFs used in this study.

3.1 Selection of Programs

UAFs occur across a wide spread of programs that have many different use cases and development cycles. Since UAFs are a type of memory error, they will occur in programs written in languages that are not memory safe. Therefore, the programs to be investigated must be at least partially written in a memory unsafe language, like C or C++. Since C and C++ are very popular languages, the total number of possible programs to choose from isn't greatly limited. It is also necessary that a selected program is open source, meaning its source code is publicly accessible. This is so the source code can be analyzed to get a better understanding of why a given UAF occurred. Another qualification is that the program must have a well documented bug tracking system that is fully accessible to the public. This is necessary because it allows the viewing of developer discussions on UAF bug reports, which include the causes of UAFs, their effects, and the code patches implemented to remove them. Finally, the selected programs should have a large user base. This was decided because open source projects with many users typically have a large subgroup of users looking for vulnerabilities in the program. This increases the chance of finding well documented UAFs and makes the data collection process more streamlined.

With these qualifications in mind, two open source and widely used programs were selected. They are the Linux kernel and the Mozilla Firefox web browser. These two programs are either partially or totally written in a memory unsafe language like C or C++. The use cases for these selections also have great variance, so they can be viewed as a representation of the range of programs UAFs occur in. Both programs have large user bases and well maintained bug tracking systems. They were also found to have many previously discovered and patched UAF vulnerabilities during a preliminary search.

3.2 Searching for Use-after-free Vulnerabilities

Now that the programs were selected, specific UAF vulnerabilities had to be found within each one. For this study, the selected UAFs were limited to ones that had already been discovered and patched by the development teams. To find the documented UAFs, the publicly accessible bug tracking systems for each program were searched. The bug tracking systems used were the Linux Kernel bug tracker [12] and the Mozilla bug tracker [13]. These are the official bug tracking systems for their respective programs, and they contain kdocumentation and developer discussion on many bugs and vulnerabilities entered into the systems over the years.

The search function of each bug tracking system was extensively used to find known UAF vulnerabilities. The search functions provided many filters and key word options to reduce the results to the desired bug reports. Key words like "UAF," "use-after-free," "dangling pointer," and similar terms were used to find the relevant bug entries. From there, each result that made it through the search filter was investigated to confirm the reported bug was truly a confirmed UAF vulnerability. In some cases, the desired search terms were included in bug reports, but they ended up being incorrect designations by the developers. These reports had later corrections on the true root causes of the bugs, so the bug entries were ignored as they were not actually UAF vulnerabilities.

Another source for finding UAF vulnerabilities in the selected programs was the Common Vulnerabilities and Exposures (CVE) database managed by the MITRE Corporation [14]. This is a publicly accessible database of program vulnerabilities and exploits that have the potential to affect many users. It includes records on numerous programs and has a search function similar to the bug trackers. Search terms like "linux uaf," "firefox uaf," and other variants of these were used to find the largest possible set of UAF-related entries for a selected program. As with the bug trackers, there were false positive results that had nothing to do with the specific program or an actual UAF vulnerability. These had to be sorted through to find entries with confirmed UAFs for the correct program.

Searching the CVE database proved to be one of the best ways to find UAF vulnerabilities for a program. In a CVE entry, it would clearly state that the vulnerability was caused by a UAF memory error for the given program. This made it easy to sort out false positives and verify true UAF vulnerabilities. The entries would also give the possible effects or attacks the exploitation of this vulnerability could lead to. The entries typically had external links to other relevant sources that verified the UAF vulnerability and exploitation effects. These links would lead to sources like the official bug tracker entry for the vulnerability, analysis of the vulnerability by security research groups, and other community discussion. In some cases, links to examples of working exploits for the vulnerability were given. These external links [15, 16, 17, 18, 19] were used along with the information in the main CVE entries to help determine the nature of the UAF vulnerabilities.

Along with the main CVE database, a secondary CVE database [20] was used to confirm exploit information. This database is maintained by the developers of the Ubuntu operating system. In many cases, the information in this database was identical to the main CVE database. In other cases, there would be extra statements that expanded upon the original entry and more clearly explained the risks and possible exploits for the vulnerability.

3.3 Use-after-free Documentation Qualifications

Once the preliminary search was conducted using the previously mentioned sources, there were still many UAF vulnerabilities to choose from for each program. Because of this, a random selection of UAFs was taken from the pool of all UAFs that met the criteria. This was done because enough time had to be allotted to each UAF for analysis, so the total number of UAFs analyzed in this study had to be limited to a reasonable amount. In short, the UAFs selected for this study are a random portion of the total number of UAFs found in the selected programs.

