Comparing and Applying Attack Surface Metrics

Jeffrey Stuckman
Department of Computer Science
University of Maryland
College Park, Maryland USA
stuckman@umd.edu

James Purtilo
Department of Computer Science
University of Maryland
College Park, Maryland USA
purtilo@cs.umd.edu

ABSTRACT

A software system's attack surface metric measures the freedom of a potential attacker to influence the system's execution, potentially exploiting a security vulnerability. Existing attack surface metrics aim to measure the security impact associated with deploying an application or component; however, a systematic evaluation of various metrics' suitability for this purpose has not yet been performed. We outline a framework for formalizing code-level attack surface metrics and deployment-time activities that reduce the attack surface of an application. We also outline a tool for measuring the attack surface of a deployed web application, along with a method to retrospectively evaluate an attack surface metric over a corpus of known vulnerabilities.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Product metrics*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Unauthorized access*

General Terms

Measurement, Security

Keywords

Attack surface, evaluation, vulnerability

1. INTRODUCTION

Attack surface metrics quantify the opportunity that an attacker has to compromise the security of a software system. Intuitively, a system with a large attack surface provides many ways for a user to influence its execution with malicious inputs, increasing the likelihood that such a user could induce the system to perform an unwanted operation. While an attack surface metric does not directly measure exploitability (a system with a small attack surface but many vulnerabilities could be more exploitable than a system with a larger attack surface), systems with small attack surfaces would ideally tend to be less exploitable and more secure.

Manadhata and Wing [6] formalized the concept of performing a low-level attack surface measurement of a system, modeling the internal data flows of a composition of components and then measuring attack surface in relation to the system's methods, channels, and data. Their work demonstrated that attack surface can be mechanically measured at the source code level without expert judgement being necessary. Other researchers measured attack surface of web applications by counting elements such as scripts [10], parameters, domains [5], or cookies.

A carefully constructed attack surface metric can help software developers and deployers evaluate their efforts to reduce risk. Developers can engineer architectures that minimize risk and identify features that merit special scrutiny, while deployers can evaluate the effects of turning off unnecessary features, reconfiguring applications, and performing other measures to "harden" their environment. Ideally, an attack surface metric would be correlated with the likelihood that the system contains unknown vulnerabilities that are exploitable by outsiders. However, we are not aware of any previous efforts to systematically relate attack surface to the risk of vulnerabilities and compare alternative attack surface metrics in this way. Furthermore, existing attack surface metrics which are specific to web applications ignore the web application's code structure and data flows, while method and channel-centric attack surface metrics are difficult to apply to page-centric web applications. Hence, to encourage the development and validation of broadly applicable attack surface metrics, we propose a framework for performing an empirical, *retrospective* attack surface metric evaluation.

The evaluation is based on the intuitive notion that the attack surface of a program grows as newly developed features are added. Because the attack surface metric should reflect the risk that these new features introduce security vulnerabilities, an ideal metric will rise more when a feature with many vulnerabilities is introduced than when a feature with few vulnerabilities is introduced. This relationship between a feature's attack surface and vulnerabilities can be validated retrospectively by assembling a corpus of programs with known security vulnerabilities and then measuring the relative changes in attack surface and vulnerability count as features are removed from these programs.

One goal of this work-in-progress is to develop attack surface metrics which are generally applicable without expert judgement and demonstrably correlated with the likelihood of introducing security vulnerabilities. In addition, the tools used to evaluate an attack surface metric could also be used to measure the attack surface of a system under development or deployment, providing developers and system administrators with this guidance.

2. VULNERABILITIES AND RETROSPECTIVE EVALUATION

An evaluation of an attack surface metric begins with assembling a *corpus* of security vulnerabilities. For the purposes of this research, a corpus of vulnerabilities consists of open-source web applications with known security vulnerabilities, along with annotations describing the locations of the vulnerabilities and possible deployment-time configurations and environmental manipulations that can limit access to the application's features. A single corpus could be used to evaluate multiple attack surface metrics for the same programming language. We are currently limiting our study to vulnerabilities in open-source PHP web applications because the source code of many such vulnerabilities is available, archived in old versions of the applications downloadable on the internet [4]. In addition, vulnerabilities in internet-facing web applications are of broad interest because they are prevalent and widespread [12].

