Dowser: A Guided Fuzzer for Finding Buffer Overflow Vulnerabilities

ISTVAN HALLER, ASIA SLOWINSKA, MATTHIAS NEUGSCHWANDTNER, AND HERBERT BOS

Istvan Haller is a PhD student in the Systems and Network Security group at the Vrije Universiteit Amsterdam. His current research focuses on automatic analysis of software systems and its application to enhance system security. i.haller@vu.nl

Asia Slowinska is an assistant professor in the Systems and Network Security group at the Vrije Universiteit Amsterdam. Her current research focuses on developing techniques to automatically analyze and reverse engineer complex software that is available only in binary form. asia@few.vu.nl

Matthias Neugschwandtner is a PhD student at the Secure Systems Lab at the Vienna University of Technology. mneug@iseclab.org

Herbert Bos is a full professor in Systems and Network Security at Vrije Universiteit Amsterdam. He obtained his PhD from Cambridge University Computer Laboratory (UK). He is proud of all his (former) students, three of whom have won the Roger Needham PhD Award for best PhD thesis in systems in Europe. In 2010, Herbert was awarded an ERC Starting Grant for a project on reverse engineering that is currently keeping him busy. herbertb@few.vu.nl

Buffer overflows have long plagued existing software systems, making them vulnerable to attackers. Our tool, Dowser, aims to tackle this issue using efficient and scalable software testing. Dowser builds on a new software testing paradigm, which we call dowsing, that focuses the testing effort around relevant application components. This paradigm proved successful in practice, as Dowser found real bugs in complex applications such as the nginx Web server and the ffmpeg multimedia framework.

Buffer overflows represent a long-standing problem in computer science, first identified in a US Air Force study in 1972 [2] and famously used in the Morris worm in 1988. Even after four decades, buffer overflows are perennially in the top three of the most dangerous software errors. Recent studies [8] suggest that the situation will not change soon. One way to handle them is to harden the binary using stack canaries, address space randomization, and the like in the hope that the program will crash when the buffer overflow bug is triggered; however, although crashing is better than being pwned, crashes are undesirable, too.

Thus, vendors prefer to squash bugs beforehand and typically try to find as many as they can by means of fuzz testing. Fuzzers feed programs invalid, unexpected, or random data to see whether they crash or exhibit unexpected behavior. Recent research in testing has led to the development of whitebox fuzzing [3, 4, 5]. By means of symbolic execution, whitebox fuzzing exercises all possible execution paths through the program and thus uncovers all possible bugs, although it may take years to do.

Imagine that you are a software tester and you are given a binary, without knowledge about the application internals or its specification. Where do you start? What features will you be looking for? Intuitively, you start from some random input that you refine based on the observed output. Seeing that you will spend most of your time figuring out the input semantics, instead of testing the underlying functionality itself, is not difficult. These are the same challenges that symbolic execution faces when testing applications without developer input.

In this article, we introduce an alternative testing approach that we call dowsing. Rather than testing all possible execution paths, this technique actively searches for a given family of bugs. The key insight is that careful analysis of a program lets us pinpoint the right places to probe and the appropriate inputs to do so. The main contribution is that our fuzzer directly homes in on the bug candidates and uses a novel "spot-check" approach in symbolic execution. Specifically, Dowser applies this approach to buffer overflow bugs, where we achieve significant speed-ups for bugs that would be hard to find with most existing symbolic execution engines.

In summary, Dowser is a new fuzzer targeted at vendors who want to test their code for buffer overflows and underflows. We implemented the analyses of Dowser as LLVM [7] passes, whereas the symbolic execution step employs S2E [4]. Finally, Dowser is a practical solution. Rather than aiming for all possible security bugs, it specifically targets the class of buffer overflows (one of the most, if not *the* most, important class of attack vectors for code injection). So far, Dowser has found several real bugs in complex programs such as nginx, ffmpeg,

and inspircd. Most of them are extremely difficult to find with existing symbolic execution tools.