When a UAF was selected out of the possible results for a given program, it had to meet a final set of criteria to be included in this study. As referenced to before, there had to be documented proof the vulnerability was caused by a use-after-free memory error. This meant that either the development team or an authoritative source, like the CVE database, needed to explicitly state the vulnerability was in fact a UAF. The vulnerability could also be confirmed as a UAF if a crash dump file generated by a program like ASan was provided. This file would explicitly state a UAF occurred in the crash report. A verification step like this was needed to avoid using bugs that were incorrectly labeled by the developers or other sources.

Once the selection was verified to be a UAF vulnerability, the next piece of information to find was what attacks could be achieved if the vulnerability was exploited. In this study, the types of attacks that can potentially be done with an exploited UAF vulnerability are referred to as the "exploit effects." If the vulnerability had a CVE entry, the exploit effect or effects were typically given outright in the entry's main description. As mentioned before, the CVE entries typically had links to external sources that could verify these exploit effects.

If instead the UAF did not have a CVE entry, the official bug tracker entry was referenced to find out what exploit effects the program developers thought were possible. In other cases, security researchers had released working examples of how the

UAF could be exploited and listed the resulting effects of those exploits. In all these sources, it was common for more than one exploit effect to be possible for a single UAF vulnerability. After these sources were parsed, the union of all reported exploit effects for a given UAF was recorded. This study chose to use the union because UAFs can be exploited in more ways than one, and it can therefore require more than one source to have a complete view of the exploit effects. The recorded exploit effects for the UAF vulnerabilities include ones that were proven to be possible with working exploit examples and ones that were theorized to be possible but had no publicly available exploit.

Having at least one exploit effect was not an absolute requirement for a UAF to be included in this study. In some cases, the UAF was confirmed to be present but there wasn't a known exploit effect, or the developers believed there was no feasible way for the UAF to be exploited. If this was the case, the UAF was still eligible to be used in the study, but it was categorized under "Unknown Impact" for its exploit effect. They were put in this category to represent a security threat that hasn't been determined. Since UAFs are memory errors that facilitate dangerous attacks, they should always be considered vulnerable bugs when they are exposed to users of the program.

Another requirement for a UAF vulnerability is that it had to have a bug status of "closed," meaning the developers finalized the fix for the vulnerability and have officially closed the issue. In conjunction with this requirement, the UAF documentation also needed to have a link to the patched code that removed the vulnerability. The patch code was used for determining the common ways in which developers fix UAF vulnerabilities. The fixed code versus the original, buggy code also gave insight into how the UAF was introduced in the first place.

Finally, for a UAF to be considered a vulnerability in this study, it had to have the potential to adversely affect users. Some UAFs were only exhibited in debug mode or the testing build, which means it wouldn't be present in the released version of the program. These UAFs were not included in this study since they wouldn't be exposed to an end-user running the release build. If a UAF vulnerability met all these requirements after being randomly selected from the initial searches, it was included in this study. This process was repeated until a reasonable number of vulnerabilities were found for each of the selected programs.

Chapter 4
USE-AFTER-FREE DATA

Utilizing the methods and criteria detailed in chapter 3, UAF vulnerabilities for the Linux kernel and the Mozilla Firefox web browser were found and recorded. In total, 36 UAF vulnerabilities were collected, with 22 recorded from Linux and 14 recorded from Firefox. In the following sections, these vulnerabilities are categorized by the type of exploit effects they exhibited, and the patch code for a select number of Linux UAF vulnerabilities are categorized by the general fix types. The results are organized into two tables. See Appendix A for a complete listing of the UAF vulnerabilities that comprise this data set.

4.1 Use-after-free Exploit Effects

The possible exploit effects for each UAF vulnerability were gathered and compiled into table 4.1. The table lists the program names in the rows and the exploit effects in the columns. For each box, the number of UAFs that exhibited the particular exploit effect is given over the total number of recorded UAFs for that program. For example, a UAF vulnerability in the Linux kernel could allow an attacker to cause a denial of service and an arbitrary code execution attack. In table 4.1, a count would be added to both the "Denial of Service" and "Arbitrary Code Execution" boxes for the Linux kernel row. The final row in table 4.1 gives the total number of UAF vulnerabilities that exhibited a certain exploit effect out of all the UAF vulnerabilities recorded for this study.

