Reviewing Open Source Web Application Vulnerability Scanners and Testing Platforms for Comprehensiveness in Detecting SQL Injection and Cross Site Scripting Vulnerabilities

Michael O'Connell

Abstract

Comprehensiveness as a metric for reviewing web vulnerability scanners has been overlooked by the current literature. Achieving comprehensiveness in protection against novel attack vectors is essential for ensuring a high level of defense. Two web vulnerability scanners, *Zed Attack Proxy* (ZAP) and *Vega* are reviewed on how comprehensive they detect SQL injection (SQLi) and Cross-Site Scripting attacks (XSS). ZAP achieved uneven SQLi comprehensiveness depending on the dialectical SQL used and Vega did not achieve comprehensiveness. Neither achieved XSS comprehensiveness.

Keywords

- **Web Application Vulnerability Scanner:** Automated tool that scans web applications and looks for various security vulnerabilities.
- **Web Application Vulnerability Benchmarks:** Test that compares the features, coverage, vulnerability detection rate and accuracy of automated web application security scanners.
- **Comprehensive testing or detection:** Exhaustive testing or detection, specifically with respect to whether WVs detect all known categories of the vulnerabilities they claim/advertise to detect or are shown to detect using a WV benchmark. For example, according to OWASP, Cross-Site Scripting (XSS) attacks have three categories (Stored XSS, Reflected XSS, and DOM Based XSS) [7]. Comprehensively detecting XSS attacks would include detecting all of the categories listed equally effectively.
- **Attack vector:** The method or way by an adversary can breach or infiltrate a system.
- **SQL**: Standard Query Language. Used to communicate with a Relational Database Management System (examples include MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access). There are different versions of the language (different *dialects*) that coincide with different systems such as SQL Server, Oracle, MySQL, and PostgreSQL.
- **SQL Injection:** An injection attack where an attacker changes the effect of an SQL query by inserting new keywords or operators. In other words, an attacker changes *how* a database is modified. This type of attack can be used to have a database release sensitive information, delete important information or allow an attacker to bypass authentication and gain administrative privileges [M].
- **Cross-Site Scripting Attack:** A type of injection attack where an attacker inserts malicious Javascript into a web page or web application via user input. The web page or application containing the malicious code (usually Javascript, but other languages such as HTML can also be used) then delivers the code to a victim's browser which then executes the code. Malicious code of this form can be used to access user's cookies, session tokens or other sensitive information retained by the user's browser and used with that site. It can also be used to modify the content of the HTML page it's linked with [8].
- **Document Object Model (DOM):** Object-oriented representation of the web page (HTML or XML documents) which can be modified with a scripting language such as Javascript. In other words, it's the interface that allows programs to change a web page's structure, style and content.

Introduction

Open Source Web Application Vulnerability Scanners (WVs) are desirable for individual developers and small teams because of the high cost of commercial software (IBM's AppScan Enterprise Reporting Only User Authorized User Single Install License + SW Subscription and Support for 12 Months is $1,650 [5]). Because there are many Open Source WVs, choosing an effective option is a task itself. In order to make a good decision, one needs an evaluation of Open Source WV's detection of significant vulnerabilities to protect against associated threats to their application.

Essential criteria for evaluating WVs is established through different benchmarks such as Web Input Vector Extractor Teaser (WIVET), Web Application Vulnerability Scanner Evaluation Project (WAVSEP) benchmark, and Open Web Application Security Project (OWASP) benchmark [13]. Using these benchmarks, security researchers have done evaluations on Open Source WVs [1, 13] which one could use to make a decision on which WV to use. Unfortunately, the notable benchmarks listed do not emphasize comprehensibility when selecting attack vectors for testing. In other words, some attack vectors are covered comprehensively while others aren't depending on the benchmark used.

To further emphasize the blind spot around comprehensively testing Open Source WVs, previous literature on reviewing or assessing Open Source WVs has efforts focused on running times or performance speed [12, 15], crawler coverage [12], detection accuracy [1, 9, 12], and criticism of evaluation methods [9]. Not enough research has been done on the detection scope of advertised vulnerabilities for Open Source WV.

