Published in the *32nd International Conference on Software Engineering & Knowledge Engineering (SEKE)*, 2020, pp. 538–543.

Characterizing Vulnerabilities in a Major Linux Distribution

Stephen R. Tate Moulika Bollinadi Joshua Moore

Department of Computer Science, UNC Greensboro, Greensboro, NC 27402
Contact email: srtate@uncg.edu

Abstract

This paper reports on a careful study of vulnerabilities in open-source software, performing both a longitudinal study over 7 years of data and an in-depth exploration of a particular type of vulnerability. First, data was mined from Ubuntu security notices from 2012 to 2019, specifically pulling security notices published within the first year of each of the four stable releases during that time. This provided a dataset covering 3,232 security vulnerabilities, which were cross-referenced with other information, allowing us to identify trends in types of vulnerabilities over the past 7 years. Within these results, we see that out-of-bounds memory access (which includes the classic "buffer overflow" vulnerability) has consistently been the most pernicious security weakness, so in the second part of this research we performed an in-depth study of a random sample of 30 recent out-of-bounds access vulnerabilities. Beginning by evaluating each vulnerability in terms of seven features, we identified trends and patterns and expanded the analysis to a total of eleven features. These results help further understanding of how out-of-bounds access vulnerabilities occur in real software, which can help both researchers looking to improve tools for vulnerability analysis and developers learning how to avoid common pitfalls.

1 Introduction

The enthusiasm of programmers to create new software has led to an explosion in the size and complexity of programs that has greatly exceeded our ability to ensure that the programs are free of errors. Of particular interest in our highly-connected society are errors that can be exploited to subvert security goals, a kind of error that is called a "vulnerability." In this project, we gain insight into what kinds of vulnerabilities occur across a wide range of open-source software that is used in Linux and Unix-like operating systems, how the frequency of various vulnerability types has changed over time, and what specific characteristics are present in the most pernicious type of vulnerability, the out-of-bounds access vulnerability. Our goal is to improve knowledge about vulnerabilities in real-world software, focusing on situations in which sourcecode is available so that patterns and trends can provide useful information for software developers.

Vulnerabilities are cataloged and classified in the "National Vulnerability Database," or NVD, maintained by the National Institute of Standards and Technology (NIST) [6]. While statistics on the entire NVD database are available, including a break-down by type of vulnerability [7], these statistics aggregate vulnerabilities across all environments, including open-source software, closed-source and proprietary software, and dedicated devices and device firmware. Since our primary motivation for this work is to find information that will be useful in settings with access to sourcecode (specifically, source-based analysis tools and developer practices), we look specifically at vulnerabilities where sourcecode is published so it can be analyzed for patterns and characteristics.

To study open-source software, we need to identify a set of projects that we can study, and from which we can extract useful information about vulnerabilities. We initially considered mining information from GitHub [4] or from the Software Heritage Graph dataset [8], but the range of software maturity and the *ad hoc* nature of vulnerability identification in these sources quickly showed this to be an unproductive approach. We then looked to major Linux distributions, since distribution maintainers have already identified interesting and mature projects for inclusion in the distribution, and they track security vulnerabilities so that their users can stay up-to-date with security patches. Exploring three major distributions, Debian [13], RedHat [10], and Ubuntu [3], we found Ubuntu to be the easiest and most effective to work with for three main reasons: the Ubuntu maintainers track a huge collection of open-source software, with over 29,000 source packages in the 18.04 release; there is a well-maintained CVE tracker and source of security notifications (the Ubuntu Security notices, or USNs); and the practice of producing stable long-term-

DOI Reference number: 10.18293/SEKE2020-033

support (LTS) releases on precise two-year intervals provides a meaningful way to perform a longitudinal study. In the end we selected four such LTS releases, 12.04, 14.04, 16.04, and 18.04, and studied both general changes between releases and specific vulnerability characteristics from the most recent release. While we use data from Ubuntu, all major Linux distributions use the same base of open-source software, so we believe these results are representative of the broader open-source software community.

Contributions: We make the following contributions in this paper.