	Denial of Service	**Privilege Escalation**	**Sensitive Data Leakage**	**Arbitrary Code Execution**	**Unknown Impact**
Linux Kernel	18 / 22	9 / 22	2 / 22	8 / 22	2 / 22
Mozilla Firefox	11 / 14	0 / 14	0 / 14	8 / 14	2 / 14
Overall	29 / 36	9 / 36	2 / 36	16 / 36	4 / 36

Table 4.1: Use-after-free Exploit Effects

4.2 Linux Kernel Use-after-free Patch Types

After the exploit effects for all the UAF vulnerabilities were determined, the code patches for 13 of the Linux kernel UAF vulnerabilities were analyzed and categorized. These specific UAF vulnerabilities, which all have entries in the CVE database, were selected because they happened to be the first set collected at the time the analysis was conducted. The code patches for each UAF vulnerability were found in the official Linux kernel git repositories [21]. In the left column, table 4.2 lists the CVE-IDs for the 13 Linux kernel UAFs that had their patch code analyzed. The right column lists the general type of patch used to remove the UAF vulnerability. Some were determined to have combinations of patch types.

CVE-ID	**Patch type(s)**
CVE-2007-0772	functionality rewrite
CVE-2009-4141	functionality rewrite; set flags
CVE-2012-2133	functionality rewrite
CVE-2013-7446	functionality rewrite
CVE-2015-1421	add/remove function calls
CVE-2016-0728	add/remove function calls
CVE-2016-8655	functionality rewrite; add lock
CVE-2017-15129	add/remove function calls
CVE-2017-17975	add/remove function calls
CVE-2017-2584	functionality rewrite
CVE-2018-10876	functionality rewrite; set flags
CVE-2018-5873	set flag
CVE-2018-6555	obsolete code removed

Table 4.2: Linux Kernel Use-after-free Patch Types

Chapter 5

ANALYSIS AND OBSERVATIONS

In the following sections, the data tables from chapter 4 will be discussed in detail. Conclusions will be drawn from these results regarding UAF vulnerability exploitation severity and patch complexity.

5.1 Use-after-free Exploit Effect Analysis

In table 4.1, the 36 recorded UAF vulnerabilities are categorized by the exploit effects they exhibited. A given vulnerability could have exhibited one or more of the four common exploit effects or have an unknown security impact. The following subsections will analyze each exploit effect from the table, discuss local versus remote exploitation, and conclude with a summary on the common trends and severity of UAF exploitation observed in this data set.

5.1.1 Denial of Service Attacks

The first and most common exploitation effect observed from the UAF vulnerabilities is a denial of service (DoS) attack. A DoS attack can take many forms, depending on the context and applications that are affected. In general, DoS attacks are a threat to program availability. They allow an attacker to disrupt the operation of a program and make it become unresponsive, stop working, or be otherwise inaccessible to legitimate users. Generally, no lasting damage is caused by a DoS attack. A program or system restart will typically reset the operation state and allow users to regain access, assuming the DoS attack is not still ongoing. However, downtime due to inaccessibility can be very costly to a user or business that depends on constant up-time for their programs and systems. It is also possible that a DoS can cause the loss or corruption of data if the program or system is forced to stop. In this case, a DoS can cause lasting damage if there are no recent backups.

In relation to UAF vulnerabilities, an attacker can cause a DoS by making a dangling pointer read or write (use-after-free) a memory location that is outside the memory region it has permission to access. This results in an access violation or segmentation fault. If this error is not handled by the program, it will be passed to the operating system. The operating system will then force the program execution to end immediately due to this violation. This is otherwise known as a program "crash" [22]. For DoS attacks on the programs used in this study, this means a crash of the entire program for Mozilla Firefox and a system crash for the Linux kernel.

It should be noted that a crash because of a UAF vulnerability is also possible without the intervention of an attacker. The occurrence of a dangling pointer accessing illegal memory can happen through the normal operation of the program or system with a UAF vulnerability present. When the illegal access happens, the uncaught error will crash the program and lead to a denial of service. In this case, the denial of service wasn't caused by the malicious actions of an attacker, but the result is still the same.

In the collected data, 18 out of the 22 (81.8%) UAFs for Linux and 11 out of 14 (78.6%) UAFs for Firefox could cause a DoS. Overall, 29 out of the 36 (80.6%) analyzed UAFs could cause a DoS. This is by far the most common type of exploit effect from the UAFs analyzed in this study. This is not too surprising, as causing a DoS with a UAF vulnerability is the most simplistic attack out of the main types identified in this study. Even though DoS attacks have the potential to not cause any lasting damage, they can still be very disruptive and frustrating to users or cause loss of business in more severe cases. The issue is also exacerbated by the fact that UAF vulnerabilities don't have to be exploited by an attacker to cause a crash, as one can occur through normal program execution when the vulnerability is present. In summary, a DoS is a very common exploit effect of the observed UAF vulnerabilities, and this exploit effect poses a varying threat level to program availability. In specific cases, a DoS also has the ability to affect program data integrity.