Dowsing for Candidate Instructions

Dowser builds on the concept of vulnerability candidates, that is, program locations that are relevant to a specific bug type, in our case buffer overflows. In other words, it scans the binary for features that are possible indications for those hard-to-find buffer overflows. For instance, for a buffer overflow to occur, we need code that accesses buffers in a loop. Additionally, we build on the intuition that code with convoluted pointer arithmetic and/or complex control flow is more prone to such memory errors than straightforward array accesses. Moreover, by focusing on such code, Dowser prioritizes bugs that are complicated—typically, the kind of vulnerabilities that static analysis or random fuzzing cannot find. The aim is to reduce the time wasted on shallow bugs that could also have been found using existing methods. In this section, we explain how we identify and rank the vulnerability candidates.

Identifying the Interesting Spots

Previous research has shown that complex code really is more error prone than simple code for bugs in general; however, Zimmermann et al. [9] also argued that we need metrics that exploit the unique characteristics of specific vulnerabilities, e.g., buffer overflows or integer overruns. So how do we design a good metric for buffer overflows?

Intuitively, convoluted pointer computations and control flows are hard to follow by a programmer, and thus more bug prone. Therefore, we select vulnerability candidates, by focusing on "complex" array accesses inside loops. Further, we limit the analysis to pointers that evolve together with loop induction variables, the pointers that are repeatedly updated to access (various) elements of an array.

Prioritize, Prioritize, Prioritize...

After obtaining a set of vulnerability candidates, Dowser prioritizes them according to the 80-20 Pareto principle: we want to discover the majority of software bugs while testing only a subset of the potentially vulnerable code fragments. While all the array accesses that evolve with induction variables are potential targets, Dowser prioritizes them according to the complexity of the data- and control-flows for the array index (pointer) calculations.

For each candidate loop, it first statically determines (1) the set of all instructions involved in modifying an array pointer (we will call this a pointer's analysis group), and (2) the conditions that guard this analysis group (for example, the condition of an if or while statement containing the array index calculations). Next, it labels all such sets with scores reflecting their complexity. It may happen that the data-flow associated with an array pointer is simple, but the value of the pointer is hard to follow due to some complex control changes. For this reason, Dowser also considers the complexity of the variables involved in conditionals. For a detailed description of the procedure, refer to [6].

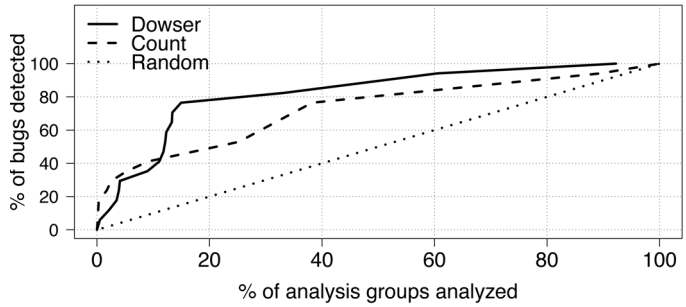

Figure 1: A comparison of random testing and two scoring functions: Dowser's and count. It illustrates how many bugs we detect if we test a particular fraction of the analysis groups.

We emphasize that our complexity metric is not the only way to rank the buffer accesses. For instance, we could also use the length of a loop, the number of computations involved in the computation of the array index, or some other heuristic. In fact, Dowser does not care which ranking function is used, as long as it prioritizes the accesses in the best possible way. In our lab, we have evaluated several such functions and, so far, the complexity metric performed best. For instance, Figure 1 compares Dowser's complexity metric to count, a straightforward scoring function that simply counts the number of instructions involved in the computation of the array pointer.

We base the evaluation on a set of known vulnerabilities from six real world programs: nginx, ffmpeg, inspircd, libexif, poppler, and snort. Additionally, we consider the vulnerabilities in sendmail tested by Zitser et al. [10]. For these applications, we analyzed all buffer overflows reported in CVE since 2009 to find 17 that match our vulnerability model. Figure 1 illustrates the results. Random ranking serves as a baseline; clearly both count and Dowser perform better. In order to detect all 17 bugs, Dowser must analyze 92.2% of all the analysis groups; however, even with only 15% of the targets, we find almost 80% (13/17) of all the bugs. At that same fraction of targets, count finds a little more than 40% of the bugs (7/17). Overall, Dowser outperforms count beyond the 10% in the ranking, and it reaches the 100% bug score earlier than the alternatives, although the difference is minimal.

