Baseline: Metrics for setting a baseline for web vulnerability scanners

Huning Dai, Michael Glass, and Gail Kaiser

Department of Computer Science,
Columbia University,
New York, NY 10027 USA
{dai,mgg2102,kaiser}@cs.columbia.com
http://www.cs.columbia.edu

Abstract. As web scanners are becoming more popular because they are faster and cheaper than security consultants, the trend of relying on these scanners also brings a great hazard: users can choose a weak or outdated scanner and trust incomplete results. Therefore, benchmarks are created to both evaluate and compare the scanners. Unfortunately, most existing benchmarks suffer from various drawbacks, often by testing against inappropriate criteria that does not reflect the user's needs. To deal with this problem, we present an approach called Baseline that coaches the user in picking the minimal set of weaknesses (i.e., a *baseline*) that a qualified scanner should be able to detect and also helps the user evaluate the effectiveness and efficiency of the scanner in detecting those chosen weaknesses. Baseline's goal is not to serve as a generic ranking system for web vulnerability scanners, but instead to help users choose the most appropriate scanner for their specific needs.

Keywords: web vulnerability scanner; benchmark; weakness; vulnerability; finite state machine; baseline

1 Introduction

As the Internet has grown in popularity, security testing is becoming a crucial part of the life cycle for software systems, especially for web service applications and websites. Web vulnerability scanners, both commercial and open-source (e.g., Rational AppScan[1], WebInspect[2], Nikto[3], etc.), were developed to automatically scan web applications to detect potential vulnerabilities. As these scanners are becoming more popular because they are faster and cheaper than security consultants, the trend of relying on these scanners also brings a great hazard: users can choose a weak or outdated scanner and trust (sometimes

[1] http://www-01.ibm.com/software/awdtools/appscan/
[2] http://www.hp.com/webinspect
[3] http://cirt.net/nikto2

highly) incomplete results. Fortunately, there have been assessment benchmarks that make comparisons between these scanners in order to determine which ones are better. Most of these benchmarks use the scanners to scan a manually crafted website with a number of known vulnerabilities, and rate the scanners based on the percentage of successful detection. These benchmarks are only capable of judging which scanner is better in the matter of how well the scanners can detect the fixed set of vulnerabilities the benchmarks picked with static selection criteria. They suffer from drawbacks by neglecting the critical questions: Does the benchmark properly reflect the user's security requirements; does it reflect the user's actual deployment environment? In helping the users choose the right scanners, answering these questions is as crucial as evaluating the effectiveness and efficiency of the scanners.

In this paper, we propose an approach called Baseline that addresses all of these problems: We implement a ranking system for dynamically generating the most suitable selection of weaknesses[4] based on the user's needs, which serves as the baseline that a qualified scanner should reach/detect. Then we pair the ranking system with a testing framework for generating test suites according to the selection of weaknesses. This framework maps a weakness into an FSM (Finite State Machine) with multiple end states that represent different types/mutations of exploitations of the weakness and each transition from state to state determined by scanner behavior, the framework then combines the FSMs of the selected weaknesses into a mimicked vulnerable website. When a scanner scans the "vulnerable" website, the transitions between the states are recorded and thus we are able to evaluate the scanner by looking at which end states were visited (effectiveness), in how much time, and over how many transitions (efficiency).

The rest of this paper is organized as follows. Section 2 proposes the Baseline approach and divides it into a weakness ranking system and a testing framework. Section 3 introduces the dynamic ranking system for weaknesses and provides some results of the ranking system. Section 4 looks at the testing framework, explains the idea of mapping a weakness to a finite state machine and shows the results of our case studies. Related work is then discussed in Section 5. We address the system's limitations in Section 6, and finally summarize our findings in Section 7.

