Automated Attack Surface Approximation

Christopher Theisen
North Carolina State University
Department of Computer Science
890 Oval Drive, #8206 Raleigh, NC
+1 919 515 2858
crtheise@ncsu.edu

ABSTRACT

While software systems are being developed and released to consumers more rapidly than ever, security remains an important issue for developers. Shorter development cycles means less time for these critical security testing and review efforts. The *attack surface* of a system is the sum of all paths for untrusted data into and out of a system. Code that lies on the attack surface therefore contains code with actual exploitable vulnerabilities. However, identifying code that lies on the attack surface requires the same contested security resources from the secure testing efforts themselves. My research proposes an automated technique to approximate attack surfaces through the analysis of stack traces. We hypothesize that stack traces user crashes represent activity that puts the system under stress, and is therefore indicative of potential security vulnerabilities. The goal of this research is *to aid software engineers in prioritizing security efforts by approximating the attack surface of a system via stack trace analysis.* In a trial on Mozilla Firefox, the attack surface approximation selected 8.4% of files and contained 72.1% of known vulnerabilities. A similar trial was performed on the Windows 8 product.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *complexity metrics, process metrics, product metrics*

General Terms

Management, Measurement, Design, Economics, Security.

Keywords

Stack traces, crash dumps, attack surface.

1. INTRODUCTION

Howard et al. introduced the concept of an attack surface, describing entry points to a system that might be vulnerable along three dimensions: targets and enablers, channels and protocols, and access rights [1]. Later, Manadhata and Wing [2] formalized the notion of attack surface, including methods, channels, untrusted data, and a direct and indirect entry and exit point framework that identifies methods through which untrusted data passes.

We still lack a practical means of identifying the parts of the system that are contained on the attack surface. If generating the attak surface of a system was a more straightforward process, security professionals could focus their efforts on code on the attack surface because it contains vulnerabilities that are reachable, and therefore exploitable, by malicious users. Code not on the attack surface may contain latent vulnerabilities, but these are unreachable by malicious users. By prioritizing security efforts for code on the attack surface, security professionals could find vulnerabilities more efficiently.

We propose *attack surface approximation*, an automated approach to identifying parts of the system that are contained on the attack surface through stack trace analysis. We parse stack traces, adding all code found in these traces onto the attack surface approximation. By definition, code that appears in stack traces caused by user activity is on the attack surface because it appears in a code path reached by users. The goal of this research is *to aid software engineers in prioritizing security efforts by approximating the attack surface of a system via stack trace analysis*.

We hypothesize that stack traces from user-initiated crashes have three desirable attributes for measuring attack surfaces: (a) they represent user activity that puts the system under stress; (b) they include both direct and indirect entry points; and (c) they provide automatically generated control and data flow graphs. We seek to assess the degree to which these attributes of stack traces support the identification of attack surfaces. We call our approach attack surface *approximation* because code entities will only be added to the attack surface when a crash has occurred. As such, the attack surface approximation will evolve over time. We assess our approach by analyzing the percentage of actual reported vulnerabilities in the code and whether they occur in our approximated attack surface.

2. BACKGROUND AND RELATED WORK

In this section, we provide a brief overview of related work and the previous study done in this area.

2.1 Attack Surface

Howard et al. [1] provided a definition of attack surface using three dimensions: targets and enablers, channels and protocols, and access rights. Not all areas of a system may be directly or indirectly exposed to the outside. Some parts of a complex system, e.g. Windows OS, may be for internal use only and cannot be reached or exploited by an attacker. For example, installation routines are left in the system after initialization, but they are never accessed again and are unlikely to have security implications for the system.

Manadhata et al. [17] describe how an attack surface might be approximated by looking at API entry points. However, this approach does not cover all exposed code, as the authors mention. Specifically, internal flow of data through a system could not be identified. While the external points of a system are a useful place to start, they do not encompass the entirety of exposed code in the system. These intermediate points within the system could also contain security vulnerabilities that the reviewer should be aware of. Further, their approach to measuring attack surfaces required expert judgment and manual effort.