- We provide a clear picture of how the prevalence of various types of vulnerabilities has changed over time. The data show quantitatively that out-of-bounds memory access has remained the most pernicious type of security vulnerability over time, consistently accounting for around a third of all security vulnerabilities.
- We report on a careful study of a random sample of 30 out-of-bounds memory access vulnerabilities, defining the notion of an "exploit flow" to capture and analyze features in the sourcecode that characterize each exploit. Our statistical breakdown reveals several interesting and useful observations, including the pervasiveness of exploit flows that cross compilation-unit boundaries (implying that analysis tools must similarly consider multi-module analysis) and the importance of pre and post condition statements at function boundaries to simplify understanding of vulnerabilities.

We believe that the results reported here will be beneficial both for researchers working to improve source-based vulnerability identification tools, and for developers learning about the types of security-related errors made in real-world software systems.

2 Types Vulnerabilities Over Time

In order to study how vulnerabilities have changed over time, we considered the four most recent Ubuntu Long-Term Support (LTS) releases, which were released two years apart in April, starting in 2012 with release 12.04. We processed archived Ubuntu Security Notices (USNs), extending exactly one year after each initial release, extracting CVE references and information on "affected distributions" from each security notice to get CVE/distribution pairs.

After manually examining some of these results, we discovered that there were a number of CVEs listed with distributions that were not actually affected by that CVE. The problem is that a USN may reference a large number of CVEs, and the USN flags a distribution as "affected" if any one of those is a vulnerability in the distribution. In one extreme case, a single USN (USN 3681-1) referenced 124 different CVEs! This leaves the possibility of a large number of CVEs in a USN not being relevant to that distribution. To correct this problem, we pulled information on all CVEs from the Ubuntu "CVE Tracker," which indicates which distribution(s) are affected by an individual CVE, and dropped a CVE/distribution pair if the CVE tracker has a distribution classification for that CVE as "DNE" (does not exist), "not affected," or "ignore." After processing, there was a modest drop in CVE/distribution pairs, with the number of CVEs affecting release 18.04 dropping from 911 (based on USN alone) to 843 (based on both USN and the more accurate CVE tracker classification).

For each CVE that affects an Ubuntu distribution (and was fixed within the first year of that distribution's release), we cross-referenced the CVE with the corresponding entry in NIST's National Vulnerability Database (NVD) to extract the severity of the vulnerability, as given by the CVSS v3.0 base score, and the type of vulnerability, as given by the Common Weakness Enumeration (CWE) code associated with the CVE. While there are 839 defined CWE codes, only 62 appear in our dataset, and these can further be categorized into a few broad classes of vulnerability types. Our starting-point was the CWE cluster definitions from MITRE, such as CWE-970 ("SFP Secondary Cluster: Faulty Buffer Access") that lists 11 fine-grained CWEs related to buffer access. CWEs that were not listed in such pre-defined clusters were examined for obvious matches, such as CWE-787 (Out-of-bounds Write). In the end, the 7 CWEs that occurred in our data set from the CWE-970 cluster along with CWE-787 formed our "Out-of-Bounds vulnerability" class, as summarized in Table 2. Occurrences of each vulnerability type in the four Ubuntu releases was collated, and information on the "top 5" most prevalent vulnerability types is given in Table 1. Full data, including full CWE-to-class mapping, occurrence rates for each class, as well as the scripts used to analyze the Ubuntu data, is available at the project web site [12].

Discussion: As can be seen in Table 1, the number of vulnerabilities has been generally increasing, with an anomalous spike at version 16.04. However the number of high or critical severity vulnerabilities has decreased, resulting in the percentage of vulnerabilities classified as high or critical severity decreasing by over half (from 44% to 19%). CVE evaluation has dramatically improved over the years, with the percentage of CVEs giving a CWE classification increasing 64% to 95%. The Out-of-Bounds access vulnerability has been the top vulnerability type in every version. This is particularly frustrating as it is one of the oldest forms of security vulnerability and a significant amount of research has gone into locating and fixing such vulnerabilities. Furthermore, pointer issues, also a memory safety violation, have also gotten much more common in the past few years. Of issues that are not related to memory safety, per-