2 Baseline Approach

Instead of trying only to define a benchmark to compare scanners, we decided to answer a more important question: whether the claims a scanner's vendor makes in terms of its coverage are valid in the first place. We propose an approach that coaches the users in selecting a minimal set of weaknesses (i.e., a baseline) that a qualified scanner should be able to detect. To achieve this, we maintain a dynamic ranking of all weaknesses from the Common Weaknesses Enumeration

[4] "weakness" as shorthand for "category of vulnerabilities", in the style of the Common Weakness Enumeration

[8] (CWE) paired with vulnerability reports from the National Vulnerability Database [6] (NVD). We rank the weaknesses based on their vulnerabilities' average severity (using their CVSS[5] scores from NVD) as well as their frequency within a given date range specified by the user. In addition, since different users might have different priorities, we have developed a formula that allows the user to specify how they weight the relative importance of frequency versus severity. After the user defines the baseline using our system, the testing framework maps each weakness to a finite state machine (FSM) and combines these FSMs to be the back end for a test website for the scanner. Each state within a FSM represents a known type/mutation of a possible exploitation of the weakness. Transitions between the states are recorded as the attempts the scanner made in order to detect the vulnerability. These transition records serve as the criteria for evaluating both the effectiveness (how many states were visited) and the efficiency (in how many transitions/in what time did it take the scanner to visit those states) of the scanner. The result provided by Baseline is then used as guidance in helping the user choose the most appropriate scanner for their specific requirements.

The baseline approach is composed of two parts: A weakness ranking system and a test framework. Although we introduce these two parts separately later in this paper, we would like to remind the readers that they are parts of the same whole and the ranking system parameterizes the test framework.

3 Dynamic ranking system

3.1 Approach

The major question we want to solve is: which weaknesses should represent the minimal set of vulnerabilities that a qualified web vulnerability scanner will find. One idea is to collect all the claims made by commercial and open-source scanners to find the most commonly detected vulnerabilities and categorize them into weaknesses. However, a baseline generated by this metric might not be fair, because many products tend to target specific weaknesses. Also when we try to collect the claims from the Top 10 Web Vulnerability Scanners [9], 9 out of 10 do not have a specified list of vulnerabilities that they are able to detect. This fact shows that using the claims will impede the legitimacy of the baseline greatly.

In order to deal with these problems, we have created a dynamic ranking system of all known weaknesses. To ensure its legitimacy, our system categorizes vulnerabilities into weaknesses using the standardized Common Weakness Enumeration (CWE) and pairs it with statistical data pulled from the National Vulnerability Database(NVD). Our system automatically updates its ranking criteria using the newest data from CWE and NVD: CWE updates their data monthly and NVD adds more than 10 entries daily. A live ranking system of the weaknesses has a lot of merits. First of all, it provides the users a more explicit view of the impact of different weaknesses (in terms of their severity and

[5] The Common Vulnerability Scoring System (CVSS) - http://www.first.org/cvss/

frequency). Second, it helps to find recent trends in vulnerability occurrence, and can thusly be used to help guide web security development. Furthermore, it serves as the most reasonable criteria in defining a baseline for vulnerability scanners by using real world severity and frequency instead of what's conceptually popular or newsworthy.

3.2 Architecture

We cache the NVD and CWE database to efficiently generate rankings upon request. The database is easily updated whenever new data is available on CWE or NVD. Our ranking system provides the option to omit system weaknesses (e.g., buffer overflows) because most web vulnerability scanners focus on weaknesses that are web application related.

We maintain a table in the database for all weaknesses from CWE; the schema is shown in Table 1. Because one weakness might belong to another weakness, we keep its ParentID in the database. For NVD, we keep a table of all reported instances of vulnerabilities together with the IDs of the weaknesses they belong to, Table 2.

