Source Code Patterns of Buffer Overflow Vulnerabilities in Firefox

Felix Schuckert [1] [2] felix.schuckert@htwg-konstanz.de
Max Hildner [1] maxhildner@fastmail.com
Basel Katt [2] basel.katt@ntnu.no
Hanno Langweg [1] [2] hanno.langweg@htwg-konstanz.de

Abstract: We investigated 50 randomly selected buffer overflow vulnerabilities in Firefox. The source code of these vulnerabilities and the corresponding patches were manually reviewed and patterns were identified. Our main contribution are taxonomies of errors, sinks and fixes seen from a developer's point of view. The results are compared to the CWE taxonomy with an emphasis on vulnerability details. Additionally, some ideas are presented on how the taxonomy could be used to improve the software security education.

Keywords: Buffer Overflow, Source Code Patterns, Vulnerabilities, Code Analysis

1 Introduction

The Common Weakness Enumeration (CWE) [Co17b] top 25 show buffer overflow vulnerabilities (CWE-120) in third place. Buffer overflows have existed for a long time. To discover the reason why buffer overflows still occur in code, we investigated source code samples from the open source web browser Firefox [Fi17]. Different categories for buffer overflow vulnerabilities already exist in CWE. These categories take a technical point of view; they look at aspects such as which memory locations are involved. For example, there are categories for accessing memory on the stack or on the heap. Such categories do not help software developers to avoid buffer overflow vulnerabilities. Developers have to know how vulnerabilities occur and what kind of source code patterns are common for vulnerabilities. Additionally, developers have to know how vulnerabilities can be mitigated. For example, it is important to check inputs carefully and to not misuse memory-critical functions as *memcpy()* that is listed as on of security development lifecycle banned function calls from Microsoft [Mi17]. To fill the gap in the current categorization approaches and provide a structured body of knowledge for software developers to mitigate buffer overflow

[1] HTWG Konstanz, Department of Computer Science, Alfred-Wachtel-Straße 8, 78462 Konstanz, Germany
[2] Department of Information Security and Communication Technology, Faculty of Information Technology and Electrical Engineering, NTNU, Norwegian University of Science and Technology, Teknologivegen 22, 2815 Gjøvik, Norway

doi:10.18420/sicherheit2018_08

vulnerabilities, we reviewed 50 source code samples of buffer overflow vulnerabilities in Firefox. In our review, we considered which types of errors the developers made, which sinks were involved in buffer overflow vulnerabilities and how developers patched the vulnerability. Categories were created based on the results from the reviews. These results are compared to the categories from CWE to see the difference from a developer's point of view.

This paper begins with an overview of related work in section 2. The following section explains how the source code was obtained as well as the review method. The categories for buffer overflows are presented in sections 4, 5 and 6. The last two sections provide a discussion of the results and suggestions for future work.

2 Previous and related work

SQL injection vulnerabilities in 50 source code samples from open source projects were analysed by [SKL17] using a similar method. The reviewed programming language was PHP. Classifications of source code patterns exist. Classifying source code into bad code, clean code and ambiguous code was done by Lerthathairat; Prompoon [LP11]. Metrics in source code like comments, the size of the function, et cetera. were analysed using fuzzy logic to determine which category the source code belongs to. Bad and ambiguous code are considered for refactoring. More security-related work is by Hui et al. [Hu10], using a software security taxonomy for software security tests. The security defects taxonomy was created based on the top 10 software security errors from authoritative vulnerability enumerations. It is categorized into into *induced causes*, *modification methods* and *reverse use methods*. Hui et al. [Hu10] suggested to use their taxonomy as security test cases.

Massacci; Nguyen [MN10] investigated different data sources for vulnerabilities, e.g. Common Vulnerabilities and Exposures (CVE) [Co17a], National Vulnerability Database (NVD) [Na17], et cetera. They checked which data sources were used by other research projects. In their work, Firefox was used as database for the analysis. Semantic templates were derived from the existing CWE database and are supposed to help understand security vulnerabilities by Wu et al. [WSG11]. The authors did an empirical study to prove that these semantic templates have a positive impact on the time until a vulnerability is completely found.