2.2 Using Crash Reports

The use of crash reporting systems, including stack traces from the crashes, is becoming a standard industry practice[1] [8][10]. Bug

[1] http://www.crashlytics.com/blog/its-finally-here-announcing-crashlytics-for android/

reports contain information to help engineers replicate and locate software defects. Liblit and Aiken [4] introduced a technique automatically reconstructing complete execution paths using stack traces and execution profiles. Later, Manevich et al. [5] added data flow analysis information on Liblit and Aiken's approach. Other studies use stack traces to localize the exact fault location [6][7][8]. Lately, an increasing number of empirical studies use bug reports and crash reports to cluster bug reports according to their similarity and diversity, e.g. Podgurski et al. [9] were among the first to take this approach. Other studies followed [10][11]. Not all crash reports are precise enough to allow for this clustering. Guo et al. [12] used crash report information to predict which bugs will get fixed. Bettenburg et al. [13] assessed the quality of bug reports to suggest better and more accurate information helping developers to fix the bug.

With respect to vulnerabilities, Huang et al. [14] used crash reports to generate new exploits while Holler et al. [15] used historic crashes reports to mutate corresponding input data to find incomplete fixes. Kim et al. [16] analyzed security bug reports to predict "top crashes"—those few crashes that account for the majority of crash reports—before new software releases. As mentioned previously, we expanded on previous studies by exploring the correlation between code appearing in a stack trace and having historical vulnerabilities [1].

3. APPROACH

Stack traces are used in attack surface approximation to determine what code is on the attack surface. If a code artifact appears on any stack trace produced by user activity on a system, than then that code artifact is placed on the attack surface approximation. A visualization of this effect is seen in Figure 1. Consider the entire graph as a representation of a call graph of a software system, while the red nodes on the graph are code that is seen on stack traces for the system. Security efforts should therefore be targeted towards the area that is under threat.

Why stack traces? Stack traces represent user activity that puts the system under stress. Stack traces are already used by attackers to determine where to focus penetration testing efforts. If a section of a system can be reliability induced to crash, then there is already a known flaw on that path. That path might also contain an exploitable flaw.

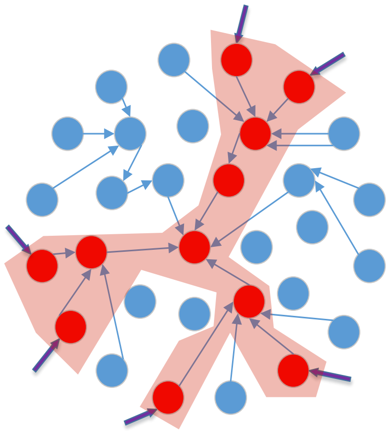

Figure 1: A visual representation of what an attack surface is for a system; the shaded area is the attack surface, where input flows through the system.

Table 1: Results of our attack surface approximation analysis for Mozilla Firefox

	files	flaws	%files	%flaws	Precision	Recall
>= 1	4998	282	8.4%	72.1%	0.056	0.721
>= 10	2691	239	4.5%	61.1%	0.089	0.611
>= 30	1853	210	3.1%	53.7%	0.113	0.537
>= 77	1244	187	2.1%	47.8%	0.150	0.478
>= 140	969	162	1.6%	41.4%	0.167	0.414
All	59437	391	-	-	-	-

In order to accomplish our goal of an automated attack surface approximation technique, the approach needs to be simple, repeatable, and demonstrably effective on multiple software systems. A feasibility study has already been performed [1], and there is ongoing effort to further expand on the attack surface metrics developed during that work.

4. CURRENT RESULTS

In the previous attack surface approximation study [1], we found a correlation between code artifacts that appear on stack traces generated by the system and where historical vulnerabilities discovered by security professionals have been fixed in code. The attack surface correlation could be useful to security professionals when targeting security reviews of codebases. By targeting security efforts at these exposed areas instead of the entire codebase, security professionals could save many engineering hours. In the previous study, it was found that 48.4% of binaries in Windows contained 94.8% of historically seen vulnerabilities [1]. Limiting security engineering efforts to half of the codebase while still finding the vast majority of potential bugs is a tradeoff teams can now make.