5.1.2 Privilege Escalation Attacks

Privilege escalation or escalation of privilege (EoP) attacks are when an attacker increases his or her access control level in the system through some exploit. This is achievable through UAF vulnerabilities because dangling pointers can point to freed but still technically valid data, or data that has been newly allocated to the old location. It is possible this location contains important data that is used by a different part of the program or system that has a higher permission level than a normal user. Through the pointer, attackers can control this object in memory that they shouldn't have access to. This allows an increase in their current privilege level, as attackers are now able to read or write information to places they would normally be denied from. More advanced methods than the one described can also be used to exploit UAFs and gain privileges in different ways.

Privilege escalation is typically used in conjunction with arbitrary code execution to allow an attacker's code to run at a higher privilege level. In some cases, arbitrary code execution combined with EoP can allow attackers to run their own code at root level. EoP is arguably more severe than a DoS, as it compromises the integrity of the privilege levels in a system and allows an attacker to gain finer control over the system or program.

EoP was found to be the third most common exploit effect in this study. It appeared in 9 out of 22 (40.9%) UAF vulnerabilities for the Linux kernel. None of the UAF vulnerabilities analyzed for Mozilla Firefox had records stating an EoP could occur if they were exploited. One explanation for this is since Mozilla Firefox is a web browser, it will normally be run at the same privilege level as the user. It could be difficult for attackers to find ways to escalate their system privileges from within an application that only has user level privileges. However, this does not completely rule out the possibility that an EoP could be achieved in Firefox using an exploited UAF vulnerability.

This contrasts with the Linux kernel, which has many modules that run at the highest permission levels. This means there can be more chances for attackers to exploit a UAF vulnerability that allows them to control restricted data and elevate their permissions. Overall, 25% of the UAFs in this study had the potential to allow an EoP, all of which occurred in the Linux kernel. These results indicate that the likelihood of an EoP is influenced by the type of program the vulnerability occurs in and the permission level it runs at. Developers of kernels and applications that run at root level must keep this in mind when assessing the potential threat of a UAF vulnerability in their system.

5.1.3 Sensitive Data Leakage Attacks

Like the exploitation method described for EoP attacks, an attacker can access freed, but still valid data via a dangling pointer. However, in this next scenario, the attacker is only able to read from a location which contain sensitive information. This is called a sensitive data leakage attack or "Sensitive Data Leakage" in table 4.1. The category of "sensitive data" is broad, as different types of programs handle different types of data, with some being more sensitive than others. Because of the variance in data types, this study considers information "sensitive" if the developers do not intend a regular user to see it. This definition of sensitive data includes information such as kernel memory contents, a different user's credentials, and internal program variables. Internal program variables are sensitive in nature because they provide attackers insight into how a program works, allowing them to further their attack.

From the observed UAF vulnerabilities, only 2 out of 22 (9.1%) for the Linux kernel lead to sensitive data leakage. None from Firefox had this exploit effect. Again, this doesn't rule out the possibility that a UAF vulnerability in Firefox could be used to leak sensitive data. Firefox does contain sensitive data such as saved passwords, which are a high value target for attackers. However, none of the UAFs analyzed for Firefox posed this specific threat.

Overall, only 2 out of the 36 (5.6%) UAF vulnerabilities exhibited a sensitive data leakage threat. The sensitive data leaked in the Linux kernel included kernel memory, which would allow an attacker to view critical system data. Because of the potential to expose important data that is dangerous in the hands of an attacker, data leakage should be considered a severe exploit made possible by UAFs.

5.1.4 Arbitrary Code Execution Attacks

Arbitrary code execution (ACE) is considered the most severe and damaging exploit caused by the observed UAF vulnerabilities. ACE occurs when an attacker exploits a UAF in a certain manner to make the program execution path go to and execute attacker constructed code. At this point, the attacker essentially has free reign over the system, and he or she can execute commands, steal information, and launch more attacks. This becomes especially dangerous when combined with an EoP attack, which allows the attacker's code to run at a higher permission level and bypass the system's permission control.

ACE ended up being the second most common UAF exploit effect seen in this study. 36.4% of the Linux UAF vulnerabilities and 57.1% of the Firefox UAF vulnerabilities were capable of being exploited to allow arbitrary code execution. Overall, 44.4% of the UAF vulnerabilities could be exploited in this way. It was unexpected to find that such a severe exploit like ACE was common in the selected UAF vulnerabilities. ACE being common in the data set could be because it is a high priority goal for attackers. ACE attacks give a significant amount of power to attackers over the exploited program and machine it runs on. This is coupled with the fact that UAF vulnerabilities give attackers control over program memory, which is a favorable starting point for constructing ACE attacks.

