Buffer Overflow Detection via Static Analysis: Expectations vs. Reality

I.A. Dudina <eupharina@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. Over the last few decades buffer overflow remains one of the main sources of program errors and vulnerabilities. Among other solutions several static analysis techniques were developed to mitigate such program defects. We analyzed different approaches and tools that address this issue to discern common practices and types of detected errors. Also, we explored some popular sets of synthetic tests (Juliet Test Suite, Toyota ITC benchmark) and set of buggy code snippets extracted from real applications to define types of defects that a static analyzer is expected to uncover. Both sources are essential to understand the design goals of a production quality static analyzer. Test suites expose a set of features to support that is easy to understand, classify, and check. On the other hand, they don't provide a real picture of a production code. Inspecting vulnerabilities is useful but provides an exploitation-biased sample. Besides, it does not include defects eliminated during the development process (probably with the help of some static analyzer). Our research has shown that interprocedural analysis, path-sensitivity and loop handling are essential. An analysis can really benefit from tracking affine relations between variables and modeling C-style strings as a very important case of buffers. Our goal is to use this knowledge to enhance our own buffer overrun detector. Now it can perform interprocedural context- and path-sensitive analysis to detect buffer overflow mainly for static and stack objects with approximately 65% true positive ratio. We think that promising directions are improving string manipulations handling and combining taint analysis with our approaches.

Keywords: software error detection; static analysis; buffer overrun

DOI: 10.15514/ISPRAS-2018-30(3)-2

For citation: Dudina I.A. Buffer Overflow Detection via Static Analysis: Expectations vs. Reality. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 3, 2018, pp. 21-30. DOI: 10.15514/ISPRAS-2018-30(3)-2

1. Introduction

Buffer overflow is a type of program defect caused by buffer access with index that exceeds buffer's bounds. This can lead to a program crash or even to a security vulnerability. Defects of such kind are still common, despite all efforts made to

eliminate them. There are several techniques one can apply to detect buffer overflows. One approach is to employ testing and dynamic analysis. These methods don't suffer from false positives, but in most cases, it's impossible to check all execution paths, so some defects can remain undetected. Another approach is to analyze program code without executing it. In this way, one can find a defect on any path, even rarely executed. In this paper, we will focus on the latter approach known as static analysis.

We are interested in building a buffer overflow detector that is applicable to large C/C++ programs with millions of lines of code while producing decent analysis performance and quality. Basic properties of the algorithms constituting such a detector are well-known and include among others interprocedural analysis, path sensitivity, and loop handling. However, after initial support for these features has been made and the quality goals achieved, it is unclear which direction to choose for the further improvement. The usual development pace that comes from the customer feedback and own code analysis may be not enough. In the following chapters, we'll overview possible sources of inspiration for the buffer overflow detector development, present our short survey that is based on the buffer overflow-related vulnerabilities sample from the CVE database, then briefly describe our experience of developing an overrun detector as a part of the Svace tool, and present our conclusions from tools and vulnerabilities analysis.

2. Buffer overflow detection techniques and tools

There exist many static analysis tools that can detect buffer overflows. In this section, we conduct a brief survey on the most popular methods.

Some buffer overflows can be detected during the process of lexical analysis, like in the ITS4 tool [1]. Most common errors and bad patterns can be found at this level. This technique can work really fast and, as it doesn't involve compilation, can be easily applied to any code, even if it is not complete. As a result, such analysis can be performed "on-the-fly" during the process of code development with IDE, so that erroneous patterns are eliminated on the very early coding stage. Of course, such a lightweight method is far from being sound, i.e. it misses many defects. Even changing the name of a variable can prevent such tools from detecting a defect.

To detect more defects a deeper analysis of code is needed. To achieve this, many tools use the idea of abstract interpretation [2]. Some tools chose different numerical abstract domains to implement the analysis of integer index values, buffer sizes, and string lengths. These domains include intervals, zones, octagons, affine equalities, interval linear equalities, convex polyhedra, tropical polyhedra, etc. [3]. Tools based on these approaches derive sound relationship between integer values listed above in varying degrees of precision. Soundness is a major advantage of such tools, but less precise domains produce large number of false positives, while analysis with more precise domains doesn't scale on many real-world programs.