Table 1. CWE table

Column Name	Description
ID	Corresponding to the weakness ID from CWE
Name	Weakness name
Type	System related or Web application related
Count	Number of vulnerabilities found in the NVD table of this weakness, including vulnerabilities of descendant weaknesses
Score	Average CVSS score of the vulnerabilities in the NVD table of this weakness, including vulnerabilities of descendant weaknesses
ParentID	Corresponding to the ID of its parent weakness

Table 2. NVD table

Column Name	Description
ID	Corresponding to record ID from NVD
WeaknessID	Corresponding to the ID of the weakness it belongs to from CWE
Score	CVSS score of this instance of vulnerability
Published	Published timestamp
Modified	Modified timestamp

After the database is populated, we can easily generate a frequency ranking of weaknesses based on the number of occurrences of related vulnerabilities from

NVD and a severity ranking using the average CVSS score of these vulnerabilities. With these two rankings, we designed a formula that allows the users weight frequency and severity to calculate which weaknesses should be included in their baseline.

Assume that we want to take a set of N representative weaknesses as the baseline, and T is the total number of weaknesses in the CWE table. We define a *frequency_score* for each weakness in CWE. The *frequency_score* for the weakness can be calculated with:

$$frequency\_score = frequency\_rank * 10/T \tag{1}$$

In which the *frequency_rank* of the weakness with the least number of occurrences is 1 and the *frequency_rank* of the most common weakness is T. We multiply *frequency_rank* by a constant, 10, because the CVSS score is 10-based and we want to map the *frequency_score* to that.

Using the *frequency_score* together with the *severity_score* (average CVSS score from NVD of all vulnerabilities associated with the weakness as well as all vulnerabilities associated with its descendant weakness) and the frequency versus severity weights decided by the user, we can derive the *final_score* of a weakness in CWE with this formula:

$$\begin{aligned} final\_score = {} & frequency\_weight * frequency\_score \\ & + severity\_weight * severity\_score \end{aligned} \tag{2}$$

Note *frequency_weight*+ *severity_weight* = 1.

Finally, our system ranks each weakness ordered by the *final_score* and takes the top N weaknesses as the set representing the baseline. It is worth pointing out that our sysyem also allow the users to omit certain weaknesses or reorder their priority in the baseline.

3.3 Case study

We tested the ranking system by requesting the top 10 web application related weaknesses from Jan 1st 1996 to May 7th 2010 with different weights on *frequency_score* and *severity_score*. Applying the formula with *frequency_weight* = 0.5 and *severity_weight* = 0.5 gives us Table 3. It is worth mentioning that the *frequency_weight* and *severity_weight* are adjustable by the user. Imagine that a government server might treat severity more seriously than frequency of attack in order to defend against fatal attacks while a high traffic portal website might care more about preventing the most common attacks, so they would use different weights to generate different baselines. Table 6 in the appendix shows the top 10 most common web weaknesses by applying the formula with *frequency_weight* = 1 and *severity_weight* = 0 and Table 7 in the appendix is the top 10 most dangerous web weaknesses generated with *frequency_weight* = 0 and *severity_weight* = 1.

Table 3. The Top 10 web weaknesses from 1996-2010 by Baseline (order by *final_score* with equal weights)

Weakness Name	Avg.Score	Count	Final score
Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection')	7.37405	2528	8.687025
Failure to Control Generation of Code	7.69732	1083	7.79616
Permissions, Privileges, and Access Controls	6.33941	1251	7.380205
Improper Input Validation	6.51184	1030	6.93992
Failure to Preserve Web Page Structure ('Cross-site Scripting')	4.24536	2218	6.85918
Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	6.44464	943	6.64332
Resource Management Errors	6.7512	707	6.5336
Improper Authentication	7.10087	462	6.181935
Numeric Errors	7.50741	459	6.121705
Information Exposure	4.90841	523	5.99082

4 Weakness FSM

4.1 Approach

After a baseline is generated by our dynamic ranking system with the user's preference, the user might search for these weaknesses or related vulnerabilities in the scanner's vendor's claim (if there is one). If indeed claimed by the vendor, it at least means that the user has found a product targeting appropriate weaknesses. However, the question, "how well does the scanner actually do what it claims?" remains unsolved. To answer this question, we developed a framework that maps each weakness to a finite state machine (FSM) with multiple end states as different successful exploitations. The FSMs for different weaknesses can be dynamically combined to test multiple weaknesses simultaneously. The FSM testing framework produces a website with each page representing a state and the scanner behavior determines the transitions from state to state. Our framework saves a scanner's transition history in the database. With these transition records, we are able to evaluate the scanner by looking at which end states were visited (effectiveness), in how much time, and over how many transitions (efficiency).