When exploited to allow arbitrary code execution, UAF vulnerabilities present a clear danger to the security of the Linux kernel, Mozilla Firefox, and any other program where UAFs occur. The fact that UAFs facilitate this kind of attack is strong evidence for the claim that they are dangerous vulnerabilities that should not be overlooked. Developers must take great care in preventing and mitigating UAF vulnerabilities because of this.

5.1.5 Unknown Security Impact

In a few cases, the analysis of a UAF vulnerability concluded without finding a reliable source that stated its possible exploitation effects. In these cases, the UAF vulnerability was categorized under "Unknown Impact" to represent a security threat that had not been formally determined. Only 2 from the Linux kernel (9.1%) and 2 from Mozilla Firefox (14.3%) were placed in this category. In total, 4 out of the 36 UAF vulnerabilities (11.1%) had an unknown security impact if they were to be exploited.

Based on the data collected for the other exploit effects, it is most likely these UAF vulnerabilities could allow for DoS attacks if exploited. Since DoS attacks were the most common exploit effect, it would not be surprising if these vulnerabilities could also be exploited in this way. However, as stated before, there was no evidence from the sources used to confirm this. Further testing would need to be done on each of these vulnerabilities to confirm their exploit effects.

5.1.6 Local Versus Remote Exploitation

Another analyzed characteristic of the UAF vulnerabilities was whether they could be exploited locally or remotely. "Local" means the vulnerability can only be exploited by an attacker who is a local system user. "Remote" means the attacker can trigger the exploit from across a network without being logged into the system. If a given vulnerability is said to be remotely exploitable, it is more than likely that it can also be exploited by local attackers, as they could create or otherwise simulate network connections to the system they are on. On the other hand, if a vulnerability is said to be locally exploitable, it should not be possible for a network-based attacker to exploit the vulnerability without first having gained local access to the system.

For UAF vulnerabilities in the Linux kernel, it was found that only 2 could be exploited remotely. The rest were found to be only exploitable by local attackers. For Mozilla Firefox, it was more difficult to confirm which UAF vulnerabilities were locally or remotely exploitable. The documentation on these vulnerabilities was sometimes inconsistent on stating whether an exploit could be remotely executed. Despite this difficulty, 8 out of the 14 UAF vulnerabilities in Firefox were confirmed to be remotely exploitable. On top of this, all 8 of these vulnerabilities allowed for ACE attacks, meaning attackers could potentially run whatever code they wanted from across the network via these exploited vulnerabilities. However, 4 other UAF vulnerabilities from Firefox were labeled as only locally exploitable, based on the available information. It is possible that these vulnerabilities could be exploited remotely, but there was not enough information to confirm this. Finally, any vulnerability that had an unknown security impact was not given a remote or local designation. This happened twice for both Linux and Firefox.

Even with the issues in confirming some of the Firefox vulnerabilities, it is clear that Firefox had a higher percentage of remotely exploitable UAF vulnerabilities compared to Linux in this study. This may be attributed to the fact that Firefox is a web browser, so a large percentage of its resources are used to interact with other systems over a network connection. This in turn can expose more vulnerable parts of the application to remote attackers, allowing them to remotely exploit the UAF vulnerabilities. The Linux kernel also has many components that interact with network interfaces, but most of the Linux UAFs in this study happened to occur in parts of the kernel used for local resources. Overall, these results indicate that program or program modules that deal with network connections and have UAF vulnerabilities are more likely to be remotely exploitable.

5.1.7 Conclusions on Exploit Effects

The exploit effects exhibited by the 36 UAF vulnerabilities in this study affected several major areas of program memory and execution, including program code, control-flow, and program data. The most common type of exploit effect was found to be DoS, but the very dangerous ACE was common as well. Along with this, many of these exploit effects could be done remotely in Mozilla Firefox. The exploitation of UAF vulnerabilities can have severe consequences, and they pose a great threat to the security of user programs and kernels. Therefore, program developers must be aware of the range and severity of this threat. They should strive to prevent UAFs from being introduced into their source code and implement extensive methods to detect and patch them out if they occur.

5.2 Linux Kernel Use-after-free Patch Analysis

To analyze the efforts of program developers in removing UAF vulnerabilities, the patch code for 13 out of the 22 Linux kernel UAF vulnerabilities were analyzed and categorized in table 4.2. Each patch for a given UAF vulnerability was put into one or more categories that describes the general type of code changes made to remove the vulnerability. Entries that have more than one patch type mean that more than one distinct method was used to create the patch.