To raise awareness of the blind spot surrounding comprehensively testing Open Source WVs and to give an example of how one would achieve comprehensive testing, the purpose of this project will primarily be to comprehensively test two effective Open Source WVs along two significant attack vectors. Testing effective Open Source WVs is important so this research can be used to give more useful information surrounding tools developers are likely to use. Using significant attack vectors for the testing is important to give more useful results surrounding prevalent and persistent attacks they will likely have to defend their products against (which, again, will be tested for using WVs).

Zed Attack Proxy (ZAP), and *Vega* were chosen as the target WVs/testing tools using the best/highest SQLi and XSS testing results from a report titled *Evaluation of Web Application Vulnerability Scanners in Modern Pentest/SSDLC Usage Scenarios* by Shay Chen, a well-established security analyst and researcher [1]. Chen evaluated SQLi using GET and POST input delivery vectors in 136 valid test cases, and 10 false positive categories and found *Arachni*, *SQLmap*, *IronWasp*, *Wapiti* and *Vega* to have the best SQL Injection detection / false-positive ratios. *Arachni* and *IronWasp* are great choices for evaluation, however both are no longer maintained [10, 11]. Chen evaluated XSS using GET and POST input delivery vectors in 66 valid test cases, and 7 false positive categorie and found *IronWasp*, *Vega*, and *ZAP* to have the best XSS detection / false-positive ratios.

SQLi and XSS were chosen as the two attack vectors because of their categorizable nature making them natural candidates for a comprehensive analysis and because of their importance in today's security climate. Both were also included on the Open Web Application Security Project (OWASP) Top Ten 2017 [14] which offers a list of the ten most critical web application vulnerabilities including cross-site scripting, injection, and sensitive data exposure. The list represents a broad expert consensus of the most critical security risks. In addition to OWASP's 2017 Top Ten report, the two were considered among the most prevalent web application attacks in the European Union Agency for Cyber Security's (ENISA) Threat Landscape report titled "Web Application Attacks" [3] which lays out the most prevalent Web Application Attacks from 2019 research. Aside from SQLi and XSS, broken authentication, session management, local file inclusions, and directory traversal were the other most prevalent attacks.

Related Work

Related work is categorized in two ways: approaches to evaluate and compare WVs and analyzing WVs effectiveness in key areas.

For approaches to evaluating and comparing WVs, Fonseca et al. injected realistic software faults (common programmer bugs) that could potentially lead to vulnerabilities using a software injection technique called G-SWIFT (which they slightly modified for web application code), which emulates the most frequent types of faults. After injection, they manually inspected the potential vulnerabilities caused by each injected fault in order to have a true positive benchmark for determining false positives detected by each WV and then performed the scans using the WVs. They used the results to determine detection accuracy and false positive rates of SQLi and XSS-related vulnerabilities for fault types that generated SQLi and XSS vulnerabilities [4]. Mburano and Si explain three benchmarks for evaluating WVs: Web Input Vector Extractor Teaser (WIVET), Web Application Vulnerability Scanner Evaluation Project (WAVSEP) benchmark, and Open Web Application Security Project (OWASP) benchmark [7]. Their paper *Evaluation of Web Vulnerability Scanners Based on OWASP Benchmark* evaluated WVs using the OWASP benchmark and compared results with WAVSEP. They concluded that WVs perform differently in different attack categories and no scanner should be considered all-round tool for vulnerabilities.

For analyzing WVs effectiveness in key areas, the paper *An Analysis of Black-Box Web Application Security Scanners against Stored SQL Injection* by Khoury et al. [6] categorized four essential stages in black-box testing (i.e. the type of testing WVs do): (1) crawling, (2) forms and entry point identification, (3) attack code construction and (4) submission and analysis and replies. Their paper gives an analysis on why WVs have a hard time detecting stored SQLi such as not using proper attack codes to exploit stored SQL injection vulnerabilities for a multi-step attack (in other words, seeing a syntax error and completing the attack based on that error). Their paper concludes with a recommendation for improving WVs based on their measured weak SQLi detection rate. Specifically, improving WVs state full scanning, input selection based on field name and label, attack vector novelty, server reply analysis and post scanning. In *Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners* [2], Doupé et al. created *WackoPicko*, an intentionally vulnerable website, to display and test vulnerabilities from the OWASP top 10. They tested WVs on *WackoPicko* across its attack vectors, analyzed the results and gave significant recommendations based on the challenges the WVs faced. Specifically, support for well-known and pervasive technology (such as Javascript) needs to be improved. The paper found this lack of support prevented WVs from reaching valuable pages altogether. "Deep" crawling was also found to be beyond the scope of the WVs evaluated. Specifically this refers to being able to handle different application states. The example they give is that some internal variables are only exposed to a user (and thus pose a vulnerability threat) when they are logged in. Lastly, they found a significant weakness in WVs failure to detect application-specific vulnerabilities.