Distribution Release date	Ubuntu 12.04 *April 26, 2012*	Ubuntu 14.04 *April 17, 2014*	Ubuntu 16.04 *April 21, 2016*	Ubuntu 18.04 *April 26, 2018*
Total CVEs fixed in Year 1	646	701	1042	843
High/Critical severity	287 (44%)	268 (38%)	258 (25%)	156 (19%)
CWE classified	415 (64%)	455 (65%)	952 (91%)	802 (95%)
Out-of-bounds access	24.3% (1)	30.8% (1)	38.9% (1)	30.2% (1)
Permissions	18.1% (3)	10.3% (3)	9.5% (3)	15.5% (2)
Pointer issues	—	2.6% (10)	8.8% (4)	11.0% (3)
Input validation	16.6% (4)	14.3% (2)	13.1% (2)	10.8% (4)
Resource management	21.9% (2)	9.0% (4)	4.8% (7)	8.5% (5)
Numeric errors	7.0% (5)	8.1% (5)	1.7% (9)	0.2% (19)

Table 1. Vulnerabilities by distribution (rank of each type in parenthesis)

CWEs in the general "out-of-bounds access" class	
CWE-118	Incorrect Access of Indexable Resource ('Range Error')
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer
CWE-122	Heap-based Buffer Overflow
CWE-123	Write-what-where Condition
CWE-125	Out-of-bounds Read
CWE-126	Buffer Over-read
CWE-129	Improper Validation of Array Index
CWE-787	Out-of-bounds Write

Table 2. Out-of-bounds access CWEs

mission and input validation issues remained in the top four vulnerability types throughout the seven years, although the percentage of input validation issues has been slowly declining. Numeric errors have also fallen dramatically over the years. Some of the changes in pointer issues and numeric errors could be due to a change in classification criteria (e.g., an integer overflow leading to out-of-bounds access might have been classified as an integer overflow in the past, but is now classified as out-of-bounds access), but digging deeper into the CVE classification criteria is beyond the scope of this project and is left as an open question.

3 Out-of-bounds Access Vulnerabilities

Given the consistent top ranking of Out-of-Bounds access vulnerabilities (abbreviated as "OoB access"), we next undertook a study to gain deeper understanding into how these vulnerabilities manifest in real-world systems. As a first step, we pulled the full list of OoB access CVEs that affected Ubuntu 18.04 in its first year of release, and randomly sorted them so that we could select the first CVEs in our random ordering as a random sample of OoB access vulnerabilities. Early in our process we discovered that the rapid release model of Firefox and family (Thunderbird, mozjs, etc.) differed greatly from other packages, and the huge patch updates made it practically impossible to locate specific vulnerabilities based on public information. For example, the fix for CVE-2018-18493 (a critical-severity OoB vulnerability) was only included as part of the Firefox update from version 63.0.3 to 64, where the diff between these versions contains close to 1.9 million lines. As a result, we excluded vulnerabilities in these packages from our list before taking our random sample.

We started our evaluation based on 7 characteristics suggested by our past experience. The authors studied 14 CVEs over a period of 6 weeks, with group discussions that identified 4 additional recurring patterns, giving a final set of 11 relevant characteristics. The 14 initial vulnerabilities were re-examined along with 16 others. Of these 30 vulnerabilities, six were excluded for various reasons: three did not apply to Ubuntu 18.04, two were mis-classified in the NVD as out-of-bounds access vulnerabilities, and one did not have enough public information available to analyze. While this process gave deep insight into vulnerability characteristics, it was highly labor-intensive, and an interesting future research direction could explore ways to automate or at least provide tools to assist in this analysis.

3.1 Exploit Flow Definition

To characterize out-of-bounds access vulnerabilities, we introduce the idea of an *exploit flow*: the shortest execution path through the program that fully explains to an informed reader how the vulnerability arises and what causes the out-of bounds access, where we allow irrelevant portions of the flow to be redacted. This definition is inherently subjective, with reference to "an informed reader," but in general it will include code that calculates an array index or a pointer that is subsequently used in the out-of-bounds access. When the out-of-bounds access involves a dynamically-sized block of memory, the exploit flow will also typically include the size

```
5658    first_object=(p[0] << 8) | p[1];
5659    last_object=(p[2] << 8) | p[3];
5660    p+=4;
5661
5662    for (i=(int) first_object;
             i <= (int) last_object; i++)
5663    {
5664      if (mng_info->exists[i] &&
              !mng_info->frozen[i])
5665        {
5666        ...
```

Figure 1. Vulnerability CVE-2017-13139

calculation and memory allocation.