4.2 Architecture

We used PHP's Object Oriented features to build a library with multiple classes for helping the programmers/testers create finite state machines for weaknesses. Figure 1 shows the UML diagram of all the classes we created for mapping a weakness into a FSM. Each Test object represents a weakness and it contains several Transition objects and State objects. State objects represent HTML pages

that contain input fields for the scanner, and the library has been designed with various features to help generate states easily without touching HTML - for instance, a state can have forms and links added programmatically. To verify whether a transition is valid, we created an interface called iTestable with a function run_test(). Currently, we only implemented the RegExTester class that verifies the transitions by checking the regular expression of three types of user inputs - GET, POST parameters and URLs. However, other validation classes that extend the iTestable class can be created to determine the transitions using different methods. For example, better SQL injection testing can be achieved using real SQL parsing libraries instead of testing inputs against common regular expression patterns of SQL statements.

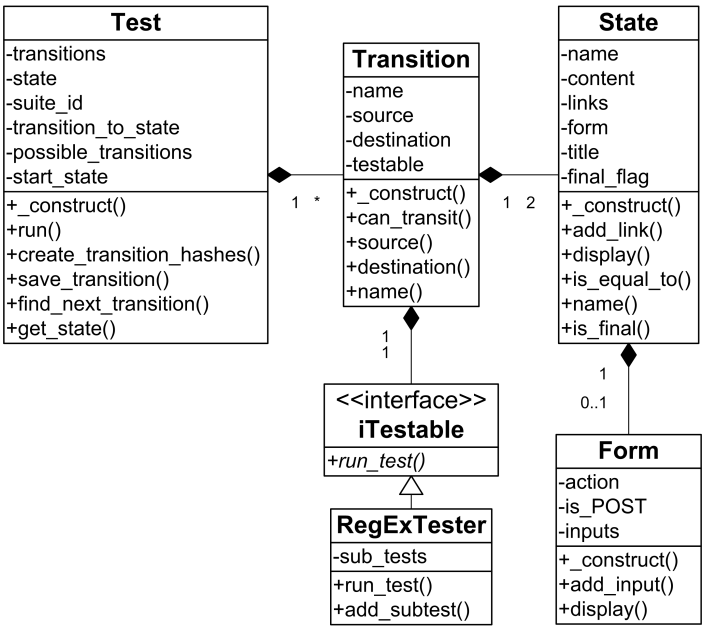

Fig. 1. UML diagram for FSM classes

4.3 Case study

SQL injection. Here we show an example finite state machine we created using our library that represents a very general SQL injection weakness [17]. The FSM is illustrated in Figure 2. There are a total of three end states, namely blind, simple and fancy. They stand for three levels/types of possible exploitations of a SQL injection weakness as classified in the SQL Injection Cheat Sheet [10]. Blind SQL injection is exploited by injecting logical statements into the legitimate SQL statement; simple SQL injection is to bypass login screen with comment symbols

or conditional statements; fancy SQL injection includes using stacking/union statements, hex bypassing and string concatenation. We used the cheat sheet to define the regular expression tests that make up the transitions between each state.