Another popular approach is symbolic execution. The main idea of this method is performing analysis by traversing all paths in a function separately. This approach can be used to build a path-sensitive detector i.e. that can find errors that, at the same time, occur only on a certain feasible function path and are not inevitable for any single point from this path alone. While processing a particular path, the analyzer keeps track of variables values and relationships and computes a path predicate, i.e. a conjunction of all corresponding branch conditions that are taken along this path. This information is used to prune infeasible paths and check buffer access instructions. Analyzing all paths in a function can be a challenging task due to the path explosion, so a number of techniques are proposed to reduce this problem. A simple, but often effective approach is to abandon the idea of full path coverage and just to stop the analysis after some threshold or time limit reached. Another approach is to merge symbolic states at join points, preserving path-sensitivity of analysis by providing guard conditions for joined states. Third approach, first introduced in Marple, is employing demand-driven analysis [4], [5], i.e. reducing the set of analyzed paths by focusing only on those that end with buffers access.

One of the main obstacles for all mentioned symbolic execution-based approaches is handling loops. Typical solution is to implement some heuristics to handle the most simple and common loops and ignore other loops. However, there are methods proposed to handle loops with multiple paths inside and summarize their effect on program values [6].

Many buffer overflow errors are caused by violations of function contracts. This can happen when a caller of a library or a user function provides unexpected data to a function, or, on the contrary, a function is not able to correctly handle all input cases implied by the contract. Interprocedural analysis is needed to detect such inconsistencies.

On the lexical analysis level, formal and actual arguments matching can be based on similar variables names and usually happens only for the well-known library callees like `memcpy`. For more rigorous scan some tools analyze the whole program as a unified inter-procedural graph. The monomorphic analysis merges information for every call-site — efficient, but imprecise approach. The polymorphic analysis treats each call site individually, so this approach provides context-sensitivity but scales poorly.

An alternative approach is using some approximation of a function's behavior when analyzing its caller. These approximations can be provided in user's annotations, but they are not always available. A tool can use its own findings obtained by the callee analysis as an approximation. This approached is called summary-based. By choosing the right function order, a tool can minimize the number of missing summaries, but handling recursion still requires additional tricks, e.g. making several analysis passes over strongly connected components of the call graph.

3. Buffer overflow detection tools benchmarking

For the past twenty years several studies have been published on evaluating and testing buffer overflow detectors. In addition, there exist different test suites, which provide sets of synthetic buggy and correct code snippets to test the abilities and false positive rate of static analysis tools.

One of the biggest and probably the most popular benchmark is Juliet Test Suite C/C++, created by NSA's Center for Assured Software (CAS) [7]. For C/C++ code it contains 64,099 test cases tagged by CWE entries. Groups corresponding to buffer overflow defects are CWE 121 — "Stackbased Buffer Overflow" (4,968 tests), CWE 122 — "Heapbased Buffer Overflow" (5,922 tests), CWE 124 — "Buffer Underwrite" (2,048 tests), CWE 126 — "Buffer Over-read" (1,452 tests), and CWE 127 — "Buffer Under-read" (2048 tests). Tests in this suite are also tagged with a number called "flow variant" that represents the complexity of control and data flow in a particular test case.

Control flow variants cover different types of conditionals (e.g. `STATIC_CONST_FIVE==5`, `globalReturnsTrueOrFalse()`, etc.) and different control statements (`switch`, `while`, etc.). Data flow variants describe many types of intraprocedural data flow and interprocedural interaction, e.g. data passing through function arguments (via pointer, C++ reference, array, container, etc.), return value, global variable, etc. There are many flow variants that represent C++-specific features and not applicable to C-tests.

We noticed that the distribution of the flow variants is close to uniform in groups of our interest. Another observation is large number of tests involving wide characters. Many tests contain library function usage, e.g. `memcpy`-like functions, string manipulations, format string processing, etc.

Toyota ITC Benchmark is a test suite created by Toyota InfoTechnology Center aimed at the static analysis tool evaluation [8]. It contains 1,276 simple tests (638 erroneous and 638 correct) divided into 9 types and 51 sub-types. Our interest is in the following tests: sub-types "static buffer overrun" (54 cases), "static buffer underrun" (13 cases) from the "static memory" type and sub-types "dynamic buffer overflow" (32 cases), "dynamic buffer underrun" (39 cases) from the "dynamic memory" type. Each case is represented by a pair of a buggy test and a fixed test.

These samples cover following features in varying combinations: (i) static, stack and heap buffers; (ii) different element types (`char`, `int`, `float`, `struct`, etc.); (iii) index calculations (constant, linear and non-linear expressions, passed as an argument or returned from a function, loop variables and array elements); (iv) obtaining buffer address (local/global variable, function argument, pointer arithmetic including loop variables and aliases); (v) buffer size (heap buffers only with constant sizes, pointer casting); (vi) access types (via index, pointer dereference, in a library function, in a string function).