Efficient Spot-Checking

The main purpose of spot-checking is to avoid the complexity stemming from whole-program testing. For example, the nginx-0.6.32 Web server [1] contains a buffer underrun vulnerability, where a specially crafted input URI tricks the program into setting a pointer to a location outside its buffer boundaries. When this pointer is later used to access memory, it allows attackers

to overwrite a function pointer and execute arbitrary code on the system. Exhaustively testing the Web server to find this bug is almost impossible due to the complexity of the HTTP packets used as input. Indeed, the existing tools didn't discover the vulnerability within eight hours. Dowser, however, ranked the vulnerable array access at the fourth most complex out of a total of 62 potentially vulnerable loops, and then found the bug within five minutes.

As a baseline, spot-checking uses concolic execution [5], a combination of concrete and symbolic execution, where the concrete (fixed) input starts off the symbolic execution. Dowser enhances concolic execution with the following two optimizations.

Finding Relevant Inputs

Typically only a part of the input influences a particular analysis group. In our example, only the URI field from the HTTP packet is relevant to the faulty parser. Dowser aims to identify and enforce this correlation automatically. In technical terms, Dowser uses dynamic taint analysis to determine which input fields influence pointers dereferenced in the analysis group. During the testing phase, Dowser only treats those fields as symbolic and keeps the remaining ones unchanged.

Eliminating Irrelevant Code

The second optimization leverages the observation that only the program instructions that influence the underlying pointer arithmetic are relevant to buffer overflows. Thus, when checking a particular spot, that is, a buffer access, Dowser analyzes the associated loop a priori to find branch outcomes that are most likely to lead to new pointer values. The results of this analysis are used to focus the testing effort around the most relevant program paths. In the URI parser example, it would prioritize branches that impact pointer arithmetic, and ignore those that only affect the parsing result.

Dowser's loop exploration procedure operates in two main phases: learning and bug finding. In the learning phase, Dowser assigns each branch a weight approximating the probability that a path following this direction contains new pointer dereferences. The weights are based on statistics of pointer value variance observed during symbolic execution with limited inputs.

In the bug finding phase, Dowser symbolically executes a real-world-sized input in the hope of finding inputs that trigger a bug. Dowser uses the weights from the learning phase to steer its symbolic execution toward new and interesting pointer dereferences. The goal of our heuristic is to avoid execution paths that are redundant from the point of view of pointer manipulation. Thus, Dowser shifts the target of symbolic execution from traditional code coverage to pointer value coverage. Therein lies the name we gave to this new search heuristic, Value Coverage Search, to emphasize the data-centric approach that Dowser takes.

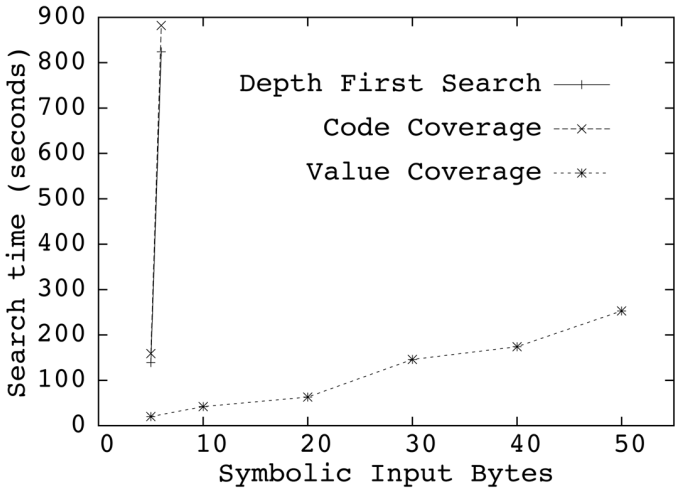

Figure 2: A comparison of the different search heuristics while testing for the vulnerability in nginx. In all instances the symbolic input is limited to the URI field.