Both approaches listed (intentional software fault injection and using benchmarks for testing) would benefit from ensuring their attack vectors used for testing were comprehensive in order to provide a more complete measurement of detection. In other words, leaving out a significant type of attack or attack mechanism may lead to undetected vulnerabilities.

Categorization of essential black-box testing and recommendations based on WV weaknesses allows for potentially more focused evaluations and testing methodologies for the improvement of WVs. Comprehensively evaluating WVs can lead to betterment of attack code construction, an essential black-box testing category.

Design and Implementation

SQL Injections

In *A Classification of SQL-Injection Attacks and Countermeasures*, W.G. Halfond et al. give a complete description of SQLi types and injection mechanisms [17]. A WV comprehensively tests for SQLi if it tests for each type of attack vector listed in the paper at least once.

Types of SQLi attacks:

- Tautologies
  Inject code in one or more conditional statements so they always evaluate to true. Attacker exploits an injectable field that is used in a query's WHERE conditional. Transforming the conditional into a tautology causes all of the rows in the database table targeted by the query to be returned.

- Illegal/Incorrect Queries
  Inject code that will help attackers gather information about the type and structure of the back-end database of a web application.

- Union Query
  Inject code to exploit a vulnerable parameter to change the data set returned for a given query.

- Piggy-Backed Queries
  Attacker attempts to inject additional queries into the original query. As a result, the database receives multiple SQL queries. The first query is normal and the following query is a malicious payload.

- Stored Procedures
  Attempt to try to execute stored procedures present in the database. Most databases come with a standard set of stored procedures that extend the functionality of the database and allow for interaction with the operating system.

- Inference
  Query is modified in the form of an action that is executed based on an answer to a true/false question about data values in a database. Blind injection (Information must be inferred from the behavior of the page by asking the server true/false questions) and timing attacks (Allows an attacker to gain information from a database by observing timing delays in the response of the database) are two well-known attack techniques based on inference.

- Alternate Encodings
  Encoding attack strings to avoid a common defensive practice to scan for certain known "bad characters", such as single quotes and comment operators.

SQLi Test Site

OWASP Juice Shop is an intentionally vulnerable open source web application written by Bjorn Kimminch. For all tests and scans, the web page was hosted locally on a Macbook Air running OS Version 10.14.6. I used Firefox to conduct manual penetration testing where necessary. The SQLi vulnerability the WVs needed to catch was Juice Shop's "login" form.

Cross-Site Scripting Attacks

OWASP gives a complete description of each type of cross-site scripting attack [7]. A WV comprehensively tests for XSS if it tests for each type of attack vector listed at least once.

Types of XSS attacks:

- Stored XSS (also called Persistent or Type 1)
  Malicious code is sent to a web page or web application by an attacker through the page/application's user input. Examples of user input include a message forum, a comment field and visitor log. Once the malicious code is stored (such as on a database), a victim unknowingly retrieves the stored code from the web page/application and the victim's browser runs the code.

Blind XSS is a subset of stored XSS, with the distinction being the victim doesn't *see* (such as in the case of seeing the attack via Javscript's alert()) the XSS being executed.

- Reflected XSS (also called Non-Persistent or Type 2)
  Malicious code is sent through user input and is not processed in such a way to make it safe to render in the browser (filtering, validating, encoding, etc.). The input (code) is immediately returned by a web application in an error message, search result or some other response that uses some portion of input as part of the request, thus allowing for the execution of the malicious code.

- DOM Based XSS (also called Type 0)
  Malicious code is sent through a source that stores the code as part of the DOM (such as in an HTML element or the URL of a web page). The code is then processed (snuck in) without filtering, validating or encoding. The code is then executed by the browser via vulnerable functions that access the DOM (such as Javascript's document.write). The distinction here is that the entire flow of malicious code processing takes place in the browser (input is in the DOM and output is also in the DOM). An example is an attacker finding that a webpage prints from the URL bar (window.location.search), then placing malicious code in the location and the browser running the malicious code without usual data validation.