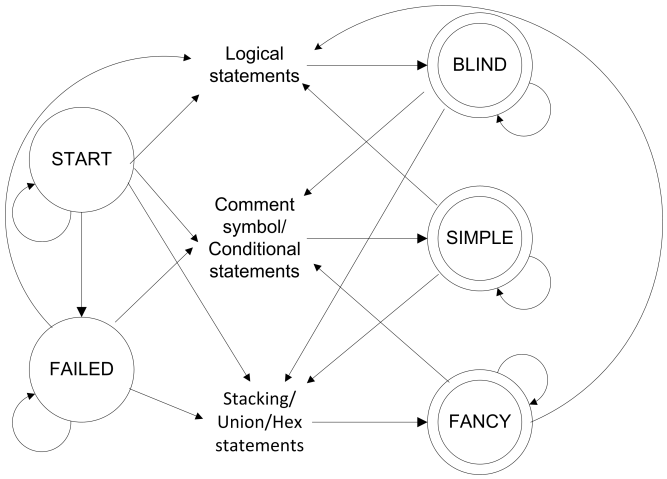

Fig. 2. Finite State Machine for SQL injection weakness

After creating the FSM for SQL injection weakness, we used Scrawlr [7], a SQL injection vulnerability scanner by HP Lab, to scan the test pages. HP Lab claims that Scrawlr is capable of identifying SQL Injection vulnerabilities in URL parameters. We did not expect Scrawlr to report that it finds any vulnerabilities since our test pages are not designed to provide realistic feedback and they are not really vulnerable. However, we can see the attempts it made while trying to detect the vulnerability by looking at the transition records. Successful visits of the end states imply that Scrawlr should be able to detect the vulnerability when scanning a real vulnerable website. Table 6 shows the records of transitions made by Scrawlr after scanning our test page.

The transition records in the database indicate that Scrawlr first submitted the form with regular inputs, and then it appended "'+OR" and "'+AND+5%3-d5+OR+'s'%3d'0" to the input fields, which would trigger an exploitation of a simple SQL injection vulnerability in a vulnerable website. While we didn't expect Scrawlr to report that it finds any vulnerabilities, we did see attack-like behavior. However, Scrawlr only managed to visit one out of three possible end states, which made us suspicious of Scrawlr's ability to actually detect most types of SQL injection vulnerabilities. We verified this suspicion by testing Scrawlr against two websites explicitly crafted with a number of known SQL injection vulnerabilities: Free Bank Online [5] by HP and Altoro Mutual [3] by Watchfire. Scrawlr announced that both websites were vulnerability-free after the scan.

Table 4. Part of the transitions Scrawlr made scanning the test page

Transition name	Source	Destination	Input (GET from Form)	Timestamp
DO NOTHING	START	FAILED	username=admin password=admin	2010-04-26 17:45:25
DO NOTHING AGAIN	FAILED	FAILED	username=admin password=admin	2010-04-26 17:45:25
FAILED TO SIMPLE	FAILED	SIMPLE	username=admin'+OR password=admin	2010-04-26 17:45:27
STAY SIMPLE	SIMPLE	SIMPLE	username=admin'+AND +5%3d5+OR+'s'%3d'0 password=admin	2010-04-26 17:45:27

XSS. XSS (Cross-site Scripting) has become one of the most common and dangerous weaknesses in the last decade [9] [1]. An exploited XSS vulnerability can be used by attackers to bypass access controls. According to [19] and XSS Cheat Sheet [11], there are three major types of XSS attacks: HTML Quote Encapsulation (HQE), which is accomplished by injecting HTML tags with metadata; URL String Evasion (USE), which is carried out by injecting URL hyperlinks; and traditional Cross-Site Scripting (XSS), which is fulfilled by injecting scripts with tricks to get around the filters [18]. Thus the FSM for XSS we created has three end states as shown in Figure 3 with all possible transitions.