An example of an out-of-bounds access vulnerability is shown in Figure 1, where the code comes from PNG image processing module for the ImageMagick library (code has been somewhat reformatted to fit in one column, but is otherwise directly taken from the `coders/png.c` file of ImageMagick version 6.9.7-4). Note, this was actually one of the vulnerabilities that ended up being excluded from our study, since it was patched prior to the official Ubuntu 18.04 release; however, due to its simplicity it serves as the best example to explain exploit flows in this paper.

The exploit flow is the following description, which traces an execution through the code to demonstrate clearly how the vulnerability can be exploited:

1. *Comment:* Variable `p` points to an input buffer containing unsigned characters, read directly from (possibly malicious) input.
2. *Lines 5658–5659:* 16-bit binary values are loaded in to unsigned int variables `first_object` and `second_object`, and are unchecked so they can be any possible 16-bit values (e.g., these variables can have values 1000 and 2000, respectively).
3. *Line 5662:* A for loop starts on line 5662, with index `i` ranging from `first_object` to `last_object`.
4. *Line 5664:* Array `mng_info->exists[i]` is accessed, which is a statically-sized array of size 256.
5. *Out-of-bounds access, line 5664:* With the sample values above, on the first iteration of the for loop, `i` is 1000 and when used as an index into an array of size 256 this results in an out-of-bounds read.

There are a few important observations to make from this example. First, it is not entirely self-contained, since the first comment simply mentions that buffer `p` is unconstrained data read from the user. This should be perfectly clear to the "informed reader" that is part of the exploit flow definition, and allows us to leave out sometimes-confusing I/O buffering code from the flow. Second, we include specific example values that could be used in a proof-of-concept exploit for the vulnerability — in all but a few cases, we have constructed actual proof-of-concept inputs to test our exploit flows. Finally, we do not include code that defines sizes of memory blocks in the exploit flow when they follow from static type declarations, and instead simply state the size of the data block (as we did with the 256-entry array in the example exploit flow).

In terms of vulnerability characteristics, this exploit flow is a very simple example of a pattern we saw regularly in out-of-bounds access vulnerabilities: An offset into a binary structure is read from user data, unpacked from a straight binary value (typically 16 or 32 bits), and then used without being checked to ensure that it is a sensible value.

3.2 Exploit Flow Characteristics

As described above, we evaluated all vulnerabilities in terms of 11 characteristics or features. In this section we define and describe these characteristics.

Spans multiple compilation units or files: Does the exploit flow include code from multiple compilation units, where "compilation unit" is the unit of source code processed by one pass of a compiler? For C and C++, a compilation unit consists of both a main source file (`.c` or `.cpp` file) and any included header files (`.h` files). This characteristic is important when considering code analysis tools, as some tools (e.g., the Clang static analyzer [5]) restrict analysis to a single compilation unit, whereas others (e.g., Infer [2] and Klee [1]) perform analysis using information from multiple compilation units.

Spans multiple functions: Does the exploit flow include code from multiple functions, where these functions may be library functions, user-defined functions, or both? Unlike the previous characteristic, functions may be in the same file or compilation unit. This characteristic is important while considering code analysis tools as some tools do not trace across function call boundaries.

Involves typecasting or type confusion: Does critical data change types during the exploit flow? Code that initializes a variable using one type and converts it into another type is type casting. If that resource is accessed using an incompatible type with the original type then it is type confusion. This can result in triggering logical errors in the source code.

Simplified with a function pre/post condition: Would a stated function pre/post condition shorten or simplify the exploit flow? Given the frequency of exploit flows that span multiple functions, it is not surprising that such conditions could help analysis. In particular, a function that fills a block of memory could have a pre-condition that the buffer must have sufficient size, and then the exploit flow

does not need to extend into the function. A static analysis tool could use such pre/post conditions to check both that pre-conditions are met when a function is called, and to start path analysis in the function using the pre-condition as a starting condition.

