Reusing Stack Traces: Automated Attack Surface Approximation

Christopher Theisen
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA
crtheise@ncsu.edu

***Abstract* —Security requirements around software systems have become more stringent as society becomes more interconnected via the Internet. New ways of prioritizing security efforts are needed so security professionals can use their time effectively to find security vulnerabilities or prevent them from occurring in the first place. The goal of this work is *to help software development teams prioritize security efforts by approximating the attack surface of a software system via stack trace analysis.* Automated attack surface approximation is a technique that uses crash dump stack traces to predict what code may contain exploitable vulnerabilities. If a code entity (a binary, file or function) appears on stack traces, then Attack Surface Approximation (ASA) considers that code entity is on the attack surface of the software system. We also explore whether number of appearances of code on stack traces correlates with where security vulnerabilities are found. To date, feasibility studies of ASA have been performed on Windows 8 and 8.1, and Mozilla Firefox. The results from these studies indicate that ASA may be useful for practitioners trying to secure their software systems. We are now working towards establishing the ground truth of what the attack surface of software systems is, along with looking at how ASA could change over time, among other metrics.**

***Index Terms*—Security, Attack Surface, Security Metrics, Stack Traces, Crashes.**

I. Technical Problem

One of the ways security professionals identify potentially vulnerable code is the concept of the attack surface of a software system. Howard et al. [1] described the attack surface as a measure of "attackability" of a software system, along three dimensions: targets and enablers, channels and protocols, and access rights. The concept of the attack surface of a system has been used previously in the context of shrinking the attack surface of a system. Geer explores the concept of limiting attack vectors using the example of two PDF readers: Adobe Reader and Foxit Reader [2]. Foxit Reader reduces its attack surface by making its document reader functionality available via plugins, which can be disabled by users if they aren't currently using that feature.

While these concepts are helpful for attempting to prevent vulnerabilities from being discovered by attacks, they do not address finding or preventing vulnerabilities in code that must be exposed for software to function. Security hardening efforts on software systems help software security professionals identify and fix vulnerabilities before malicious parties do – or even help them to engineer code in a secure manner so vulnerabilities are never inadvertently created in the first place. In previous work, we explored how stack traces from crash dumps can be used to approximate the attack surface of software systems, specifically Windows 8 [3]. In the Windows 8 study, we found that 48.4% of binaries were seen on at least one stack trace from Windows 8 crashes. At the same time, 94.8% of the code that vulnerabilities were fixed in was in the same 48.4% subset of code. In another study on Mozilla Firefox [4], 8.4% of files appeared on at least one stack trace, while 72.1% of that subset of files had vulnerabilities that were fixed. The result suggests that security professionals may be well served by focusing security rework efforts on the subset of code appearing on stack traces, saving effort in the security hardening effort space. Reducing the amount of code to be inspected may help improve the economics of security assessments and allow for more proactive reviews of potentially vulnerable code. Further exploration of the use of stack trace code entities as a metric for security efforts may be useful in the prioritization of practitioner's efforts in securing software systems. We call a stack trace-based approach *Attack Surface Approximation* (ASA).

The goal of this work is *to help software development teams prioritize security efforts by approximating the attack surface of a software system via stack trace analysis.* By further exploring the idea of using stack traces from crash dumps to determine where security vulnerabilities might be, we can show how robust the process might be for security vulnerability identification, along with the practicality of the approach compared to the current ground truth in attack surface identification. We plan to explore additional metrics derived from crash dump stack traces in order to assist security professionals in their efforts.

Our expected contributions include:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ICSE '16 Companion, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4205-6/16/05 $15.00
DOI: http://dx.doi.org/10.1145/2889160.2889263

- A practical measure of the attack surface of software systems, using code entities on stack traces from crashes as the primary metric.
- A determination of the ground truth of the attack surface of software systems is, and how it compares to ASA.
- An exploration of the generalizability of ASA, and toolsets to help practitioners apply ASA to their own software systems.

II. Research Questions

The research questions are as follows:

RQ1: How does attack surface approximation compare with the "ground truth" of the attack surface?

RQ2: How does the result of attack surface approximation change over time, and with different amounts of available data?

RQ3: Is attack surface approximation a practical, robust, and effective approach for prioritization of security efforts?

III. Background and Related Work

In this section, we present background information on the definition of attack surface, use of stack traces as a software development metric, and current work in attack surface metrics.