The work by Bishop et al. [Bi12] [Bi10] presents a taxonomy of how buffer overflow vulnerabilities occur, considering which preconditions are required to exploit a vulnerability. These preconditions are not suited to teach software developers to mitigate vulnerabilities. For example, taking into consideration the category that the program can jump to a memory location in the stack. This is relevant for exploiting the vulnerability, but it will not help to understand what kind of mistakes were done in developing the source code. Kratkiewicz; Lippmann [KL05] used a taxonomy of buffer overflow vulnerabilities to create a data set of 291 small C programs. The data set was analysed with static and dynamic code analysis tools. The tools were then evaluated regarding their detection rate, false positive rate, et ceterea. Ye et al. [Ye16] analysed 100 buffer overflow vulnerabilities and the corresponding patches, using the data to evaluate static code analysis tools. The evaluated tools were *Fortify*, *Checkmarx* and *Splint*. Shahriar; Haddad [SH13] showed how to automatically patch buffer overflow vulnerabilities, including a classification of different types of buffer overflow vulnerabilities. For each of these categories, rules were offered to mitigate the vulnerability. The SEI CERT coding standards [St16] provide an overview on how to implement memory-critical parts in C. The standards are presented as necessary to ensure safety, reliability and security. Non-compliant and compliant code examples help developers to better understand the coding standards.

3 Method

To create the source code pattern categories, selected data sets are needed for the review process. We focus on vulnerabilities which are tracked in the CVE database. We chose source code samples from Firefox because it has many reported buffer overflow vulnerabilities - on average about 30 buffer overflow vulnerabilities per year. Additionally, the Bugzilla [Bu17] platform offers a public discussion about the bug fixes, which helps to identify the relevant source code pattern for the vulnerability. 187 CVE reports are connected to buffer overflow vulnerabilities and Firefox in the time frame from 2010 to 2015 (six years). We choose 2015 as a cut-off to ensure we would have access to a patch as well. We use 50 randomly selected CVE reports which also provide a patch to fix the vulnerability. The patch is determined by a *CONFIRM* flag in the CVE report which indicates the correct Bugzilla report. The bug report contains a link to the patch which fixes the vulnerability. Firefox patches are managed with a version control tool. For each of theses patches, the hash value of the parent version is specified. That version is used as a source code sample containing the buffer overflow vulnerability.

The vulnerable version was reviewed regarding the types of errors made by developers and which sinks were used. A sink is the last instance where unchecked user input can exploit a vulnerability. For example, the function *memcpy()* is a common sink for buffer overflow vulnerabilities. In order to see which errors were made and which sinks were used, a data flow analysis was performed. This was done manually because within the bounds of our project, we could not find a proper tool that was able to analyse such a huge project like Firefox. It is possible that types of errors appear several times because a combinations of errors can also result in a vulnerability. The errors were considered from a developer's point of view. However, the sink is more focused on which critical functions and source code parts are used. This helps developers to recognize critical functions. The patch was used to see how developers fixed the vulnerability. The review of the patch was used to create categories for the fixes.

4 Types of errors

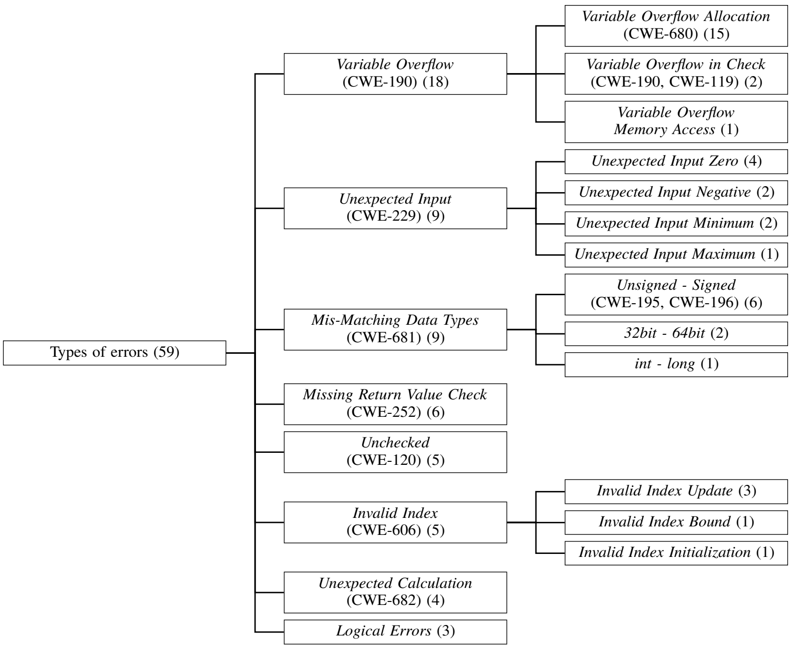
Types of errors (59)
Variable Overflow (CWE-190) (18)
Variable Overflow Allocation (CWE-680) (15)
Variable Overflow in Check (CWE-190, CWE-119) (2)
Variable Overflow Memory Access (1)
Unexpected Input (CWE-229) (9)
Unexpected Input Zero (4)
Unexpected Input Negative (2)
Unexpected Input Minimum (2)
Unexpected Input Maximum (1)
Mis-Matching Data Types (CWE-681) (9)
Unsigned - Signed (CWE-195, CWE-196) (6)
32bit - 64bit (2)
int - long (1)
Missing Return Value Check (CWE-252) (6)
Unchecked (CWE-120) (5)
Invalid Index (CWE-606) (5)
Invalid Index Update (3)
Invalid Index Bound (1)
Invalid Index Initialization (1)
Unexpected Calculation (CWE-682) (4)
Logical Errors (3)

