(Do Not Redistribute) Vulnerability corpora and vulnerability representations

Jeff Stuckman
Computer Science Department
University of Maryland

April 2013

1 Vulnerability corpora in empirical vulnerability research

We define a *corpus* of vulnerabilities as a database cataloging and providing detail on security vulnerabilities present in one or more programs. Having a vulnerability corpus is an essential step toward developing and evaluating vulnerability indicators and related metrics, as otherwise the indicators or metrics would be developed without any reference to the empirical phenomenon being tested. A corpus of vulnerabilities provides the dataset that is needed to test candidate metrics for validity as vulnerability indicators. A vulnerability corpus can also be used to estimate the parameters of vulnerability prediction models or support exploratory data analysis to informally discover characteristics of vulnerable code that could yield a metric.

Many past studies of vulnerabilities, such as [Walden et al., 2010, Seixas et al., 2009, Shin and Williams, 2011], compiled collections of existing applications and vulnerabilities to provide data for their research. However, many of these studies collected data from a small number of applications, and none made the dataset available to the wider research community. To perform the analysis described in this proposal, we will need a corpus of vulnerabilities collected from a diverse set of applications. Therefore, one goal of this research is to compile a vulnerability corpus that could support both this research and future vulnerability studies. A secondary goal is to make the corpus publicly available in order to make our experiments repeatable and encourage similar research.

1.1 Actual and synthetic vulnerabilities

We intend to build this corpus by compiling a database of vulnerabilities which were discovered by other developers in operational software products, having been introduced over the natural course of development. We considered other methods of creating a vulnerability database more quickly, but rejected them for the following reasons:

- **Statically discovering vulnerabilities in existing applications** Static vulnerability discovery tools analyze application source code and identify suspected vulnerabilities [Walden and Doyle, 2012]. Using these static analysis tools would speed up the corpus development process by automatically encoding the vulnerabilities in the applications that were added; however, these tools yield large numbers of false positives, most of which are concentrated in particular vulnerability types [Austin and Williams, 2011]. Automatically discovering vulnerabilities is a hard problem due to the need to comprehend the intended behavior of the programs being analyzed, and we expect that such tools would yield a large number of false negatives for the same reasons. Using vulnerabilities discovered by static analysis as statistical support for an metric evaluation would skew the evaluation in accordance with the peculiarities of the static analysis tool, and types of vulnerabilities overrepresented in the tools' results would have an outsize effect in the analysis.

- **Injecting vulnerabilities into existing applications** Some security studies (such as [Fonseca et al., 2007]) created vulnerable applications by deliberately modifying the code of existing applications to introduce vulnerabilities. Although vulnerability injection may be an appropriate technique for obtaining vulnerabilities for some purposes, such vulnerabilities would not be appropriate for validating metrics as vulnerability indicators. Because vulnerability indicators are used to predict which source code locations are more likely to contain vulnerabilities, injected vulnerabilities used to validate these indicators would have to be injected in places where they are typically found in actual applications, necessitating foreknowledge of the relevance of the indicators that the research will ultimately validate.
- **Creating synthetic, vulnerable applications** Some vulnerability research efforts (such as [Fong et al., 2008]) developed custom, deliberately vulnerable applications which were specifically designed to contain all the kinds of vulnerabilities needed for the study. For the same reasons as in the vulnerability injection case, such applications are not appropriate for this research.

Ultimately, for our validated indicators to be useful for predicting vulnerabilities in some population, we need to ensure that the process or mechanism by which vulnerabilities arose in the corpus is similar to how vulnerabilities arise in the population in question. The safest way to ensure this is to build a corpus with actual examples of applications and vulnerabilities from the population in question (being applications developed in a real-life setting and put into production).

1.2 Choosing applications and vulnerabilities for a vulnerability corpus

To build a vulnerability corpus, it is necessary to obtain the source code of numerous applications which contain known vulnerabilities. We have chosen to limit our corpus to vulnerabilities in open-source PHP web applications which have already been discovered and reported to the public. Limiting the scope of the corpus does raise the possibility that the metrics will not be generalizable to closed-source or non-PHP applications. Because the source code for open-source applications is readily available on the internet (even the source code of old versions [Fonseca and Vieira, 2008]), building a corpus consisting entirely of open-source applications is much more feasible. Also, we are not aware of any research that reveals differences between open-source and commercial software in how bugs or vulnerabilities are introduced. Limiting the corpus to PHP web applications is a larger threat to generality, but using a single platform eases the construction of the static analysis tools needed to analyze the exploitability of vulnerabilities (discussed in a later chapter). Vulnerable web applications are considered Top Cyber Security Risks to be a top security risk, and applications written in PHP are perceived to be (and possibly are measurably [Walden et al., 2010]) insecure relative to other languages. We have observed that open-source PHP applications have a heavy presence in publicly available vulnerability databases, making data on these applications easy to obtain and suggesting that even security vulnerability indicators that are limited to PHP would be valuable. Finally, because PHP is an interpreted scripting language, the source code for the applications being analyzed is guaranteed to be available.