We highlight the benefits gained via spot-checking on the same nginx example used so far. As mentioned in the beginning of this section, the application itself is too complex for the baseline concolic execution engine, which was unable to trigger the bug within eight hours. Limiting the symbolic input to the given URI field does allow S2E to detect the bug using its built-in search heuristics (Depth-First Search and Code Coverage), as we show in Figure 2; however, the reader can also notice an exponential explosion in the search time, making the traditional search heuristics inefficient when the input size grows beyond six bytes. Although many tools recommend code coverage [5] as the primary strategy to find bugs, in our experience it does not help with buffer overflows, because memory corruptions require a particular execution context. Even if 100% code coverage is reached, these bugs may stay undetected. In contrast with these results, our Value Coverage heuristic shows excellent scalability with an almost linear increase in execution time in relation with the input size.

Dowser in the Real World

Dowser detected nine memory corruptions from six real-world applications of several tens of thousands LOC, including the ffmpeg videoplayer of300k LOC. The other applications that we looked at were nginx, inspircd, poppler, libexif, and snort. The bug in ffmpeg and one of the bugs in poppler were also not documented before. We run S2E for as short a time as possible, (e.g., a single request/response in nginx and transcoding a single frame in ffmpeg). Still, in most applications, vanilla S2E fails to find bugs within eight hours, whereas Dowser is always capable of triggering the bug within 15 minutes of testing the appropriate

analysis group. More details about the evaluation can be found in our paper [6].

Although our paper applies dowsing to the concrete class of buffer overflows, the underlying principles are also valid for a wide variety of bug families. Once we identify the unique feature set characterizing each of them, we will be able to discover more vulnerable locations. Recent developments in the analysis of legacy binaries also suggest that the techniques required by Dowser may soon be applicable without the need of source code information. Such developments would enable the efficient testing of legacy binaries to learn about possible zero-day attacks within.

Conclusion

Dowser is a guided fuzzer that combines static analysis, dynamic taint analysis, and symbolic execution to find buffer overflow vulnerabilities deep in a program's logic. It leverages a new testing approach, called dowsing, that aims to actively search for bugs in specific code fragments without having to deal with the complexity of the whole binary. Dowser is a new, practical, and complete fuzzing approach that scales to real applications and complex bugs that would be hard or impossible to find with existing techniques.

Acknowledgments

This work is supported by the European Research Council through project ERC-2010-StG 259108-ROSETTA, the EU FP7 SysSec Network of Excellence and by the Microsoft Research PhD Scholarship Program through the project MRL 2011-049. The authors would like to thank Bartek Knapik for his help in designing the statistical evaluation.

References

[1] CVE-2009-2629: Buffer Underflow Vulnerability in Nginx: http://Cve.Mitre.Org/Cgi-Bin/Cvename.Cgi?Name =CVE-2009-2629, 2009.

[2] J. P. Anderson, "Computer Security Technology Planning Study," Tech. Rep., Deputy for Command and Management System, USA, 1972.

[3] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (2008), OSDI '08.

[4] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems," Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (2011), ASPLOS '11.

[5] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated Whitebox Fuzz Testing," Proceedings of the 15th Annual Network and Distributed System Security Symposium (2008), NDSS '08.

[6] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," Proceedings of USENIX Security '13 (Washington, DC, August 2013), USENIX.

[7] C. Lattner and V. Adve, "Llvm: A Compilation Framework for Lifelong Program Analysis and Transformation," Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO '04) (Palo Alto, California, March 2004).

[8] V. van der Veen, N. Dutt-Sharma, L. Cavallaro, and H. Bos, "Memory Errors: The Past, the Present, and the Future," Proceedings of the 15th international Symposium on Research in Attacks, Intrusions and Defenses (2012), RAID '12.

[9] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista," Proceedings of the 3rd international Conference on Software Testing, Verification and Validation (April 2010), ICST '10.

[10] M. Zitser, R. Lippmann, and T. Leek, "Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code," Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (2004), SIGSOFT '04/FSE-12.