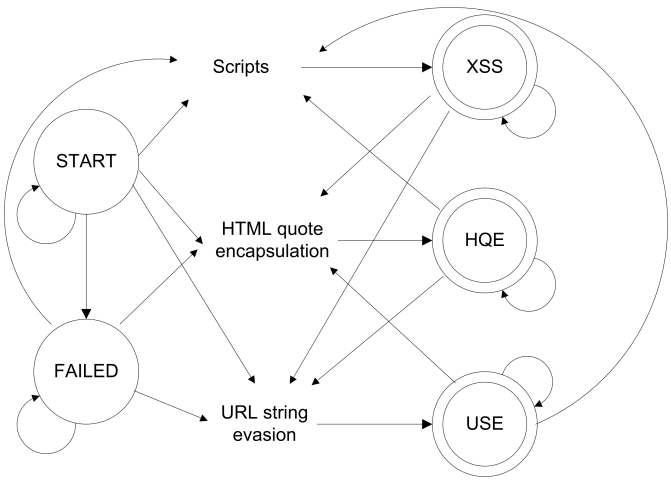

Fig. 3. Finite State Machine for XSS weakness

After we created the FSM for XSS weakness, we used it to test the free edition of Acunetix Web Vulnerability Scanner [2], which is claimed to"identify Cross Site Scripting (XSS) Vulnerabilities". After the scan, the records reveal

that Acunetix Web Vulnerability Scanner visited the XSS and HQE end states multiple times. Part of the record is shown in Table 5.

Table 5. Part of the transitions Acunetix made scanning the test page

Transition name	Source	Destination	Input (GET from Form)	Timestamp
FAILED TO XSS	FAILED	XSS	username=1<script> alert(41206)</script> password=g00dPa$$w0rD	2010-04-27 14:27:36
FAILED TO HQE	FAILED	HQE	username=1<iframe/+ /onload=alert(41386)> </iframe> password=g00dPa$$w0rD	2010-04-27 14:27:38

Acunetix Web Vulnerability Scanner was able to visit two out of three end states and therefore should be able to detect most of the XSS vulnerabilities when scanning a vulnerable website. We verified this by using Acunetix to scan Free Bank Online and Altoro Mutual; Acunetix detected all vulnerable pages as documented. However, since Acunetix failed to visit the USE state, it would not be able to find the pages that are only vulnerable to URL string evasion attacks. An example would be a web page that allows a malicious user to inject example.com w here regular URLs are disallowed or filtered.

Due to space limitations, we only show two preliminary FSMs here. Eventually, we should have sophisticated FSMs for all known weaknesses and developers/testers can easily create new FSMs with the library we built. By having the scanner scan the FSMs of the weaknesses within the baseline, one can evaluate the scanners' effectiveness by looking at how many end states were visited and the efficiency by checking how much time was spent.

5 Related work

Many companies use web vulnerability scanners to assess the security of their website, because scanners are faster and cheaper than security consultants. However, a user might pick a weak or outdated scanner and trust incomplete results. Therefore, researchers have created many benchmarks to both evaluate and compare scanners. A great percentage of the benchmarks (e.g. [20] [12] [21]) use manually crafted websites with known vulnerabilities as criteria for judging the scanners and present a ranking based on the percentage of vulnerabilities successfully detected. However, the results certainly suffer from bias as most benchmarks are created by scanner developers. This suspicion of bias seems to be confirmed by other vendors' responses to the results of benchmarks; most responses used or expressed the word "unfair" [14] [25]. In addition, using crafted websites provides no guarantee of the coverage of the weakness space.

Some benchmarks, such as [15] [13], focus on one or several weaknesses that they consider the most important, and compare the scanners based on their performance in detecting the vulnerabilities of these "important" weaknesses. Some of them [24] also focus on certain web technologies the websites use, such as PHP and Ajax. These benchmarks often use the most common or most dangerous weaknesses without considering the fact that different user might have different view of which weaknesses are of the most importance, e.g. a portal website with heavy traffic might worry more about the most common weaknesses while a government website might care more about the dangerous ones.

Our Baseline approach includes a dynamic ranking system that coaches the users to decide the minimal set of weaknesses that a qualified scanner should be able to detect. The always up-to-date weakness ranking system together with the weight-based formula provides flexibility in guiding the users in choosing their desirable scanner. In addition, Baseline doesn't address the website technology. The idea is that vulnerabilities are triggered by improper and incomplete input validation (via URL, POST/GET, etc.), and Baseline is designed to report what transitions the scanners followed (what inputs the scanners provide) in order to detect the vulnerabilities regardless of the technology being used.