Several free, publicly available vulnerability databases are available on the web, such as Injector [inj], the National Vulnerability Database [NVD], the Open Source Vulnerability Database [OSVDB], and the Exploit Database [exp]. At the very least, all of these databases contain lists of applications and descriptions of specific vulnerabilities that have been discovered in them. Some databases also provide downloads of vulnerable versions of applications or downloadable exploit code. Although, in the case of open-source software, these databases may duplicate information already found on the open source projects' own websites, these databases perform the valuable function of aggregating the available information, even though different source of vulnerability information sometimes contradict [Massacci and Nguyen, 2010]. We intend to obtain vulnerabilities for our corpus by sampling vulnerabilities from these databases. (There are numerous considerations on how a valid sample can be built, which are described in a later chapter.)

1.3 Collecting vulnerability details

Although online vulnerability databases compile useful information on vulnerabilities, they cannot serve as vulnerability corpora by themselves. Most vulnerability databases do not include a copy of the vulnerable

application's source code in the database entry, and some only give a vague description of how the vulnerability can be exploited. A corpus must contain the source code of the vulnerable application and enough information to determine what is vulnerable and where the vulnerability can be found in the source code.

Because our corpus will only contain open-source applications, the vulnerable version of each application can typically be downloadable from the project website or elsewhere on the web. Information allowing us to determine the exact nature of the vulnerability can be discovered in several ways:

- If the source code repository for the application is publicly accessible, then it may be possible to locate the code commit that fixed the vulnerability. The vulnerability localization definition could be derived from this, keeping in mind that it may have been possible to fix the vulnerability by modifying multiple code locations. One issue when mining source code repositories for vulnerability localization is that vulnerability patches are sometimes intermingled with non-security-related fixes [DaCosta et al., 2003].
- If exploit source code for the vulnerability is available, the exploit can be executed on a test system, and the issue leading the vulnerability can be discovered by collecting execution traces and reverse-engineering the reason for the undesired behavior. Johnson et al. [2011] developed a tool to facilitate this reverse-engineering process.
- If a patched version of the program was released by the developers, the patched and unpatched source code can be compared to locate the original issue causing the vulnerability. We and others [Fonseca and Vieira, 2008] have observed that security releases sometimes include other, unrelated code changes, so some code inspection is required to find the actual vulnerability. Linux distributions often release security updates which narrowly target vulnerabilities, and such updates even sometimes include explicit patches revealing which lines are affected [DaCosta et al., 2003]; however, only the most popular PHP web applications are commonly packaged for Linux distributions.
- The above methods can be combined – for example, informally examining the differences between two versions of a program may reveal additional clues about a vulnerability that had an intentionally vague database entry, and this information can then be used to construct an exploit to better understand which code was actually affected.

The workflow for building the corpus will include browsing/sampling vulnerability databases for a vulnerability, obtaining and installing the vulnerable application in order to test it, running the application and inspecting its source code to collect localization definition (as described in the next section), and recording the information collected in a machine-readable corpus. At the very least, this corpus will contain the vulnerable code, the vulnerability's localization definition, and any information (links to vulnerability databases, file attachments) that was used when building the record in the corpus so the record could be traced back to its original sources if necessary. The corpus must be kept in a structured, machine-readable format, so automated large-scale analysis of indicators versus vulnerabilities can later be performed.

2 Vulnerability localization definitions

Intuitively, security vulnerabilities are associated with specific features of a program or specific portions of a program's source code. We refer to the practice of mapping a vulnerability to its affected functionality or code as "vulnerability localization". In order to use a corpus to evaluate metrics, this intuitive notion of localization must be formalized and a machine-readable format created so vulnerabilities can be associated with the relevant metric values at the same locations. Past security vulnerability research inevitably involved choosing a vulnerability localization definition representation; however, these representations were typically defined and chosen in an ad-hoc manner. Below, we propose several ways to represent vulnerability localization definitions, including a novel trace-based representation. In this research, we intend to validate security vulnerability indicators with multiple vulnerability definition representations in order to determine if the choice of representation significantly impacts the ability to validate vulnerability indicator metrics using the vulnerabilities in the corpus.