A. Attack surface

The attack surface of an application, as defined by the Open Web Application Security Project (OWASP) [5] is:

1. The sum of all paths for data/commands into and out of the application,
2. The code that protects these paths,
3. All valuable data used in the application, including secrets and keys, intellectual property, critical business data, personal data, and personally identifiable information (PII),
4. The code that protects these data.

Some examples of resources that comprise a system's attack surface include the following (but is not an exhaustive list) [6]: open ports, services available on the inside of the firewall, code that processes incoming data, and user interface forms and fields. An attack can use these and other resources to attack a software system. However, this definition of attack surface only focuses on configuration of systems, and is used in practice to limit the attack surface of software systems as much as possible. Practically, some of these attack vectors much be left open in order for software to function for users.

Research efforts into the configuration definition of attack surface include Heumann et al.'s work on the *attack surface indicator* (ASI) metric [7]. ASI is an aggregation of several metrics of web applications that affect the attack surface, such as URL parameters, file upload fields, search fields, and number of domains. ASI provides a picture of the deployed application's attack surface rather than the software system itself. For example, if a single web application was deployed on multiple different servers, various configuration permutations for an application could result in two completely different ASI values for the software system.

From a software engineering perspective, Howard et al. [1] described the attack surface along three dimensions. *Targets and enablers* refer to the assets attackers want to access, and the resources they use to access them. *Channels and protocols* are the messaging structure software uses to pass messages to one another. *Access rights* give legitimate users of a software system access to data. While the Howard definition is a good definition of attack surface, the work does not provide a method for practitioners to apply the definition to their own products.

B. Crash dump stack traces as a metric

The focus of most crash reporting systems is to identify why a software system has crashed. Examples include CrashLocator by Wu et al. [8], which uses stack traces from crashes to narrow down the location of the fault in the code that caused the crash. ReBucket by Dang et al. [9] clusters crash reports by similarities in the attached stack traces, aggregating reports for engineers to triage. Both of these tools are examples of crash reports being processed by researchers to provide benefits for practitioners trying to find defects in their software. Other researchers have built tools to determine where the exact fault location is based on stack traces from crashes. Jin and Orso [10] built F3, a fault localization tool for failure that describes where the final fault is from crash information.

C. Attack surface metrics

Manadhata et al. [11] performed early work on approximating the attack surface of software system. By scanning all API entry points into a system, the researchers created their own approximation of the attack surface of the software system they examined. Manadhata et al.'s approach has several drawbacks. First, their approach only covers publicly disclosed entry points, and do not cover paths through the system or exit points. While API scanning is a useful place to start, a more complete picture of the attack surface is needed. Second, the API scanning approach only covers entry points into a system, ignoring the paths data takes within the system. Identifying the entry and exit points of the attack surface of a software system is not enough. To properly apply a "defense-in-depth" strategy for protecting software systems, knowing the paths data takes through the software system is necessary so those paths can be hardened against attack.

As mentioned previously, ASA is an approach for approximating the attack surface of software systems by looking at crash dump stack traces from the system. Our first results from the Windows 8™ operating system (OS) [3] revealed a correlation between binaries that appear on crash dump stack traces generated by the system and historical vulnerabilities discovered by security professionals that have been fixed in the code. The correlation could be useful to security professionals when targeting security reviews and testing of code bases. The effectiveness of ASA was analyzed by comparing the approximation of the attack surface against

the location of historical vulnerabilities in Windows 8 OS. The result revealed that 48.4% of shipped binaries seen in at least one crash dump stack trace in Windows 8 OS contained 94.8% of the vulnerabilities seen over the same time period. We created a vulnerability prediction model (VPM) based on previous VPM work by Zimmermann et al. [12], though these results had issues with precision and recall. [3]. Precision of the VPM was 0.69, while recall was 0.04.

IV. Methodology and Evaluation

In this section, we explore the process of evaluating each of the research questions established in Section II.

A. How does attack surface approximation compare with the "ground truth" of the attack surface? (RQ1)

Figure 1 represents a visualization of a software system as a graph, with individual nodes being code entities in the system. The API nodes should be identified by the Manadhata approach via their API scanning technique [13], but may miss the intermediate code entities (colored in red). ASA aims for a "defense-in-depth" approach, where these intermediate code entities are just as important from a security perspective.