4. Survey on overflow-related CVEs

We believe that although evaluating with a test suite could give a good insight in a particular tool's abilities, any test suite alone cannot perfectly represent the whole populations of buffer overflow defects in real code. One (but not the only one) noble goal for static analyzers is to prevent security vulnerabilities to sneak in the project source code. We wanted a better understanding of the features of a static analyzer that are more or less important for achieving this goal. Our survey technique was inspired by [9] and we mostly followed in their footsteps to produce a set of vulnerabilities to classify.

We have to note that detection of exploitable vulnerabilities is not the only goal of a static analyzer. Still there are some types of defects that don't lead to vulnerabilities or may not be exploited with ease, but it is undesirable to have those in the source code. Besides, we believe that nowadays developers more intensively use different (static and/or dynamic) analysis tools before releasing the product. For this reason, many simple defects are eliminated during the development process and don't appear in the vulnerability databases. Consequently, we think that analysis of the vulnerabilities can reveal the weakest sides of modern static analysis and show potential improvement directions.

First of all, we have randomly picked 100 entries from the "overflow" category from the CVE database [10]. For 25 of them we could find a source code of the vulnerable version to inspect. For each defect, we have studied its causes in the code and then classified the defect by several attributes. Our set of attributes is based on the taxonomy provided in [11] with some changes.

Our first insight is that there are some trends in our sample that can be explained by the source of this sample (vulnerability database): (i) most of the overflows from our sample (72%) happened on write memory access, only few on read access; (ii) only the upper bounds of buffers are exceeded in the defects from our sample; (iii) almost all defects (92%) occurred when tainted data (unbounded data from network, file read, input parameters etc.) overflowed some buffer.

We also noticed that simple errors (e.g. using unsafe functions like `strcpy`) are present in the old code (before 2010), but rarely in the late entries. We believe that this can be partially explained by the usage of code analysis tools.

In our sample about a half of overflowed buffers (48%) reside on a stack, other half (48%) is allocated on a heap, and just a few are global variables.

40% of all defects have overflowed buffer accessed via index (e.g. `buf[i]`), 12% via pointer dereference, 44% via library calls, 24% of which are string functions. The latter requires C-strings modeling to properly analyze such patterns. When buffer is accessed in a library call, we think of size/limit argument as an index (when it's reasonable) for further investigation.

According to our data, 48% of all vulnerable buffers have constant size (all stack and static buffers and a few buffers on the heap). Another 16% have a size that is calculated as a linear combination of other variables. As a result, almost half of all inspected defects require deep analysis of integer variables relationship to detect them.

Another feature that we have evaluated for every entry is whether buffer allocation is global or resides in the same function with buffer access. We have found that this is true only for 24% of defects. On the other hand, all index calculations are in the same function with the access in 32% of defects. Both properties are true for 12% of defects.

It follows from the foregoing that interprocedural analysis is essential for buffer overflow detection.

Last thing that we have checked is whether there exists a program point that any path through this point will lead to a corresponding error. If there is no such point, then we assume that path-sensitive analysis is needed to detect this defect. Our sample contains only 28% of defects, for which such a program point exists. This means that path-sensitivity will provide the real advantage for a static analysis tool.

5. Svace buffer overrun detector

Svace is a static analysis tool that is designed to find as many defects of different types as possible with few false positives and acceptable analysis time [12]. The purpose of this work is to improve the Svace buffer overflow detector with the most needed features. Our detector implements the interprocedural path-sensitive detection algorithm based on symbolic execution with state merging [13]. For now, the analysis scope is limited to detection overflows of buffers with compile-time-known size. Our detector looks for faulty paths in a function, i.e. it reports a warning if it finds a path that for any input values is either infeasible or produces an error. Such a strict defect definition is chosen to prevent many false positives caused by unknown function preconditions.

For a buffer access instruction, we collect a predicate that implies that there exists a faulty path through this instruction. We use an SMT solver to search a solution for this predicate if any. In case of this formula is satisfiable, we use its model provided by the solver to extract a faulty path. It follows from our experience that simply asking solver for any index value that exceeds buffer bounds in our case leads to many false positives. Reasons for that are unknown function precondition and symbolic path conditions being not precise enough (due to poor loop handling, calls of unknown or complex functions, etc.).

Our interprocedural analysis is implemented using summaries. In the function summary, we save the information about relationships between integer values on function entry and exit points. We also save overflow conditions for those input-dependent buffer accesses whose correctness can only be checked in the caller context. Such facts can be propagated to the caller more than once, so the analysis can find an overflow of a buffer allocated in a function that is far away on the call stack from a function with the access instruction. We also implemented a heuristic to handle simple loops that have an inductive variable iterating over an arithmetic progression. Currently on Android 5.0.2 our detector emits 351 warnings with 65% true-positive ratio.