2.1 Line-based vulnerability definitions

Most past security research localized vulnerabilities by selecting one line (or a set of lines) from the program and declaring that those lines collectively encompassed the vulnerability. Fonseca and Vieira [2008] analyzed vulnerability patches and defined vulnerabilities by locating lines of source code that appeared in the security-related portions of the patch. They noted that in some cases, the patch extensively refactored or rearchitected the code to fix the vulnerability, making it difficult to determine where the vulnerability was originally located. A modified version of this method was performed by Seixas et al. [2009], who considered each line changed in a patch to be a separate vulnerability for analysis purposes. Variants of this method include classifying functions [Shin and Williams, 2008] or files [Shin et al., 2011] as vulnerable if any subsequent security-related patches modified code in that function or file.

Although line-based vulnerability definitions are simple and intuitive, they assume that particular lines of code (or functions, or classes) are predominantly responsible for the vulnerabilities. This is true to an extent (especially on the function or class level), but security vulnerabilities can also result from an entire sequence of operations performed improperly. For example, cross-site scripting attacks can be defeated by properly sanitizing a malicious payload, and this sanitization can be accomplished anywhere along the dataflow between the attacker and the victim (which may even span multiple requests). In situations like these, the choice of which line of code to tag as vulnerable is based on a developer's intuition of where the vulnerability should have been fixed, which is somewhat arbitrary but appropriate for some metric use cases (as described below).

2.2 Run-based vulnerability definitions

To explore alternate methods of determining which code in a program is related to a vulnerability, we revisit our notion of the distinction between a "vulnerability" and an "exploit" which was described in the first chapter. Consider that over one run of a program (one PHP web request), some inputs will be considered exploits (which induce the unwanted behavior) while others will not. (In cases where exploits involve multiple PHP web requests, we only consider the individual request which caused the post-request state of the system to assume a property not intended by the developer.) Therefore, instead of defining vulnerabilities by identifying which *lines of code* are vulnerable, we can define vulnerabilities by identifying which *inputs* or *runs* induce the malicious behavior enabled by the vulnerability.

2.2.1 Existing run-based vulnerability definition representations

Brumley et al. [2008] developed a theory and representation to characterize vulnerabilities in terms of the exploits against them. In this paper, a vulnerability signature is defined as an oracle that recognizes inputs which exploit a particular vulnerability. Two properties for vulnerability signatures are introduced: a *soundness* property (all inputs detected as exploits are actually exploits) and a *completeness* property (all exploits are detected as such). Representations and detection mechanisms for vulnerability signatures based on Turing machines, regular expressions, and symbolic constraints are introduced. In addition, they distinguish between *monomorphic execution path (MEP)* and *polymorphic execution path (PEP)* signatures, where polymorphic signatures recognize exploits that follow any program path, and MEP signatures only recognize exploits that use one, particular program path. Typically, only PEP signatures can be complete.

Brumley et al. [2008] also introduce the concept of a *vulnerability point*, which is the point where the application is deemed to have been exploited if a safety property (expressed as a predicate) has been violated. Both vulnerability signature and vulnerability point definitions qualify as run-based vulnerability definitions, because both kinds of definitions can serve as an oracle to identify vulnerable inputs or runs.

While both of these run-based vulnerability definition representations capture the essence of vulnerabilities much more precisely than the line-based definitions do, they cannot directly be used to validate security vulnerability indicators in their current form. To validate indicators, particular locations in the code associated with the vulnerability must be positively associated with the presence of the indicators. While these vulnerability definitions representations precisely describe the vulnerability's functionality with respect to

the program, it is difficult or impossible to transform this description of functionality into source code locations that were coded improperly. We propose an alternative vulnerability definition representation, a trace-based representation, which allows for run-based vulnerability definitions to be associated with source code locations.

2.2.2 Trace-based vulnerability definition representations

We propose a trace-based vulnerability definition representation which consists of an expression that classifies traces of program runs as *potentially exploitative* or *non-exploitative*. Traces are classified by testing them against an expression that matches portions of traces in the same way that regular expressions match portions of strings (including, for example, boolean and repetition operators that allow for statements to be repeated such as in loops). If the expression matches any portion of the trace in question (allowing for entries at the beginning and end of the trace to be omitted, but requiring the matched portion to be one continuous run of entries), then the trace is potentially vulnerable. Traces are classified without regard to the program's input parameters (although, given any assignment of input parameters, a trace can be generated by simply running the program).