Simplified with a data structure invariant: Would a stated data structure invariant shorten or simplify the exploit flow? The most common way that this is relevant in our study would be an invariant that relates to the size of a dynamically allocated buffer, so that the buffer size is known to a static analyzer without requiring the exploit flow to trace all the way back to the actual size calculation and allocation.

Dynamically-sized memory block: Are memory needs determined at runtime, such as when they are dependent on user input? Dynamic allocation poses a particular problem for static analysis tools, as the tool often does not have any information about the size of the memory block (this is related to, and potentially addressed by, a data structure invariant as mentioned above). In addition, with dynamic memory allocation there is a possibility that the user/attacker introduces a large value and the system cannot allocate enough memory for it, so no buffer at all is allocated. While programs can detect this situation by checking the allocation return value, it is unfortunately common that programmers omit this error-check.

Binary data format processing: Does the data being handled in the exploit flow come from a raw binary data format, such as image or audio files? Vulnerabilities can arise from code that unpacks and uses values such as sizes or offsets without first checking them for validity. While legitimate data in such formats is produced by software that will only output data conforming to certain rules, attacker-supplied data is not restricted to sensible values.

Other characteristics: We evaluated all vulnerabilities with respect to several other characteristics which ended up being less significant. In particular, we considered whether the exploit flow was asynchronous, as might be common in event-driven programming; whether bugs were involved in parsing textual input; whether there are a significant number of branching decisions (including loop iterations), which would lead to path explosion in analysis; and whether the exploit flow is determined by dynamic type resolution, such as through a method dispatch table in C++. Any of these characteristics would complicate the task of code analysis, but all turned out to be rare in our random sample of OoB vulnerabilities. This is encouraging because it implies that significant improvements in static analysis can be made without having to address these particularly challenging situations. For example, only one CVE had an asynchronous exploit flow, and most had very limited branching (some included loops, but the vulnerability could typically be triggered on the first iteration of the loop).

OoB Access Characteristics		
Multiple Functions	17/24	71%
Multiple Compilation Units	12/24	50%
Type confusion/casting	9/24	38%
Simplifying Pre/Post-condition	15/24	63%
Simplifying Invariant	10/24	42%
Dynamically-sized memory	19/24	79%
Binary data formats	19/24	79%

Table 3. Out-of-Bounds Characteristics

As we investigated characteristics of exploit flows, another interesting fact emerged: the prominence of fuzz testing in detecting vulnerabilities. The method used to detect the vulnerability was clearly stated in slightly over half of the studied CVEs, and in every single one of those cases the vulnerability was found by fuzzing (typically using AFL [14] or a variant). In several other cases, while not stated explicitly there was evidence that fuzzing was used to locate the vulnerability, and in only a single case did the evidence suggest that the vulnerability was found in some other way (probably manual code review). This does raise an interesting question, which would require further study to resolve: Is improved success in finding OoB vulnerabilities masking a decline in frequency of occurrence? In other words, how much of the OoB frequency in Table 1 is due to actual prevalence of the type of vulnerability and how much is due to our success in locating such vulnerabilities? The widespread use of fuzzing could also explain the predominance of vulnerabilities in binary file format processing, as fuzzing works particularly well on such data.

3.3 Results

The prevalence of the various vulnerability characteristics described above, across the 24 usable samples, is given in Table 3. The frequency of exploit flows that cross function and even compilation-unit boundaries clearly demonstrates that effective analysis tools must be able to reason about multi-unit execution paths. However, a promising aspect of this analysis is that a large number of such flows (roughly two-thirds of the multi-function exploit flows) could be greatly simplified through proper use of function pre/post conditions. Programmers naturally divide programs into pieces with clear logical requirements so that they can cope with complexity as a developer. Making these logical requirements explicit could significantly help analysis in most of the OoB vulnerabilities that we studied.

Furthermore, dynamic memory allocation is very common in modern software, and our results show that it also plays an important role in many vulnerabilities. Again,

buffer sizes are generally logically designed by programmers, even if such sizes are not available (in C and C++) to static analysis tools. Taking this logical design out of the mind of the programmer and turning it into an explicitly-stated data structure invariant could help identify vulnerabilities in almost half of the vulnerabilities studied.