6. Conclusion

We have inspected a number of buffer overflow test suites, related CVE entries, and the source code of large production projects that our tool regularly analyzes. All three sources are essential to understand the design goals of a production quality static analyzer. Test suites expose a set of features to support that is easy to understand, classify, and check. On the other hand, they don't provide a real picture of a production code. Inspecting vulnerabilities is useful but provides an exploitation-biased sample. Besides, it does not include defects eliminated during the development process (probably with the help of some static analyzer). Finally, while developing a static analyzer one always deals with false positives produced by the tool and reported by customers, but getting false negative samples is much more difficult. True positives reported by the other tools could be useful, but most of the state-of-the-art tools are proprietary and their results are closed.

From what has been said above it follows that interprocedural analysis, path-sensitivity and loop handling are essential. An analysis can really benefit from tracking affine relations between variables and modeling C-style strings as a very important case of buffers.

Our current goal is to improve the Svace buffer overflow detector to reduce the number of false negatives while preserving the moderate level of false positives. For the aforementioned reasons, we think that the most promising directions are handling buffers with dynamic size, C-string modeling, and tracking tainted values. We are working now on the extension of our detection technique described in Section 5 by tracking string length changes happening during string operations in much the same way as we track buffer indexes while calculating integer values. We believe that this will be sufficient for most of cases, but there are some promising works in the area of string solvers [14] that would additionally allow to track also string contents.

As we have seen, static analysis detection of buffer overflows requires a number of techniques from vastly various fields to move on the road from expectations to real code, and there will always be a way to go.

References

[1]. J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In Proceedings of the 16th Annual Computer Security Applications Conference, 2000, pp. 257-269.

[2]. P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1977, pp. 238–252.

[3]. X. Allamigeon. Static analysis of memory manipulations by abstract interpretation – Algorithmics of tropical polyhedra, and application to abstract interpretation. PhD thesis, Ecole Polytechnique X, Nov. 2009. [Online]. Available: https://pastel.archives-ouvertes.fr/pastel-00005850, accessed: 2018-04-08.

[4]. W. Le and M. L. Soffa. Marple: A Demand-Driven Path-Sensitive Buffer Overflow Detector. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, 2008, p. 272-282.

[5]. L. Li, C. Cifuentes, and N. Keynes. Practical and effective symbolic analysis for buffer overflow detection. In Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, pp. 317– 326.

[6]. X. Xie, Y. Liu, W. Le, X. Li, and H. Chen. S-looper: automatic summarization for multipath string loops. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, 2015, pp. 188–198.

[7]. Juliet Test Suite v1.2 for C/C++. User Guide. Available: https://samate.nist.gov/SRD/around.php#juliet_documents, accessed: 2018-04-08.

[8]. S. Shiraishi, V. Mohan, and H. Marimuthu. Test suites for benchmarks of static analysis tools. In Proceedings of the 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Nov 2015, pp. 12–15.

[9]. T. Ye, L. Zhang, L. Wang, and X. Li. An Empirical Study on Detecting and Fixing Buffer Overflow Bugs. In Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation, 2016, pp. 91–101.

[10]. CVE security vulnerability database. Security vulnerabilities, exploits, references and more. Available: https://www.cvedetails.com/index.php, accessed: 2018-04-08.

[11]. K. Kratkiewicz and R. Lippmann. 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, vol. 500, 2006, pp. 44-51.

[12]. A. Borodin and A. Belevantcev. A static analysis tool Svace as a collection of analyzers with various complexity levels. Trudy ISP RAN /Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 111–134.

[13]. I.A. Dudina and A.A. Belevantsev. Using static symbolic execution to detect buffer overflows. Programming and Computer Software, vol. 43, no. 5, 2017, pp. 277–288. DOI: 10.1134/S0361768817050024.

[14]. Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 114–124.