5.2.1 Analysis of Patch Types

Starting with the simplest patch type observed, "obsolete code removed" refers to the developers simply removing the old, vulnerable code from the code base. This entirely removes the UAF vulnerability. For CVE-2018-6555, the code which contained the UAF vulnerability was part of a module that had become obsolete. The developers were coincidentally planning to remove the module anyway, as its functionality was no longer needed.

The next patch type involves correcting the value of module flags ("set flag(s)"). Flags are typically represented by variables like an `int` in Linux kernel modules, and they act as signals to notify other modules if a certain event has occurred, among other uses. If a flag is not set to the correct value at a critical point, as was the case in 3 out of the 13 UAF vulnerabilities, another module may attempt to do something it shouldn't. For example, it might try to access a resource that is not actually available. It is easy to see how this can lead to a use-after-free, such as when a resource is freed in memory, but a flag is not correctly set to reflect the unavailability. In these cases, the fix was typically a simple one-line addition to correctly set the flag for a previously unaccounted program state. This simple fix indicates the developers had initially overlooked an edge case, as opposed to creating an inherently flawed implementation that would have taken much more effort to fix.

Another patch type deals with a UAF vulnerability that was caused by a race condition. Among other methods, race conditions can be prevented by adding a lock on the resource under contention. This was the case for only 1 out of the 13 vulnerabilities (CVE-2016-8655). Race conditions can be difficult to discover, but once they are found, the actual implementation of a lock is simple in comparison. This makes the patch relatively easy to implement. Race conditions like this are another example of kernel developers not accounting for edge cases in a module's operation.

The "add/remove function calls" patch type involves slightly more code changes than the previous patches. In 4 out of the 13 vulnerabilities, it was determined that an incorrect function call or calls were the cause of the UAF. This was either due to an incorrect use of one function call over another, a missing function call that should have been present, or a function call that was used when it shouldn't have. Any one of these combinations can result in data being freed or accessed when it isn't available, resulting in a UAF vulnerability. The average patch of this kind involved either removing or adding the correct function calls, which required only a few line changes. This type of patch indicates that the UAF vulnerabilities were introduced because certain cases in the incorrectly used or forgotten functions were overlooked. Since no major rewrites were required to remove these vulnerabilities, this wouldn't be considered a major design flaw.

The final patch type, "functionality rewrite," required the most code changes to implement. 7 out of the 13 UAF vulnerabilities analyzed required this type of patch to remove the vulnerability. In these cases, it was determined that there was a significant flaw in the implementation of a module's functionality. This flaw caused the UAF vulnerability to occur, and the only way to remove it was by rewriting a significant portion of the relevant code. In over half of these instances, approximately 30 total insertions and deletions were needed to complete the rewrite. This included adding new functions that corrected the implementation and the removal of flawed functions.

In other cases, over 100 total additions and removals were made across one or more source code files (CVE-2009-4141 and CVE-2013-7446). In the worst case, over 200 total code additions and removals were needed to remove the vulnerability and fix the implementation (CVE-2012-2133).

In most of the patches analyzed, there were more additions made to the source code than removals. This indicates that the buggy implementations were lacking features needed to prevent the UAF vulnerabilities from occurring. This is opposed to having unnecessary and buggy code that needed to be simplified. A final important observation is that functionality rewrites were the most common patch type of the examined UAF vulnerabilities. This indicates that UAF vulnerabilities are likely to require more substantial code changes to remove them, as opposed to short and simple fixes.

5.2.2 Conclusions on Use-after-free Patch Analysis

The results of these analyzed patches reveal two common trends. The first is that the amount of work that goes into patching UAF vulnerabilities can vary greatly. A patch can range from a single line fix to over 200 combined additions and removals across several source code files. Second, these results show that UAF vulnerabilities can arise from not only complex implementations with subtle flaws, but also from edge cases that were simply unaccounted for. This reveals that even a simple mistake can lead to a dangerous UAF vulnerability with accompanying exploits. These results indicate that developers of the Linux kernel and other software must be prepared to potentially dedicate a large amount of time to fully remove UAF vulnerabilities from their source code. They also need to be aware that even small code changes can cause UAF vulnerabilities to occur if edge cases are not accounted for.

Chapter 6
FUTURE WORK

There are two main avenues of future research that can be pursued using the data and analysis from this study. The first is a continuation of the empirical study on UAF vulnerabilities in commonly used software. This new study would increase the scope of programs and the number of UAF vulnerabilities to be analyzed. The second direction of research involves using the UAF vulnerability analysis data to develop a new program fuzzer that can discover UAFs more effectively than current fuzzers. A reasonable approach would be to first conduct the expanded empirical study to gain more base knowledge and identify patterns on how UAF vulnerabilities are commonly introduced. Then the specialized UAF fuzzer can be developed based on these patterns to more effectively test for UAFs. Below are sections that detail how the two paths would be carried out as future research on the nature of UAF vulnerabilities.