Finally, type confusion and casting is a less common but still significant problem. Casting issues, such as from an unsigned size to a signed size, could be easily identified by even simple static analysis tools. As a next step in our work, we will study how often such casting issues occur and how often they lead to vulnerabilities. We suspect that the number of safe uses of typecasting far exceeds the unsafe uses, which would lead such a static analysis tool to produce too many false positives to be useful. Analysis of how to identify just the unsafe uses is an interesting open question.

4 Related Work

Rigorous work classifying vulnerabilities across a wide span of open-source software is uncommon, but a few recent projects are related to our work. Ponta *et al.* manually curated a large collection of 624 vulnerabilities and associated patches in open-source Java packages related to their area of interest [9]. As their work focuses specifically on Java code in a niche area, it does not provide the broad picture of open-source vulnerabilities that our study seeks.

In a similar effort to ours, reviewing reported buffer-overflow vulnerabilities, Shuckert *et al.* performed a review of such vulnerabilities that had been reported in the Firefox web browser [11]. This study provides some interesting observations, but results are reported in a more qualitative than quantitative fashion, with no statistics on the prevalence of various vulnerability characteristics reported. Furthermore, focusing on a single software package provides excellent insight into that package, but it is unclear how much the results reflect overall open-source development characteristics as opposed to particular coding practices and standards for that particular development team.

Other work on buffer-overflow vulnerabilities considers, as part of the background to their work, investigation of certain characteristics related to their work, but these are not broadly-focused studies that provide insight into bigger picture of open-source vulnerabilities.

5 Conclusion

In this paper, we examined characteristics of real-world vulnerabilities in open-source software. Mining a large set of 3,232 vulnerabilities over 7 years revealed several interesting trends, and clearly showed that out-of-bounds access vulnerabilities have remained the most commonly-occurring danger. With an eye toward identifying promising directions for program analysis tools and techniques, we carefully studied a random sample of out-of-bounds access vulnerabilities, identifying several particularly promising directions for future work. First, there is a strong need for analysis that spans function or even compilation unit boundaries. Second, the use of pre/post conditions and data structure invariants, perhaps provided to analysis tools through sourcecode annotations, could greatly simply the reasoning required to identify vulnerabilities. And finally, exploring how tools can distinguish between safe and unsafe uses of typecasting could produce interesting and practical results.

References

[1] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), OSDI'08, pp. 209–224.

[2] CALCAGNO, C., AND DISTEFANO, D. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods*. 2011, pp. 459–465.

[3] CANONICAL, LTD. Ubuntu website. `https://ubuntu.com/`.

[4] GITHUB, INC. Github website. `https://github.com/`. Accessed: 2020-01-30.

[5] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization* (2004).

[6] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. National vulnerability database. `https://nvd.nist.gov/`.

[7] ÖZKAN, S. CVE Details website. `https://www.cvedetails.com/`. Accessed: 2020-01-30.

[8] PIETRI, A., SPINELLIS, D., AND ZACCHIROLI, S. The Software Heritage Graph Dataset: Public Software Development under One Roof. In *Proc. of the 16th International Conference on Mining Software Repositories* (2019), MSR '19, pp. 138–142.

[9] PONTA, S. E., PLATE, H., SABETTA, A., BEZZI, M., AND DANGREMONT, C. A Manually-curated Dataset of Fixes to Vulnerabilities of Open-source Software. In *Proc. of the 16th International Conference on Mining Software Repositories* (2019), MSR '19, pp. 383–387.

[10] REDHAT. Website. `http://www.redhat.com/`.

[11] SCHUCKERT, F., HILDNER, M., KATT, B., AND LANGWEG, H. *Source Code Patterns of Buffer Overflow Vulnerabilities in Firefox*. Gesellschaft für Informatik e.V., 2018.

[12] TATE, S., BOLLINADI, M., AND MOORE, J. Ubuntu vulnerability study project. `https://span.uncg.edu/vulnerabilities`.

[13] THE DEBIAN PROJECT. Website. `https://www.debian.org/`.

[14] ZALEWSKI, M. American Fuzzy Lop. `http://lcamtuf.coredump.cx/afl/`.