Before additional work in the area of ASA can be completed, we must establish how ASA relates to the ground truth of what the attack surface of software systems actually is. While there have been theoretical attempts to define what the attack surface of a software system is [1], to the author's knowledge there is little literature of these approaches applied to actual software systems.

A suitable software system that ASA, the Manadhata approach, and the ground truth can all be executed on will be chosen for the experiment. Determination of the "ground truth attack surface" will be performed via the following steps:

1. Identify all functions in the codebase that accept input from locations outside the software system,
2. Recursively, identify all functions called from functions identified in the previous step until all remaining functions call no other functions in the software system,
3. All of the functions identified in steps 1-2 are on the attack surface of the software system.

We will then compare the three approaches across three metrics: *accuracy, detection efficiency, and reachability.*

- The *accuracy* is a comparison of the classification derived by a technique to the ground truth, considering the ground truth as the baseline.

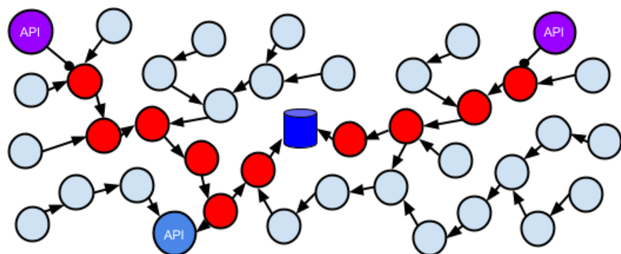

Figure 1: *A visualization of the attack surface of a system.*

- The *detection efficiency* is the time required to do the attack surface analysis for the system. Detection efficiency will be measured by the time it takes to apply the approach to a software system and the level of expertise required to implement it.

- The *reachability* metric indicates the percentage of discovered vulnerabilities that were found on the attack surface.

We will present the results of each approach measured by these metrics. From there, we can make recommendations for researchers and practitioners based on the interaction of these metrics and the needs of each domain.

B. Attack surface metrics (RQ2)

In previous work on ASA [3], the metric for inclusion on the attack surface is an on/off metric. If a code entity appears on at least one stack trace, then we consider it to be on the attack surface of the software system [3]. The original ASA metric may not be an effective approach for identification of the attack surface, as it does not consider possible code changes, resulting in code being removed from the attack surface. To determine the temporal constraints of ASA, we will perform a study that considers the following. In addition, the previous studies used millions of stack traces – 10 million for the Windows 8 study [3], and 1 million for a preliminary study on Mozilla Firefox [14]. For many organizations, millions of stack traces may be unreasonable to collect. Determining whether ASA can be performed with reasonable results when there is orders of magnitude less data available is important for the practicality of the approach.

To answer this research question, we will perform several studies. For temporal constraints, we will answer the following smaller questions:

- How does the ground truth attack surface (described in RQ1) change over a specified time period?
- How does ASA change over the same time period?

To answer the question about the level of data required, we will perform ASA on a chosen software system repeatedly, with different sized random samples of stack traces from the software system. We will then determine the answers to the following questions:

- How many stack traces are necessary to *stabilize* the result (or, at what point are additional stack traces not helping to improve the result)
- How many stack traces are necessary for a *practically* useful result?

To answer the practicality question, we will work with industry and open source partners on their own software systems and get their feedback on the results from these studies.

C. Is Attack Surface Approximation effective? (RQ3)

To show the broad applicability of ASA, many replications are necessary across a variety of domains. By working with a variety of organizations on these replication efforts, we can both show the generalizability of ASA as well as helping these organizations secure their products. Many organizations collect and store stack traces from crash dumps, along with version control information that could be used to replicate ASA. Many organizations also collect data about the security vulnerabilities seen over the lifetime of their products. Using these datasets, we can evaluate ASA in new contexts, and also determine how it could have helped find or prevent security vulnerabilities.

To evaluate the effectiveness of ASA, we can correlate the approximation of the attack surface found by the approach with security vulnerabilities seen in the target software system. We can also correlate any additional metrics we develop around ASA, such as frequency of appearance on stack traces. If ASA provides meaningful feedback to practitioners on where security vulnerabilities have been previously seen *without the approach knowing about the vulnerabilities*, then the approach will be considered effective. As an example: a target software system had 10,000 stack traces from 2014, and 100 security vulnerabilities were fixed the same year. If ASA had a recall of 0.95 with a precision of 0.3, it is reasonable to conclude that 10,000 stack traces from 2015 could provide reasonable coverage of the security vulnerabilities seen in 2015.