Fig. 1: Taxonomy of errors developers introduced based on the data set.

Figure 1 shows an overview of the categories for the types of errors found in our data set. It has more than 50 assignments because one type of error can lead to other types of errors which then result in a buffer overflow vulnerability. The categories created for the data are the following:

1. ***Variable Overflow***: Many instances of buffer overflows in our review are correlated to integer overflows or underflows. These over-/under-flowed variables are used to check the input size (*Variable Overflow in Check*). Because of the wrong value, these checks pass inputs resulting in a buffer overflow condition. This type of error can be represented in a combination of the following CWE ids: The CWE-190 (Integer Overflow or Wraparound) connected with the keyword *CanProcede* to CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer).

   An overflowed or underflowed variable is used for allocating memory (*Variable Overflow Allocation*). The allocated memory is smaller than the input copied into it. This results in a buffer overflow. The related CWE id is CWE-680, which states that a calculation result is used to allocate memory and an integer overflow causes less memory to be allocated. The allocation of an insufficient amount of memory in our data set occurred in the following sub-patterns:

a) *Allocation too small*: An integer overflow can either have a negative result (signed int) or very small result (unsigned int). These integer overflows occur because user data is included in a calculation. This can be a simple addition to a static value or it can be methods computing a length. As an example, the length of the user input could be the sum of multiple user inputs. If memory is allocated from an integer overflow result, the later usage of the memory will result in a buffer overflow vulnerability.

b) *Existing buffer size check*: Some data sets used already existing buffers and checked if the buffer size had to be increased. An integer overflow in such a check also results in a buffer that is too small.

2. ***Unchecked***: The review shows vulnerabilities where user input reaches methods that are vulnerable for buffer overflows. Source code samples without any checks fall into this category. The corresponding CWE id (CWE-120) explains it as follows: "The program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow." Accordingly, this is the classic buffer overflow where input is not checked and then is able to reach critical functions like *memcpy()*.
3. ***Unexpected Input***: This category covers unexpected user inputs. Usually, all of the error types could fall into this category, but it covers inputs that the developers did not expect to occur. For example, the parameter of a method is the size of a file. Accordingly, the parameter must not be negative (*Unexpected Input Negative*). Another example would be a parameter that has a minimum size, thus, falls into the category *Unexpected Input Minimum*. Nevertheless, the parameter can be outside of the expected range because of some other preconditions or special inputs. For example, a specially crafted file would return a negative result as the content length. If a developer uses such premises for memory-critical parts, a buffer overflow vulnerability could occur. One sample also had expected a maximum input (*Unexpected Input Maximum*) of a value. This vulnerability was related to shaders programs which are programs running on the graphics processor. Developers did not think that the value of the input could be higher than the number of existing shaders. CWE-229 (Improper Handling of Values) is best suited to our *Unexpected Input* category because the inputs are not handled properly. The CWE category covers multiple variants like missing values or undefined values. It does not cover numerical values which are too small, too high or in an unexpected range.
4. ***Mis-Matching Data Types***: This category covers vulnerabilities where values of different data types are assigned to each other which is presented in CWE-681 (Incorrect Conversion between Numeric Types). A common example is the assignment from *unsigned int* to *signed int*. These assignments are also covered by CWE-119 (Signed to Unsigned Conversion Error) and CWE-196 (Unsigned to Signed Conversion Error). This type of error occurred in our data set in combination with *Unexpected Calculation* or just as a simple conversion with the outcome of a buffer overflow vulnerability. Also, some samples contain assignments of different variable lengths,