Статический анализ для поиска переполнения буфера: актуальные направления развития

И.А. Дудина <eupharina@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
Московский государственный университет им. М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. В последние десятилетия переполнение буфера остаётся одним из главных источников программных ошибок и эксплуатируемых уязвимостей. Среди прочих подходов к устранению подобных дефектов активное развитие получили различные методы статического анализа. В работе рассматриваются основные подходы и инструменты, используемые для решения этой задачи, с целью выявить наиболее популярные методы и типы обнаруживаемых ошибок. Также исследованы наборы синтетических тестов (Juliet Test Suite, Toyota ITC benchmark) и выборка фрагментов кода реальных приложений, содержащих эксплуатируемую ошибку переполнения буфера. Для понимания направлений развития промышленного статического анализатора важно рассматривать оба эти источника примеров ошибочных программ. Наборы тестов очерчивают круг ситуаций, которые необходимо поддержать в анализаторе, при этом их легко понять, классифицировать и проверить. С другой стороны, они не отражают распределение таких ситуаций в реальном коде. Выборка уязвимостей из промышленных проектов также представляет интерес для исследования, но оказывается смещённой в сторону эксплуатируемых ошибок и к тому же не включает ошибки, исправленные на стадии разработки (возможно, как раз с использованием статического анализатора). Полученные данные были использованы для выделения основных шаблонов дефектов, которые должен обнаруживать статической анализатор с точки зрения пользователя. В результате исследования к наиболее важным возможностям статического анализатора были отнесены межпроцедурный путе- и контекстно-чувствительный анализ, а также базовая поддержка циклов. Кроме того, полезными оказываются отслеживание аффинных отношений между переменными и моделирование строк как важного случая использования массивов. Результаты данного исследования используются для улучшения детектора переполнения буфера, реализованного в рамках инфраструктуры статического анализатора Svace. На данный момент используется межпроцедурный чувствительный к путям и контексту анализ, позволяющий обнаруживать переполнения буфера на стеке и в статической памяти с долей истинных срабатываний 65%. По результатам исследования наиболее перспективными направлениями представляются поддержка строковых операций и внедрение анализа помеченных данных в имеющиеся подходы.

Ключевые слова: анализ программ; статический анализ; переполнение буфера

DOI: 10.15514/ISPRAS-2018-30(3)-2

Для цитирования: Дудина И.А. Статический анализ для поиска переполнения буфера: актуальные направления развития. Труды ИСП РАН, том 30, вып. 3, 2018 г., стр. 21-30 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(3)-2

Список литературы

[1]. J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: A static vulnerability scanner for c and c++ code. In Proceedings of the 16th Annual Computer Security Applications Conference, 2000, pp. 257-269.

[2]. P. Cousot and R. Cousot. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1977, pp. 238–252.

[3]. X. Allamigeon. Static analysis of memory manipulations by abstract interpretation – Algorithmics of tropical polyhedra, and application to abstract interpretation. PhD thesis, Ecole Polytechnique X, Nov. 2009. [Online]. Available: https://pastel.archives-ouvertes.fr/pastel-00005850, accessed: 2018-04-08.

[4]. W. Le and M. L. Soffa. Marple: A Demand-Driven Path-Sensitive Buffer Overflow Detector. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, 2008, p. 272-282.

[5]. L. Li, C. Cifuentes, and N. Keynes. Practical and effective symbolic analysis for buffer overflow detection. In Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, pp. 317– 326.

[6]. X. Xie, Y. Liu, W. Le, X. Li, and H. Chen. S-looper: automatic summarization for multipath string loops. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, 2015, pp. 188–198.

[7]. Juliet Test Suite v1.2 for C/C++. User Guide. Режим доступа: https://samate.nist.gov/SRD/around.php#juliet_documents, дата обращения: 2018-04-08.

[8]. S. Shiraishi, V. Mohan, and H. Marimuthu. Test suites for benchmarks of static analysis tools. In Proceedings of the 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), Nov 2015, pp. 12–15.

[9]. T. Ye, L. Zhang, L. Wang, and X. Li. An Empirical Study on Detecting and Fixing Buffer Overflow Bugs. In Proceedings of the 2016 IEEE International Conference on Software Testing, Verification and Validation, 2016, pp. 91–101.

[10]. CVE security vulnerability database. Security vulnerabilities, exploits, references and more. Режим доступа: https://www.cvedetails.com/index.php, дата обращения: 2018-04-08.

[11]. K. Kratkiewicz and R. Lippmann. 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, vol. 500, 2006, pp. 44-51.

[12]. A. Borodin and A. Belevantcev. A static analysis tool Svace as a collection of analyzers with various complexity levels. Trudy ISP RAN /Proc. ISP RAS, vol. 27, issue 6, 2015, pp. 111–134. DOI: 10.15514/ISPRAS-2015-27(6)-8.

[13]. I.A. Dudina and A.A. Belevantsev. Using static symbolic execution to detect buffer overflows. Programming and Computer Software, vol. 43, no. 5, 2017, pp. 277–288. DOI: 10.1134/S0361768817050024.

[14]. Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, 2013, pp. 114–124.