6.1 Expanded Empirical Study

There are many areas of this empirical study that can be expanded upon to create a follow-up study. The first area would involve analyzing more UAFs from a wider variety of programs. This would help solidify the patterns seen in this study with more evidence. The expanded study would include the analysis of more UAFs from the Linux kernel and Mozilla Firefox as well. Exploit effects and patches for the new vulnerabilities would be analyzed and categorized in a similar manner to this current study. Lastly, a new aspect of UAF vulnerabilities that could be investigated is whether there are any differences in the way UAFs are introduced in kernels compared to user-level applications.

6.2 Developing a Specialized Fuzzer

As mentioned in chapter 2.4.2, fuzzers are a useful tool to test for UAFs. However, UAF vulnerabilities still occur in programs, including ones that make use of this technology. It is clear that new developments must be made in this area. A future avenue of research is to build a fuzzer that can better detect UAFs. By using the common patterns seen in UAF vulnerability patches and recognizing how UAFs are typically introduced, a systematic method of fuzzing for UAFs can be developed. This would also involve studying the ways state-of-the-art fuzzers discover UAFs and identifying their shortcomings. When this new fuzzer is created, it will be tested to see if it is more effective at finding known instances of UAFs and whether it can find previously unknown UAFs in user-level programs and kernels.

Chapter 7
CONCLUSION

This study analyzed 36 real-world use-after-free vulnerabilities that occurred in the Linux kernel and Mozilla Firefox web browser. The analysis of these vulnerabilities revealed that attackers can exploit them to cause denials of service, escalate their privileges, leak sensitive data, and execute arbitrary code on a victim's machine. It was found that 80.6% of the analyzed UAF vulnerabilities could allow for a denial of service attack and 44.4% could allow for arbitrary code execution. The majority of the UAF vulnerabilities analyzed for Mozilla Firefox were also found to be remotely exploitable by attackers.

This study then analyzed a selection of 13 UAF vulnerability patches for the Linux kernel and found that a significant amount of work can be required to fully remove UAF vulnerabilities. In one case, over 200 combined additions and removals across several source code files were needed to remove the vulnerability. Finally, the patch analysis revealed that UAF vulnerabilities can be introduced in many ways, including complex implementations with subtle flaws and unaccounted edge cases.

Based on the results of this study, it is evident that UAF vulnerabilities pose a severe threat to programs written in languages that are not memory safe. It is also apparent that better tools for finding UAF vulnerabilities need to be developed for the effective detection and prevention of this security threat. It is hoped that the results of this study will lead to future studies on UAF vulnerabilities and the development of more effective testing tools.

REFERENCES

[1] The MITRE Corporation, “Cwe-416: Use after free.” https://cwe.mitre.org/data/definitions/416.html, June 2019. Accessed: 2019-7-1.

[2] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *2013 IEEE Symposium on Security and Privacy*, pp. 48–62, May 2013.

[3] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “Addresssanitizer: A fast address sanity checker,” in *USENIX ATC 2012*, 2012.

[4] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, “Preventing use-after-free with dangling pointers nullification,” in *2015 Network and Distributed System Security Symposium*, NDSS ’15, Feb. 2015.

[5] T. H. Dang, P. Maniatis, and D. Wagner, “Oscar: A practical page-permissions-based scheme for thwarting dangling pointers,” in *26th USENIX Security Symposium*, USENIX Security ’17, pp. 815–832, USENIX Association, 2017.

[6] D. Liu, M. Zhang, and H. Wang, “A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’18, pp. 1635–1648, ACM, 2018.

[7] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating fuzz testing,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS ’18, pp. 2123–2138, ACM, 2018.

[8] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of unix utilities,” *Communications of the ACM*, vol. 33, pp. 32–44, Dec. 1990.

[9] M. Zalewski, “american fuzzy lop.” http://lcamtuf.coredump.cx/afl/, Nov. 2017. Accessed: 2019-7-31.

[10] S. Lu, S. Park, E. Seo, and Y. Zhou, “Learning from mistakes: A comprehensive study on real world concurrency bug characteristics,” in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pp. 329–339, ACM, 2008.

[11] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, “Understanding and detecting real-world performance bugs,” in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’12, pp. 77–88, ACM, 2012.

[12] Linux Kernel Organization, Inc., “Kernel.org bugzilla main page.” https://bugzilla.kernel.org/, 2019. Accessed: 2019-2-7.