Attack surface approximation has also been replicated for a publication under review on the Mozilla Firefox product. In Table 1, we see the results of the Firefox study. We set five cutoffs for minimum number of appearances on stack traces for inclusion on the attack surface approximation, then determine what percentage of vulnerabilities are in the approximation. As seen in the table, increasing the cutoff shrinks our approximation, causing fewer vulnerabilities to appear on the approximation. However, the density of these vulnerabilities increases. This suggests that stack trace frequency may be a good metric for security efforts, not just reliability efforts.

During the course of this work, the value of visualizations of the attack surface of a system has become apparent when talking to professionals in the field. To that end, we have developed a series of visualizations for attack surface approximation, including the graph representation previously seen in Figure 1.

5. CONTRIBUTIONS

Attack surface approximation may create several positive impacts on the software engineering community. An automated approach to attack surface generation could save many engineering hours for organizations, as they would not need to tie up resources developing the understanding of the attack surface themselves. For organizations without a security team at all, automatic attack surface approximation gives them the first steps toward targeting what limited security resources they have internally or externally.

6. REFERENCES

[1] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams, “Approximating Attack Surfaces with Stack Traces”, in *Companion Proceedings of 37th International Conference on Software Engineering,* 2015.

[2] P. Manadhata and J. Wing, "An Attack Surface Metric," Software Engineering, IEEE Transactions on, vol. 37, no. 3, pp. 371-386, 2011.

[3] M. Howard, J. Pincus and J. M. Wing, "Measuring Relative Attack Surfaces," in *Computer Security in the 21st Century*, Springer US, 2005, pp. 109-137.

[4] B. Liblit and A. Aiken, "Building a Better Backtrace: Techniques for Postmortem Program Analysis," University of California, Berkeley, Berkeley, 2002.

[5] R. Manevich, M. Sridharan, S. Adams, M. Das and Z. Yang, "PSE: Explaining Program Failures via Postmortem Static Analysis," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, Newport Beach, CA, USA, 2004.

[6] W. Jin and A. Orso, "F3: Fault Localization for Field Failures," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013.

[7] R. Wu, H. Zhang, S.-C. Cheung and S. Kim, "CrashLocator: Locating Crashing Faults Based on Crash Stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014.

[8] S. Wang, F. Khomh and Y. Zou, "Improving bug localization using correlations in crash reports," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, 2013.

[9] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun and B. Wang, "Automated support for classifying software failure reports," in *Software Engineering, 2003. Proceedings. 25th International Conference on*, 2003.

[10] Y. Dang, R. Wu, H. Zhang, D. Zhang and P. Nobel, "ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[11] S. Kim, T. Zimmermann and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, 2011.

[12] P. J. Guo, T. Zimmermann, N. Nagappan and B. Murphy, "Characterizing and Predicting Which Bugs Get Fixed: An Empirical Study of Microsoft Windows," in *Proceedings of the 32th International Conference on Software Engineering*, 2010.

[13] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj and T. Zimmermann, "What makes a good bug report?," in *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008.

[14] S.-K. Huang, M.-H. Huang, P.-Y. Huang, H.-L. Lu and C.-W. Lai, "Software Crash Analysis for Automatic Exploit Generation on Binary Programs," *Reliability, IEEE Transactions on,* vol. 63, pp. 270-289, March 2014.

[15] C. Holler, K. Herzig and A. Zeller, "Fuzzing with Code Fragments," in *Proceedings of the 21st USENIX Conference on Security Symposium acmid = 2362831*, 2012.

[16] D. Kim, X. Wang, S. Kim, A. Zeller, S. Cheung and S. Park, "Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts," *Software Engineering, IEEE Transactions on,* vol. 37, no. 3, pp. 430-447, 2011.

[17] Manadhata, P., Wing, J., Flynn, M., & McQueen, M. (2006, October). Measuring the attack surfaces of two FTP daemons. In *Proceedings of the 2nd ACM workshop on Quality of protection* (pp. 3-10). ACM.