for example, assignments between 32 bit and 64 bit variables. C/C++ does allow the assignment of variables with different data types. It will interpret the bits according to the new variable type. For example, if a negative value is assigned to an unsigned variable, the first bit will be interpreted as the highest value bit. If such an interpretation is unwanted, subsequent checks and the usage of the variable will be problematic. In our sample, this results in buffer overflow vulnerabilities.

5. ***Missing Return Value Check***: These are vulnerabilities where developers do not check the return value. In our data set it was common that the return value of a memory allocation function was not checked. If the allocation is not possible, the allocation functions returns an error code. If the return value is ignored, the pointer will point to a random memory address. Using this pointer to access memory will likely result in a buffer overflow vulnerability. Usually such a situation only happens when the system or program is out of memory. CWE-252 (Unchecked Return Value) is the related category in the CWE list.
6. ***Invalid Index***: These error types include the usage of an invalid index for a loop. It is split into three subcategories. The first is the *Invalid Index Bound* where the bound is invalid. This can happen because of previous errors like an *Unexpected Calculation*. Samples where such a bound is invalid and the index is used to access memory are counted in this category. Another error is that developers did not update the index correctly (*Invalid Index Update*) which also results in a buffer overflow. One sample had an index initialized to an invalid value (*Invalid Index Initialization*). The best fitting CWE category is CWE-606 (Unchecked Input for Loop Condition) because the *Invalid Index* category is related to loops.
7. ***Unexpected Calculation***: This category covers source code samples where unexpected results are obtained during calculation. All the samples had a negative result. The developers did not expect the result to be negative and the values were used in memory-unsafe functions. Another example is assigning a negative result to an unsigned integer. The unsigned integer will interpret the highest bit which is a 1 as a very large value because it was negative when it was represented in a signed datatype. Such an example is represented in CWE ids with the following: CWE-682 (Incorrect Calculation) connected with the keyword *CanFollow* to CWE-681 (Incorrect Conversion between Numeric Types).
8. ***Logical Errors***: Two vulnerable samples showed developers made logical errors. For example, not enough memory was allocated regardless of the input and the following code did write into unintended memory parts. Another sample had an issue where the length of a variable was not updated correctly and that length was used in memory-critical parts. Three samples showed buffer overflow conditions because they had logical errors.

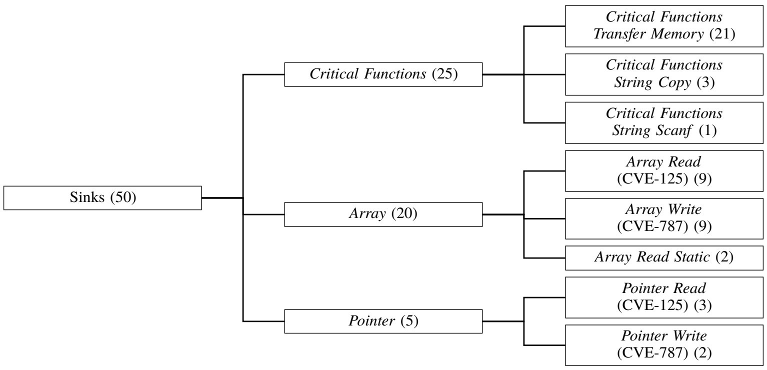
Sinks (50)
Critical Functions (25)
Array (20)
Pointer (5)
Critical Functions Transfer Memory (21)
Critical Functions String Copy (3)
Critical Functions String Scanf (1)
Array Read (CVE-125) (9)
Array Write (CVE-787) (9)
Array Read Static (2)
Pointer Read (CVE-125) (3)
Pointer Write (CVE-787) (2)

Fig. 2: Taxonomy of sinks based on the data set.