[13] Mozilla Foundation, “Bugzilla main page.” https://bugzilla.mozilla.org, 2019. Accessed: 2019-2-11.

[14] The MITRE Corporation, “Search cve list.” https://cve.mitre.org/cve/search_cve_list.html, Jan. 2019. Accessed: 2019-7-24.

[15] J. Spaans, “The linux kernel mailing list archive.” https://lkml.org, 2018. Accessed: 2019-1-27.

[16] Red Hat, Inc., “Red hat bugzilla main page.” https://bugzilla.redhat.com/, 2019. Accessed: 2019-1-22.

[17] Perception Point Research Team, “Analysis and exploitation of a linux kernel vulnerability.” https://perception-point.io/resources/research/analysis-and-exploitation-of-a-linux-kernel-vulnerability/, Mar. 2018. Accessed: 2019-1-4.

[18] Google LLC, “Android security bulletin - july 2018.” https://source.android.com/security/bulletin/2018-07-01, July 2018. Accessed: 2019-7-22.

[19] Mozilla Foundation, “Mozilla foundation security advisories.” https://www.mozilla.org/en-US/security/advisories/, 2019. Accessed: 2019-7-31.

[20] Canonical Ltd., “Ubuntu cve tracker.” https://people.canonical.com/~ubuntu-security/cve/, July 2019. Accessed: 2019-7-24.

[21] Linux Kernel Organization, Inc., “Kernel.org git repositories.” https://git.kernel.org/, 2019. Accessed: 2019-7-19.

[22] MSDN Archive, “Why does software crash 1 - the access violation.” https://blogs.msdn.microsoft.com/chappell/2005/01/12/why-does-software-crash-1-the-access-violation/, Jan. 2005. Accessed: 2019-8-2.

Appendix

USE-AFTER-FREE VULNERABILITY REFERENCE TABLES

This appendix contains tables for all the UAFs used in this empirical study. For each program, the bug tracker ID assigned to the UAF by its development team is listed. If a UAF vulnerability has a corresponding entry in the CVE database, then the "CVE-ID" is listed instead of the bug tracker ID. Each entry has the exploit effects that are possible for an attacker to create using the vulnerability: Denial of Service (DoS), Escalation of Privilege (EoP), Sensitive Data Leakage (SDL), Arbitrary Code Execution (ACE), and Unknown Impact (UI). The final character in an entry states whether the UAF vulnerability can be exploited locally (L) or remotely (R). Local or remote exploitation is given only when an exploit effect is known.

UAFs for the Linux kernel were found with the official Linux kernel bug tracker [12], the CVE database [14], and a secondary CVE database run by the developers of Ubuntu [20]. Other sources used to confirm these UAFs are: [15, 16, 17, 18].

Bug 10050	UI		CVE-2015-1421	DoS, EoP; R
Bug 59371	DoS; L		CVE-2016-0728	DoS, EoP, ACE; L
Bug 188941	UI		CVE-2016-8655	DoS, EoP, ACE; L
Bug 198295	DoS; L		CVE-2017-15129	DoS, EoP, ACE; L
Bug 199443	DoS; L		CVE-2017-17975	DoS, ACE; L
Bug 199839	DoS; L		CVE-2017-2584	DoS, SDL; L
Bug 200179	DoS; L		CVE-2018-5873	EoP; L
CVE-2007-0772	DoS; R		CVE-2018-6555	DoS, ACE; L
CVE-2009-4141	EoP; L		CVE-2018-10876	DoS, ACE; L
CVE-2012-2133	DoS, EoP; L		CVE-2018-10879	DoS, ACE; L
CVE-2013-7446	DoS, EoP, SDL; L		CVE-2019-8912	DoS, EoP, ACE; L

Table A.1: Linux Kernel Use-after-frees and Exploit Effects

UAFs for the Mozilla Firefox web browser were found with the official Mozilla bug tracker [13], the CVE database [14], a secondary CVE database run by the developers of Ubuntu [20], and the Mozilla Foundation Security Advisories [19].

Bug 1033006	UI		CVE-2013-5600	DoS, ACE; R
Bug 1161332	DoS; L		CVE-2014-1537	DoS, ACE; R
Bug 1273678	UI		CVE-2014-1592	DoS, ACE; R
CVE-2010-0183	DoS, ACE; R		CVE-2016-5276	DoS, ACE; R
CVE-2011-0065	ACE; R		CVE-2017-5434	DoS; L
CVE-2012-1958	DoS, ACE; R		CVE-2017-7793	DoS; L
CVE-2013-0766	DoS, ACE; R		CVE-2018-5154	DoS; L

Table A.2: Mozilla Firefox Use-after-frees and Exploit Effects