Previous research has proposed the use of vulnerability corpora to perform retrospective analysis on security tools, such as intrusion prevention systems [11] and static analysis tools [13]. Although source-level vulnerability data has been successfully mined from software repositories in the past, some difficulties have been noted. Vulnerabilities are enumerated in convenient online databases such as the OSVDB [9] and National Vulnerability Database [8]; however, using these databases for research can be difficult because different sources of vulnerability information may provide different data [7] and code repositories may mix security and non-security revisions [3]. To alleviate these difficulties, we are currently developing a semi-automated vulnerability data collection tool, which assists the user in the course of collecting application vulnerability information, verifying that the vulnerability is present in the given application, capturing code paths characteristic to an exploit, and annotating the application with possible configurations and environmental manipulations.

One naïve approach for validating an attack surface metric is to correlate the number of known vulnerabilities in a program with the program's attack surface. Such an approach is unsound because it assumes *complete* knowledge of a program's vulnerabilities and it ignores *confounding factors* influencing the absolute number of vulnerabilities in a program. We cannot guarantee that all vulnerabilities in a particular program have been discovered (no matter how popular the program is), and two groups of developers with differing skill implementing the same design may create two programs with the same attack surface but different vulnerabilities. Our proposed attack surface metric evaluation works on a *sample* of vulnerabilities, without having to know the absolute number of vulnerabilities found (or present) in any program in the corpus (although to ensure the validity of the evaluation, the sample from the population of all reported vulnerabilities should be as random or representative as possible).

3. PROGRAM CONSTRAINTS AND SPECIALIZATION

To evaluate an attack surface metric, we compare the attack surfaces of a program before and after applying *constraints*. A constraint dictates the set of possible values for variables that are derived from a program's environment, such the contents or attributes of files, the data in a database or configuration repository, or cookies received from the user. Constraints are expressed as symbolic constraints on a certain line of code. Applying constraints limits the state space of the program, and portions of the program may become unreachable. Because vulnerabilities in the corpus are associated with portions of the program, a vulnerability can disappear if the vulnerability's code lines or paths become unreachable. When a vulnerability is defined by a set of code paths (as opposed to lines) and there are multiple paths possible (which is often the case [2]), all paths must become infeasible to eliminate the vulnerability.

Authentication constraints express that a potential attacker possesses (or does not possess) a certain authentication credential, such as a password or cookie. Authentication constraints specialize an attack surface measurement for a specific class of user – for example, some functionality may only be visible to a system administrator, who already has access to the entire system and need not exploit any vulnerabilities. Such functionality should not be incorporated into an attack surface measurement if it is provably unreachable by non-administrators. *Configuration and environmental constraints* describe activities that deployers can perform to disable features of a program and reduce the program's attack surface. For example, in a content management system, a deployer could disable blog commenting in a configuration file (a configuration constraint) or simply ensure that the database never has any blog posts (an environmental constraint), rendering the blog comment processing code unreachable. Deployers could also protect a web application with a firewall (another environmental constraint), preventing certain URLs from being accessed. Ultimately, constraints represent efforts by system administrators to "harden" programs and reduce their attack surfaces.

3.1 Analyzing program data flow and potential configuration constraints

Some authentication and environmental constraints are explicitly enumerated by the user when running an attack surface analyzer; however, configuration constraints would be difficult to manually enumerate due to the large number of configuration settings and possible values. For this reason, a data flow analysis is performed for each program to enumerate potential configuration constraints and analyze the program's data flow:

1. Construct a control flow graph for the program.

2. Manually annotate embedded configuration variables (which allow PHP applications to be configured by editing constants in source files) as configuration constraints on the control flow graph, binding a program variable to a configuration setting.

$$\forall c, \mathit{surface}\left(\mathit{prog}\right) \supseteq \mathit{surface}\left(\mathit{prog}_c\right) \Rightarrow \forall c, \mathit{metric}\left(\mathit{prog}\right) \geq \mathit{metric}\left(\mathit{prog}_c\right) \tag{1}$$

$$r = \mathit{corr}\left(\bigcup_{\alpha \in A_{user}(\mathit{prog})} \bigcup_{\beta \in C(\mathit{prog}) \cup E(\mathit{prog})} \left(\frac{\mathit{metric}\left(\mathit{prog}_\alpha\right) - \mathit{metric}\left(\mathit{prog}_{\alpha\beta}\right)}{\mathit{metric}\left(\mathit{prog}_\alpha\right)}, \frac{\mathit{vul}\left(\mathit{prog}_\alpha\right) - \mathit{vul}\left(\mathit{prog}_{\alpha\beta}\right)}{\mathit{vul}\left(\mathit{prog}_\alpha\right)}\right)\right) \tag{2}$$

3. Identify nodes on the control flow graph that prepare or execute a SQL statement returning a result set, and annotate each node with a constraint expressing that the result set will be empty unless a corresponding predicate holds over the database's data.