5 Sink categories

What kind of sinks were used in the data set are shown in figure 2. These are classified into the following categories:

1. ***Critical Functions***: Sinks of this category are memory-critical functions. Common functions in C/C++ are *memcpy()* or *memset()*. These functions are categorized into the subcategory *transfer memory*. Three sinks of the data set used a string copy function (*strcpy()*) and one sample used the *scanf()* function. These are functions which are also found in the banned functions list for security development lifecycle [Mi17].
2. ***Array***: Arrays in C/C++ are very similar to pointers. The memory for an array is arranged such that all entries are next to each other. If an array field is accessed using an invalid index, a buffer overflow vulnerability exists. All data sets where the sinks are arrays fall into this category. This can be split into write (CWE-787: Out-of-bounds write) and read (CWE-125: Out-of-bounds read). Two samples performed a read access with a static index. Both of them used the index zero, which is typically used to get the first element of an array.
3. ***Pointer***: The last category for sinks is the misuse of pointers. These are sinks where pointers are used to access memory. This category can be mapped to the CWE-468 (Incorrect Pointer Scaling) category. This category can also be split into subcategories of read (CWE-125: Out-of-bounds read) and write (CWE-787: Out-of-bounds write).

6 Fix categories

The results for the different fixes are categorized and seen in figure 3. They are connected to the different problem types. Fixes are categorized as follows:

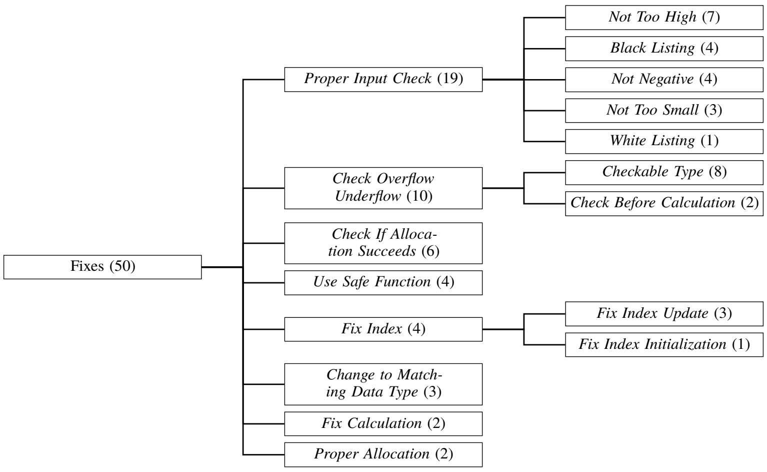

Fig. 3: Taxonomy of fixes for the vulnerabilities based on the data set.

1. ***Proper Input Check***: Fixes for this category are input checks which were completely missing (*Unchecked*). Also fixes which check inputs that developers didn't have in mind fall into this category (*Unexpected Input*). The subcategories are for the different kinds of checks. For example, negative values are a common input developers did not expect. Also some vulnerabilities which have *Variable Overflow in Check* and *Variable Overflow Allocation* as error categories were fixed by checking if the input value was not too high. Also some fixes did just check if a value was not too small. This is commonly a fix when developers thought the input could not be that small. Black listing where specific inputs are filtered out and white listing where only specific inputs are allowed were found as fixes in the data set.
2. ***Check Overflow Underflow***: Firefox has a *checkedint* class which allows to check if an overflow or underflow occurred. Accordingly, fixes used these classes instead of *int* variables and checked for over- and underflow occurrences. Two fixes did check the input, i.e., if an integer overflow occurred in the calculation before using it. This fixes problems from the *Variable Overflow in Check* and *Variable Overflow Allocation* categories.
3. ***Check If Allocation Succeeds***: Some vulnerabilities fall in the error category *Missing Return Value Check*. In our data set, these missing return value checks are related to allocating memory. As the name already hints, fixes in this category check these return values and change the control path accordingly.
4. ***Use Safe Function***: Memory related functions provide a secure function which requires an additional parameter. This parameter is used to restrict the size which is used in the memory-critical function. A common example is *strcpy* and *strcpy_s*. The additional parameter is used to provide the size of the string. This prevents a vulnerability where the source string has no null character or the size of the source

string is larger than the size of the destination string. Four fixes used such functions to remove the vulnerability.