This representation is motivated by the fact that many vulnerabilities are caused by a malicious set of inputs being insufficiently checked, sanitized, or reset before they reach a security-critical area of the program. For example, cross-site scripting attacks result from insufficient sanitization between the user's inputs and the HTML page's outputs (with multiple requests, linked by database storage, possibly intervening). Authentication bypass attacks result from the user's privilege not being checked before the user's request to perform a security-critical operation is executed. In the first example, the vulnerability lies along the path from the input code to the output code; in the second, the vulnerability is on the path from the input code to the operation. The trace-based vulnerability definition representation in both of these cases would be constructed to match the code paths in question, while being careful to capture any unexpected paths that a user could follow to reach the end-point of the exploit. As with MEP vulnerability signatures, the trace-based definition must be complete and should be as sound as possible. Trace-based vulnerability definitions must be constructed by a knowledgeable programmer who ensures completeness while attempting to avoid including code irrelevant to the vulnerability (which would reduce the accuracy of the indicator validation) in the trace.

This process of constructing trace-based vulnerability definitions requires human judgment, but the definitions should have the following properties:

- All traces produced by runs which exploit the vulnerability must match the vulnerability definition. When testing a vulnerability definition, a variety of exploits (including uncommon exploits that reach the vulnerability through an uncommon code path) should be tested against the definition to ensure that this holds in all relevant cases.

- Traces produced by runs which do not exploit the vulnerability should match the vulnerability definition as seldom as possible. This property ensures that the vulnerability definition is actually highlighting the vulnerable code, rather than simply highlighting common code that pertains to both exploit and non-exploit cases. In some cases, it may not be possible to distinguish exploits from non-exploits by examining traces (such as when a payload is improperly sanitized, causing the exploit code path to match the non-exploit code path, with the two being distinguished only by the maliciousness of the payload.) In these cases, the vulnerability definition will match many non-exploit runs, although the definition should at least highlight the vulnerable code path.

- The vulnerability definition should match a large number of trace entries in exploit traces, rather than a small number. Because a trace-based vulnerability definition can match a portion of a trace and declare a trace to be an exploit based on that portion, it is possible to write very short definitions that match a single line of code, rather than the entire dataflow that was relevant to the exploit. Longer definitions that match entire dataflows are more useful, as they highlight all the parts of the program where the code could conceivably be modified to mitigate the vulnerability.

- The vulnerability definition should not explicitly match portions of traces through irrelevant parts of the code (such as library functions not related to the vulnerability). Instead, wildcards should be employed to allow traces to pass through such code. This ensures that the vulnerability definitions don't imply that the vulnerability could be mitigated by modifying clearly irrelevant code, even if exploit traces pass through such code.

2.2.3 Compiling trace-based vulnerability definitions

In order to compare the effectiveness of line-based and trace-based vulnerability localization definition representations, we intend to encode both a line-based and a trace-based definition for each vulnerability into the corpus. When validating indicators of security vulnerabilities, we then intend to test if trace-based vulnerability localization definitions can validate more indicators by providing increased statistical power (e.g. by reducing variance by expressing the vulnerability's location more accurately.)

A completely manual process for creating trace-based definitions for each vulnerability would be time-consuming, as the traces would potentially be long and complex, so we intend to develop semi-automated tools that assist in this task. We have already written code that observes exploit and non-exploit runs against a vulnerability in the corpus and records the traces to disk. We intend to use this code to develop a tool that recommends trace fragments that could be used to build a trace-based definition. This tool would ideally provide a GUI allowing the user to browse the program's source code and visualize an overlaid control flow graph, allowing the user to highlight the relevant data flows (expressed as traces through their control flows), replacing irrelevant portions such as library calls and irrelevant branches with appropriate wildcards along the way. The tool would then test the resulting definition by observing which exploit and non-exploit examples matched it. In cases where exploit and non-exploit traces differed significantly (such as when an authentication check is bypassed), such a tool could quickly locate trace segments relevant to the vulnerability; however, when exploit and non-exploit traces are identical (such as in SQL injection attacks), the tool could still narrow down the number of trace segments that the user must consider as part of the definition.