4. Annotate control flow graph nodes which are directly affected by potential environmental constraints. This can be done manually or with a rule defining that a property of a resource in the environment applies throughout the program.

5. Annotate control flow graph nodes which are directly affected by potential authentication constraints. This is done by locating nodes reading cookies or parameters used for authentication and annotating them with a constraint equivalent to the appropriate authentication predicate.

6. On the control flow graph, propagate all annotations across variable assignments, database resultset operations, and accesses to cookie or parameter maps.

7. To enumerate potential configuration constraints, find control flow graph nodes where propagated configuration values are compared with constants. For each configuration variable, select a set of potential values such that each such comparison will pass and fail at least once. Add these values as potential configuration constraints. Also add the database data predicates previously determined (along with the negations of such predicates).

3.2 Program specialization

Eliminating lines of code rendered unreachable by constraints is commonly known as *program specialization*. In this paper, a program prog specialized under constraint c is represented as prog_c. After analyzing the program's data flow and enumerating potential constraints, the program can be specialized under a constraint as follows:

1. Search the control flow graph for annotations related to the entity which is actually being constrained. Replace the symbolic constraint with the actual value to which the object is constrained, effectively fixing the value of a variable at some point in the program.

2. Find expressions in the program where the value of a fixed variable is tested. Annotate control flow graph nodes where a test always evaluates to true (or false) due to a fixed variable value.

3. Remove branches which are unreachable due to a fixed variable value. Remove any resulting dead code (unreachable control flow graph nodes) as well.

4. COMPUTING AN ATTACK SURFACE METRIC

Attack surface can be measured by enumerating *reachable elements* that contribute to the attack surface. An *element*, as defined by this research, represents an opportunity for a potential attacker to influence a program's data or control flow. An element is reachable when it remains after specializing the program under a constraint. Generalizing the security "principle of least privilege" commonly applied to operating systems, an element represents a user's privilege to induce a particular execution path or inject a value that the program uses in a particular way. Possible elements include, but are not limited to:

- Access to input channels or methods; access to files and other persistent data items [6]
- Reachable lines of code; feasible branches or paths; feasible paths involving output to persistent storage or another user
- Access to input vectors such as URL parameters and forms; ability to induce the execution of AJAX and other client side content [5]
- Control over outputs which are stored and later used as inputs to another module [1]

We represent the set of (reachable) attack surface elements in a (potentially specialized) program prog as $\mathit{surface}\left(\mathit{prog}\right)$. Each attack surface element e is assigned a *score* (some function returning a positive number defined by the specific attack surface metric in question) $\mathit{score}\left(e\right)$, representing its contribution to the attack surface metric. Therefore, the attack surface metric of a program prog under authentication constraint a is:

$$\mathit{metric}\left(\mathit{prog}_a\right) = \sum_{e \in \mathit{surface}(\mathit{prog}_a)} \mathit{score}\left(e\right)$$

Note that elements will normally be tied to specific lines, variables, or objects of a program. Because specialization eliminates portions of a program rendered unreachable under constraints, a specialized program's attack surface will not be greater than that of the original program (Equation 1), except in the case of non-standard metrics where an element can exist in a specialized program but not the original program.

5. EVALUATING AN ATTACK SURFACE METRIC

We consider a "good" attack surface metric to be one that identifies sections of code that are most likely to eliminate vulnerabilities upon being disabled. Sections of code can be disabled by applying configuration and environmental constraints as described previously. An attack surface metric with this property would be useful to deployers who wish to improve security by disabling features associated with large increases in attack surface, as well to software developers who wish to determine the security impact of a newly added feature.