5. ***Fix Index***: This category is related to the error type *Invalid Index*. These errors were fixed by using valid indexes. One instance was fixed by changing the data type so that the index was not invalid any more. The fixes are split into the same subcategories as the error type. For example, one index update fix was implemented by inserting a *break* statement.
6. ***Change to Matching Data Type***: Three vulnerabilities were patched by changing the data type. This fix is related to the *Mis-Matching Data Types* error type category. Three of these kind of errors were patched by changing the data type. The remaining samples were fixed by correcting a previous error which then only resulted in a vulnerability because of mis-matching data types. For example, an integer overflow was fixed, which resulted in a negative result that was assigned to an unsigned integer variable. As long as the value was positive, this did not create a problem.
7. ***Fix Calculation*** Two samples were patched by fixing the calculation. The calculation was adjusted accordingly such that the undesired results will no longer occur.
8. ***Proper Allocation*** The last category of fixes are patches where the allocations were fixed. For example, the allocation did not reserve enough memory. If the allocation was changed such that it allocated the right amount of memory, it falls into this category. Two samples patched the vulnerability by correctly allocating memory.

7 Discussion

Firefox is a well-known open source product and the source code is reviewed a lot. Accordingly, the vulnerabilities from Firefox usually had input checks before potentially dangerous functions or memory accesses were used. The vulnerabilities most often existed because an integer overflow or underflow occurred. It is important to teach developers the right use of variables which may overflow/underflow. Also the assignment of variables with different data types in *C/C*++ is problematic and should be avoided. Nevertheless, if these assignments are required, they should be used carefully.

The error types were related to existing CWE categories. CWE-888 contains software fault pattern clusters. The containing category CWE-890 (SFP Primary Cluster: Memory Access) is related to buffer overflow vulnerabilities. This category also has the following subcategories:

- CWE-970 SFP 2. Cluster: Faulty Buffer Access: **covered**
- CWE-971 SFP 2. Cluster: Faulty Pointer Use: **did not occur in data set**
- CWE-972 SFP 2. Cluster: Faulty String Expansion: **did not occur in data set**
- CWE-973 SFP 2. Cluster: Improper NULL Termination: **did not occur in data set**
- CWE-974 SFP 2. Cluster: Incorrect Buffer Length Computation: **covered**

CWE-970 and CWE-974 are covered by our data set. Surprisingly, CWE-971 did not occur in our data set. This is due to the fact that this category has only very specific CWE subcategories, for example, when using a *null* pointer or using pointers to determine a length. Also CWE-972 and CWE-973 did not occur in our data set. There was no vulnerability sample related to an improper *null* termination of a *String* variable. Another related cluster is CWE-969 (SFP Secondary Cluster: Faulty Memory Release) which covers vulnerabilities where memory is released and still used later on. This includes vulnerabilities like "double free" or releasing memory which is not on the heap. Unfortunately, our data set did not cover vulnerabilities which fit into this cluster.

As already stated, Microsoft released a list of banned functions for the security development lifecycle [Mi17]. Most of our sinks that fall into the category *Critical Functions* are found on the list. Our data set contained the critical functions *memmove()* and *memset()*, which are not found in the banned list because these functions are using a restricting length parameter. Only four of samples using a banned function were fixed by using a safe function. 17 of the sinks in our data set did use the function *memcpy()*. According to the list, the function *memcpy_s()* should be used which requires an additional parameter defining the size of destination. None of the patches used the function to fix the vulnerability. It is easy to tell developers to avoid buffer overflow vulnerabilities, but there is a huge list of critical functions. Developers have to know which functions are critical. Static code analysis tools might be useful to find these functions. Nevertheless, in our data set only half of the vulnerabilities use critical functions. Additionally, there are many different permutations of buffer overflow vulnerabilities which makes the mitigation for developers problematic.

Our results show that buffer overflow vulnerabilities are not simply avoided by having a list of critical functions. Buffer overflow vulnerabilities occur in many different permutations and in combination of errors. Accordingly they are not easy to prevent by just learning simple vulnerabilities. Our results provide an overview of source code patterns which are found in our data set. These can be used to teach developers that all kinds of permutations of our categories can result in a buffer overflow vulnerabilities.