A third type of benchmark scores the scanners in several categories and evaluates each independently. For instance, [23] divides a scanner into three components: getting all the pages, finding all the input vectors and providing invalid, unexpected, or random data to all the input vectors. It, surprisingly, discovered that a large portion of the scanners cannot even reach the vulnerable pages via link traversal, and thusly, cannot even start to exploit vulnerabilities. However, most scanner vendors claim the benchmark's results are invalid since a tester needs to provide credentials to help the scanner crawl all pages [16].

Our approach lets the user decide what is important, instead of a scanner developer, while still making sure what the user decides is valid by leveraging CWE and NVD. Furthermore, the open framework of Baseline allows users that are interested specifically in testing a scanner's crawling ability, for example, to easily develop FSMs that map to that test criteria. Finally, by monitoring a scanner's behavior while scanning the webpage our Baseline system created, we can evaluate both the scanner's effectiveness in finding vulnerabilities as well as its performance in getting to the end results.

6 Limitations & future work

A major limitation resides in the current implementation of validating the transitions between the states. Our system now relies merely on searching for matches of specific attack patterns in the input fields. However, some scanners might take a different approach to detecting vulnerabilities other than trying to actively exploit them, such as using static analysis of the source code [22]. Our system will not be suitable to evaluate those scanners yet. We are now investigating different ways of identifying the scanners' attempts to detect potential vulnerabilities and create new extended classes of the iTestable class. By actually involving source

from webpages in our iTestable classes, we will not only get more realistic test results, but will also allow systems that scan source code for vulnerabilities to still function and identify vulnerabilities. For instance, for an XSS vulnerability scanner, we could implement an iTestable class that leverages a web framework, like Drupal's form parsing code [4]. A scanner that searches for vulnerabilities in source code would have access to the source and individual attacks that it selected would both a) be appropriately vulnerable in the original code and b) could be caught/identified in the iTestable wrapper.

Another limitation of our implementation is that we haven't built all the FSMs for all known weaknesses. We have implemented the FSMs for generic SQL injection, XSS and path traversal, respectively. Nevertheless, with the weakness FSM library we built anyone can create FSMs for specific weaknesses with little instruction. We will also continue working on writing more robust FSMs.

Finally, one limitation out of our hands is that the mapping of National Vulnerability Database (NVD) vulnerabilities to Common Weakness Enumeration (CWE) categories is highly limited - the NVD uses only 19 of the 810 weaknesses in CWE to categorize its vulnerabilities. This is useful, but more specific categorization would be more compelling. Effective selection of weaknesses would be helped significantly with better NVD categorization.

7 Conclusion

In this paper, we explored an approach for setting a baseline for qualified web vulnerability scanners in the domain of security testing, developed a dynamic ranking system of all known weaknesses and a framework that maps weaknesses into finite state machines so as to mimic vulnerable websites. Most existing benchmarks for web vulnerability scanners suffer from bias by using arbitrary static criteria that do not consider the user's needs. Our proposed approach, Baseline, deals with these problems by helping the user decide the minimal set of weaknesses that a qualified scanner should be able to detect and evaluating the effectiveness and efficiency of the scanner in detecting the vulnerabilities of the chosen weaknesses. Baseline doesn't serve as criteria in comparing between the scanners but serves as guidance for the users to choose the suitable ones for their needs. We are currently investigating different ways of identifying successful scanner attacks, as well as improving our system by creating more realistic finite state machines for more weaknesses. We believe that our work can both help users choose an appropriate scanner for their websites and help developers build better scanners.

Acknowledgements

The authors are members of the Programming Systems Lab, funded in part by NSF CNS-0905246, CNS-0717544, CNS-0627473 and CNS-0426623, and NIH 1 U54 CA121852-01A1.