We represent the possible configuration and environmental constraints for a program *prog* as $C\left(prog\right)$ and $E\left(prog\right)$, respectively. We represent the set of authentication constraints corresponding to ordinary (non-administrator) user roles as $A_{user}\left(prog\right)$. We also represent the set of vulnerabilities in a (potentially specialized) program as $vul\left(prog\right)$. Vulnerabilities may disappear upon program specialization, because vulnerabilities in the corpus are marked as being dependent on the execution of particular lines or path(s), and lines or paths may become unreachable or infeasible upon specialization:

$$\forall c, vul\left(prog\right) \supseteq vul\left(prog_c\right)$$

To retrospectively evaluate an attack surface metric on a program in the corpus, we measure both the attack surface and the number of vulnerabilities under all combinations of authentication and hardening (configuration or environmental) constraints. The quality of the attack surface metric is the Pearson's correlation, r, between the surface measurements and vulnerability counts (Equation 2). Higher values of r indicate that the attack surface metric better identified features in the software where more vulnerabilities fell. An overall score for each attack surface metric can then be calculated by averaging r over each program in the corpus.

6. SUMMARY AND FUTURE WORK

In summary, developers and deployers can use an attack surface metric to evaluate the security impact of adding or disabling features of a program. We proposed a method that can evaluate the suitability of an attack surface metric for this purpose, as long as the metric is expressed as a sum of weighted attack surface elements. We also proposed a method to evaluate the attack surface impact of a configuration or environmental change, which could help system administrators tune their applications to optimize security.

We have designed an algorithm to specialize programs under configuration or environmental constraints, and we are currently developing tools to assist in the compilation of a vulnerability corpus, as part of an initiative to develop a vulnerability dataset for this and other [11] efforts. In the future, we plan to finish compiling a corpus of security vulnerabilities and develop a tool to measure various attack surface metrics of a PHP application, with or without constraints. This tool would then serve the dual purposes of evaluating attack surface metrics and measuring the attack surfaces of individual applications.

7. ACKNOWLEDGMENTS

We thank the United States Office of Naval Research, which partially supported this research under contract N000141210147.

8. REFERENCES

[1] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna. Multi-module vulnerability analysis of web-based applications. In *Proceedings of ACM CCS '07 Conference on Computer and communications security*, pages 25–35, 2007.

[2] D. Brumley, Z. Liang, J. Newsome, and D. Song. Towards practical automatic generation of multipath vulnerability signatures. Technical Report CMU-CS-07-150, CMU, Pittsburgh, PA, USA, 2007.

[3] D. DaCosta, C. Dahn, S. Mancoridis, and V. Prevelakis. Characterizing the 'security vulnerability likelihood' of software functions. In *Proceedings of the IEEE ICSM 2003 International Conference on Software Maintenance*, pages 266–, 2003.

[4] J. Fonseca and M. Vieira. Mapping software faults with web security vulnerabilities. In *Proceedings of IEEE DSN '08 Conference on Dependable Systems and Networks*, pages 257 –266, 2008.

[5] T. Heumann, S. Türpe, and J. Keller. Quantifying the attack surface of a web application. In *ISSE/Sicherheit 2010: Information Security Solutions Europe - Sicherheit, Schutz und Zuverlässigkeit.*, volume P-170 of *Lecture Notes in Informatics (LNI)*, pages 305–316. Gesellschaft für Informatik (GI) e.V., Bonner Köllen Verlag, 2010.

[6] P. Manadhata and J. Wing. An attack surface metric. *Software Engineering, IEEE Transactions on*, 37(3):371 –386, May-June 2011.

[7] F. Massacci and V. H. Nguyen. Which is the right source for vulnerability studies?: An empirical analysis on Mozilla Firefox. In *Proceedings of MetriSec '10 International Workshop on Security Measurements and Metrics*, pages 4:1–4:8. ACM, 2010.

[8] National vulnerability database. http://nvd.nist.gov/.

[9] OSVDB: The open source vulnerability database. http://www.osvdb.org/.

[10] T. Scholte, D. Balzarotti, and E. Kirda. Have things changed now? An empirical study on input validation vulnerabilities in web applications. *Computers & Security*, 31(3):344 – 356, 2012.

[11] J. Stuckman and J. Purtilo. A testbed for the evaluation of web intrusion prevention systems. In *Proceedings of MetriSec 2011 International Workshop on Security Measurements and Metrics*, pages 66 –75. IEEE, Sept 2011.

[12] Top cyber security risks. (SANS Institute) http://www.sans.org/top-cyber-security-risks/, 2009.

[13] M. Zitser, R. Lippmann, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. *ACM SIGSOFT Softw. Eng. Notes*, 29:97–106, October 2004.