The practice of characterizing defects in programs by observing execution differences between multiple runs has been used by other researchers in the past. Johnson et al. [2011] built a tool that described security vulnerabilities by analyzing the differences between exploit and non-exploit traces and generating a graph showing the chain of events that caused the differences in observed program behavior when exploited. The authors note that spurious differences between executions (such as pointers differing between two runs) can cause insignificant differences between the traces, a problem we will also have to address when comparing traces. Jones and Harrold [2005] localized faults in programs by locating statements and branches that predominately appeared in failing (or passing) test cases. Such a technique could be adopted by a vulnerability definition compilation tool that attempts to automatically locate the vulnerable region of code, although the user would still have to select the vulnerability's relevant trace segments, and a large number of exploit and non-exploit examples would be required.

2.2.4 Vulnerability definition representations and the purpose of security metrics

Although trace-based vulnerability definition representations paint a more complete picture of security vulnerabilities than the line-based definitions, using line-based definitions may result in security vulnerability indicators that are more suitable for some some purposes. Because line-based definitions are typically based on where code was changed to fix a vulnerability, these definitions highlight the "obvious" areas in a program where security vulnerabilities can be fixed. These areas of the program may also be the areas where code inspection is most likely to productively find vulnerabilities for the same reasons (although no past research has examined this hypothesis, because past research has not explicitly explored security vulnerability representation.)

Hence, the ideal style of vulnerability definition representation (line-based or trace-based) used to evaluate security vulnerability indicator metrics may ultimately depend on the way that the indicators will be used. If the indicators will be used to select areas of the code for inspection, line-based representations would

be better, while if the indicators were used to gauge the security impact of a new feature or estimate the comparative security risk of two programs, the trace-based representations may give better results.

References

Exploit db. http://www.exploit-db.com/.

inj3ct0r. http://1337day.com/.

A. Austin and L. Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 97 –106, sept. 2011. doi: 10.1109/ESEM.2011.18.

D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Theory and techniques for automatic generation of vulnerability-based signatures. *Dependable and Secure Computing, IEEE Transactions on*, 5(4):224 –241, oct.-dec. 2008. ISSN 1545-5971. doi: 10.1109/TDSC.2008.55.

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. ISBN 0-7695-1905-9. URL `http://dl.acm.org/citation.cfm?id=942800.943588`.

E. Fong, R. Gaucher, V. Okun, and P. Black. Building a test suite for web application scanners. In *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, page 478, jan. 2008. doi: 10.1109/HICSS.2008.79.

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. doi: 10.1109/DSN.2008.4630094.

J. Fonseca, M. Vieira, and H. Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 365 –372, dec. 2007. doi: 10.1109/PRDC.2007.55.

N. Johnson, J. Caballero, K. Chen, S. McCamant, P. Poosankam, D. Reynaud, and D. Song. Differential slicing: Identifying causal execution differences for security applications. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 347 –362, may 2011. doi: 10.1109/SP.2011.41.

J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM. ISBN 1-58113-993-4. doi: 10.1145/1101908.1101949. URL `http://doi.acm.org/10.1145/1101908.1101949`.

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. ISBN 978-1-4503-0340-8. doi: 10.1145/1853919.1853925. URL `http://doi.acm.org/10.1145/1853919.1853925`.

NVD. National vulnerability database. http://nvd.nist.gov/.

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

N. Seixas, J. Fonseca, M. Vieira, and H. Madeira. Looking at web security vulnerabilities from the programming language perspective: A field study. In *Proceedings of the 2009 20th International Symposium on Software Reliability Engineering*, ISSRE '09, pages 129–135, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3878-5. doi: http://dx.doi.org/10.1109/ISSRE.2009.30. URL `http://dx.doi.org/10.1109/ISSRE.2009.30`.

Y. Shin and L. Williams. Is complexity really the enemy of software security? In *Proceedings of the 4th ACM workshop on Quality of protection*, QoP '08, pages 47–50, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-321-1. doi: 10.1145/1456362.1456372. URL `http://doi.acm.org/10.1145/1456362.1456372`.

Y. Shin and L. Williams. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, pages 1–7. ACM, 2011.

Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *Software Engineering, IEEE Transactions on*, 37 (6):772–787, 2011.

Top Cyber Security Risks. Top cyber security risks. (SANS Institute) http://www.sans.org/top-cyber-security-risks/, 2009.

J. Walden and M. Doyle. Savi: Static-analysis vulnerability indicator. *Security Privacy, IEEE*, 10(3):32 –39, may-june 2012. ISSN 1540-7993. doi: 10.1109/MSP.2012.1.

J. Walden, M. Doyle, R. Lenhof, and J. Murray. Java vs. php: Security implications of language choice for web applications. In *International Symposium on Engineering Secure Software and Systems (ESSoS)(February 2010)*, 2010.