References

1. 2010 CWE top 25 most dangerous software errors, http://cwe.mitre.org/top25/
2. Acunetix web vulnerability scanner (Free Edition) , http://www.acunetix.com/cross-site-scripting/scanner.htm
3. Altoro Mutual, http://demo.testfire.net/
4. Drupal, http://drupal.org/
5. Free Bank Online, http://zero.webappsecurity.com/
6. National Vulnerability Database, http://nvd.nist.gov/
7. Scralwr by HP, https://h30406.www3.hp.com/campaigns/2008/wwcampaign/1-57C4K/index.php
8. Common Weakness Enumeration, http://cwe.mitre.org/
9. OWASP top 10 for 2010, http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
10. SQL injection cheat sheet, http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
11. XSS cheat sheet, http://ha.ckers.org/xss.html
12. Web vulnerability scanners evaluation (Jan 2009), http://anantasec.blogspot.com/
13. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.: State of the art: Automated black-box web application vulnerability testing. In: IEEE Symposium on Security and Privacy (2010)
14. Ewe, L.: Web vulnerability scanner comparison, continued. The Cenzic Blog (April 2010), http://blog.cenzic.com/public/item/253998
15. Fonseca, J., Vieira, M., Madeira, H.: Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks. Pacific Rim International Symposium on Dependable Computing, IEEE 0, 365–372 (2007)
16. Forristal, J.: Analysis of Larry Suto's Oct/2007 web scanner review (November 2007)
17. Halfond, W.G., Viegas, J., Orso, A.: A classification of SQL-injection attacks and countermeasures. In: Proceedings of the IEEE International Symposium on Secure Software Engineering. Arlington, VA, USA (March 2006)
18. Nava, E.V., Lindsay, D.: Our favorite XSS filters and how to attack them. BlackHat USA (Aug 2009)
19. Spett, K.: Cross-site scripting. SPI Labs (2005)
20. Suto, L.: Analyzing the effectiveness and coverage of web application security scanners (October 2007)
21. Suto, L.: Analyzing the accuracy and time costs of web application security scanners (February 2010)
22. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. pp. 32–41. ACM, New York, NY, USA (2007)
23. Wiegenstein, A., Weidemann, F., Schumacher, D.M., Schinzel, S.: Web application vulnerability scanners - A benchmark (October 2006)
24. Wiens, J.: Rolling review: Web app scanners still have trouble with Ajax (2007), http://www.informationweek.com/news/security/showArticle.jhtml?articleID=202201216
25. Wood, M.: On web application scanner comparisons. The HP Security Laboratory Blog (Feb 2010), http://h30507.www3.hp.com/t5/The-HP-Security-Laboratory-Blog/On-Web-Application-Scanner-Comparisons/ba-p/33138

Appendices: Results of the weakness ranking system

Table 6. The Top 10 most common web weaknesses from 1996-2010 by Baseline (order merely by *frequency_score*)

Weakness Name	Count	Avg.Score
Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection')	2528	7.37405
Failure to Preserve Web Page Structure ('Cross-site Scripting')	2218	4.24536
Permissions, Privileges, and Access Controls	1251	6.33941
Failure to Control Generation of Code	1083	7.69732
Improper Input Validation	1030	6.51184
Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	943	6.44464
Resource Management Errors	707	6.7512
Information Exposure	523	4.90841
Improper Authentication	462	7.10087
Numeric Errors	459	7.50741

Table 7. The Top 10 most dangerous web weaknesses from 1996-2010 by Baseline (order merely by *severity_score*)

Weakness Name	Count	Avg.Score
Improper Sanitization of Special Elements used in an OS Command ('OS Command Injection')	8.26562	32
Failure to Control Generation of Code	7.69732	1083
Numeric Errors	7.50741	459
Uncontrolled Format String	7.4093	86
Improper Sanitization of Special Elements used in an SQL Command ('SQL Injection')	7.37405	2528
Improper Authentication	7.10087	462
Resource Management Errors	6.7512	707
Improper Input Validation	6.51184	1030
Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	6.44464	943
Permissions, Privileges, and Access Controls	6.33941	1251