8 Conclusions and future work

To minimize the occurrence of buffer overflow vulnerabilities, different source code patterns have to be detected and avoided. To gain a better understanding of how such patterns look like, we analysed 50 buffer overflow CVE reports related to Firefox. We created categories for the types of errors the developers made, what kind of sinks were used and how the developers fixed the vulnerability. These categories were compared to existing CWE categories. Some categories are not found as a direct CWE category. Likewise, our data set does not include all CWE categories. The focus of the categories is seen from a developer's point of view instead of a technical representation of the vulnerability details. This helps to use the categories to teach developers which source code patterns and errors are common for buffer overflow vulnerabilities.

Our patterns could be used to create different learning exercises using different permutations. An interesting point will be to create these exercises automatically. The LAVA tool [Do16] injects buffer overflow vulnerabilities in C code. It would be interesting to integrate our patterns into this tool. This will be an important step because malicious developers might already have developed such tools. It would reveal some limitations and maybe risks which might occur by automatically creating vulnerabilities in the future. Our earlier work [Sc16] is a tool which injects SQL injection vulnerabilities in Java source code using an abstract syntax tree. A similar approach might be possible to inject buffer overflow vulnerabilities in C/C++ code. Another avenue of research would be using these categories to benchmark static code analysis tools. Data sets could be created using different permutations of our categories. It will be interesting to see if all permutations are detected by static code analysis tools as well as the false positives and the false negatives rates of the tools.

References

[Bi10] Bishop, M.; Howard, D.; Engle, S.; Whalen, S.: A taxonomy of buffer overflow preconditions. In. 2010.

[Bi12] Bishop, M.; Engle, S.; Howard, D.; Whalen, S.: A taxonomy of buffer overflow characteristics. In. Vol. 9, pp. 305–317, 2012.

[Bu17] Bugzilla, 2017, URL: `https://bugzilla.mozilla.org`.

[Co17a] Common Vulnerabilities and Exposures, 2017, URL: `https://cve.mitre.org/`.

[Co17b] Common Weakness Enumeration, 2017, URL: `https://cwe.mitre.org/`.

[Do16] Dolan-Gavitt, B.; Hulin, P.; Kirda, E.; Leek, T.; Mambretti, A.; Robertson, W.; Ulrich, F.; Whelan, R.: Lava: Large-scale automated vulnerability addition. In: Security and Privacy (SP), 2016 IEEE Symposium on. IEEE, pp. 110–121, 2016.

[Fi17] Firefox, 2017, URL: `https://www.mozilla.org/de/firefox/`.

[Hu10] Hui, Z.; Huang, S.; Hu, B.; Ren, Z.: A taxonomy of software security defects for SST. In. Pp. 99–103, 2010.

[KL05] Kratkiewicz, K.; Lippmann, R.: A taxonomy of buffer overflows for evaluating static and dynamic software testing tools. In: Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics. Pp. 500–265, 2005.

[LP11] Lerthathairat, P.; Prompoon, N.: An approach for source code classification to enhance maintainability. In. Pp. 319–324, 2011.

[Mi17] Microsoft: Security Development Lifecycle (SDL) Banned Function Calls, 2017, URL: `https://msdn.microsoft.com/en-us/library/bb288454.aspx`.

[MN10] Massacci, F.; Nguyen, V. H.: Which is the right source for vulnerability studies?: An empirical analysis on Mozilla Firefox. In. 4:1–4:8, 2010, ISBN: 978-1-4503-0340-8.

[Na17] National Vulnerability Database, 2017, URL: https://nvd.nist.gov/.

[Sc16] Schuckert, F.: PT: Generating Security Vulnerabilities in Source Code. In: Sicherheit 2016 - Sicherheit, Schutz und Zuverlässigkeit. Pp. 177–182, 2016.

[SH13] Shahriar, H.; Haddad, H. M.: Rule-based source level patching of buffer overflow vulnerabilities. In. Pp. 627–632, 2013.

[SKL17] Schuckert, F.; Katt, B.; Langweg, H.: Source Code Patterns of SQL Injection Vulnerabilities. In: Proceedings of the 12th International Conference on Availability, Reliability and Security. ACM, 72:1–72:7, 2017.

[St16] Standard, C. C.: SEI CERT. In. 2016.

[WSG11] Wu, Y.; Siy, H.; Gandhi, R.: Empirical results on the study of software vulnerabilities. In. Pp. 964–967, 2011.

[Ye16] Ye, T.; Zhang, L.; Wang, L.; Li, X.: An Empirical Study on Detecting and Fixing Buffer Overflow Bugs. In. Pp. 91–101, 2016.