V. Current Progress

We have completed a feasibility study into ASA using data from Microsoft's Windows 8 product [3]. The study showed that 48.4% of binaries appear in at least one crash dump stack trace pulled from a sample of 10 million stack traces, while 94.8% of vulnerable binaries fixed over the same time period were in the same subset. In addition to these results, we also determined from interviews and feedback that visualizations of the attack surface of systems could be beneficial for security practitioners. We have submitted publications exploring a replication of ASA at the file level of granularity, with preliminary results appearing in the Student Research Competition at the Foundations of Software Engineering conference in 2015 [14]. A full conference paper on the file level of granularity and ASA has been submitted as well [4]. We have created a set of Python scripts for analysis of stack traces in the context of ASA.

VI. Proposed Work

The first step for continuing the work on ASA is to complete a systematic literature review in the area of attack surface metrics, so the current ground truth of attack surfaces can be properly identified for comparison to ASA. Next, the ground truth of measuring the attack surface needs to be determined. ASA and other attack surface metrics would then be compared against the ground truth to determine the limitations and benefits of each approach as outlined previously. Finally, replications of ASA will be performed as data is made available from industry, academic, and open source partners to further establish the generalizability of ASA.

VII. Acknowledgments

The work in this paper was performed at Microsoft Research UK – Cambridge in the summers of 2014 and 2015, and at North Carolina State University from 2014-2015. The Realsearch group at NCSU has provided feedback throughout this work. My advisor, Dr. Laurie Williams, has provided essential ideas and feedback throughout this process. Brendan Murphy and Kim Herzig of Microsoft Research have also been significant figures throughout the development of ASA.

VIII. References

[1] M. Howard, J. Pincus, and J. M. Wing, "Measuring Relative Attack Surfaces," *Comput. Secur. 21st Century*, no. CMU-TR-03–169, pp. 109–137, 2005.

[2] D. E. Geer, "Attack surface inflation," *IEEE Secur. Priv.*, vol. 9, no. 4, pp. 85–86, 2011.

[3] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams, "Approximating Attack Surfaces with Stack Traces," in *IEEE/ACM 37th IEEE International Conference on Software Engineering*, Florence, Italy 2015.

[4] C. Theisen, R. Krishna, and L. Williams, "Strengthening the Evidence that Attack Surfaces Can Be Approximated with Stack Traces," in *North Carolina State University Department of Computer Science TR2015-10, submitted to International Conference on Software Testing, Verification, and Validation (ICST) 2016*, 2015.

[5] J. Bird and J. Manico, "OWASP Attack Surface Analysis Cheat Sheet," July 18, 2015, https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet.

[6] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Trans. Softw. Eng.*, vol. 37, no. 3, pp. 371–386, May/June 2011.

[7] T. Heumann, S. Türpe, and J. Keller., "Quantifying the Attack Surface of a Web Application," in *GI SICHERHIET 2010*, pp. 305–316, 2010.

[8] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "CrashLocator: locating crashing faults based on crash stacks," in *International Symposium on Software Testing and Analysis (ISSTA)* San Jose, CA, USA, pp. 204–214, 2014

[9] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "ReBucket: A method for clustering duplicate crash reports based on call stack similarity," *International Conference on Software Engineering (ICSE)* Zurich, Switzerland, pp. 1084–1093, 2012.

[10] W. Jin and A. Orso, "F3: Fault Localization for Field Failures," in *International Symposium on Software Testing and Analysis* Lugano, Switzerland, pp. 213–223, 2013.

[11] P. Manadhata, J. Wing, M. Flynn, and M. McQueen, "Measuring the attack surfaces of two FTP daemons," *2nd ACM Workshop on Quality of Protection*, p. 3-10, 2006.

[12] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for Windows Vista," *International Conference on Software Testing, Verification, and Validation*, pp. 421–428, 2010.

[13] P. K. Manadhata and J. M. Wing, "Measuring a System's Attack Surface," *School of Computer Science, Carnegie Mellon University* CMU-CS-04-102, 2004.

[14] C. Theisen, "Automated Attack Surface Approximation," in *The 23rd ACM SIGSOFT International Symposium on the Foundations of Software Engineering - Student Research Competition*, 2015, pp. 1063–1065.