XSS Test Sites

OWASP Juice Shop (explained in "SQLi Test Site" above) contains a DOM based XSS vulnerability accessible via the site's search bar.

Altoro Mutual is an intentionally vulnerable website published by the IBM Corporation for the sole purpose of demonstrating the effectiveness of AppScan (a paid WV) in detecting web application vulnerabilities and website defects. I used the same setup and configuration described in the SQLi Test Site" above. *Altoro Mutual* contains a Reflective XSS vulnerability accessible via the site's search bar.

Hackazon is an open-source project developed by Rapid7 that incorporates a purposely vulnerable practical ecommerce website. I used the same setup and configuration described in the SQLi Test Site" above on *Hackazon*. *Hackazon* contains a stored XSS vulnerability accessible via the site's Helpdesk "Enquiries" feature.

Zed Attack Proxy (ZAP)

OWASP's Zed attack proxy (ZAP) is a proxy that sits in between your computer and the web applications you visit and allows you to intercept traffic and modify responses sent between you and the application. ZAP sets itself up on a port that you redirect all your traffic through on the local host. ZAP has two particular use cases I'll focus on: automated scanning and proxying via zap and then scanning.

ZAP SQLi Detection

Once an application is shown to be vulnerable to SQLi, using ZAP's Fuzzer (uses fuzz testing which is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program) with ZAP's SQLi testing scripts on the vulnerable location allow for at least one test for every type of SQLi. The scripts are located from the Fuzzer in the following path: Add → Add Payload (select "File Fuzzers") → jbrofuzz → Injection. This functionality is based on code from the OWASP JBroFuzz project and includes files from the fuzzdb project.

The configuration used with active scan to check for SQL injections (SQLi) on the *OWASP Juice Shop* webpage had the "SQL injection" attack category on the maximum attack setting (threshold set to "high", strength set to "insane") and all other attack categories on the minimum attack setting (threshold set to "low" and strength set to "low").

ZAP’s active scan on the *OWASP Juice Shop* webpage successfully revealed an endpoint (/rest/products/search) vulnerable to SQL injection. After manually submitting a POST request through Juice Shop’s “login” form in Manual explore mode, ZAP successfully revealed the form to be vulnerable to SQL injection.

ZAP XSS Detection

To detect XSS vulnerabilities, *OWASP Juice Shop, Altoro Mutual* and *Hackazon* were used as the test sites and I used the same set set-up (macOS, Firefox, etc) as the section above.

The configuration used with active scan to check for XSS vulnerabilities on each webpage had the “XSS” attack category on the maximum attack setting (threshold set to “high”, strength set to “insane”) and all other attack categories on the minimum attack setting (threshold set to “low” and strength set to “low”).

OWASP Juice Shop has a DOM-based XSS vulnerability that can be exploited using either the page’s search bar or by modifying the corresponding URL (e.g. https://localhost:3000/#/search?q=test). ZAP’s active scan and manual scan (after submitting a search request through the search bar) did *not* detect this vulnerability.

ZAP was able to detect the Reflective XSS vulnerability on *Altoro Mutual* using both active scan and manual explore mode.

ZAP was unable to detect the Stored XSS vulnerability on *Hackazon* described above using either active scan or manual explore mode.

Results and Analysis

The following table summarizes the types of SQLis that can be tested from ZAP’s Fuzzer using the scripts (at the *OWASP Juice Shop* SQLi vulnerability) mentioned above.

Type	Script	Example Attack Code	Syntactical Support
Tautology	MySQL Injection 101 MS SQL Injection i Oracle SQL Injection	1 or 1=1 ' or 1=1 -- ' or '1'='1	MySQL MS SQL Oracle SQL
Illegal/Incorrect Query	MySQL/MS SQL Common Injection Oracle SQL Injection	' or username is not NULL or username = ' ' AND 1=utl_inaddr.get_host_address((SELECT SYS.LOGIN_USER FROM DUAL)) AND 'i'='i	MySQL/MS SQL Oracle SQL
Union Query	MS SQL Injection i	' union (select @@version) --	MS SQL

	MySQL/MS SQL Common Injection	1 union all select 1,2,3,4,5,6,name from sysobjects where xtype = 'u' --	MySQL
Piggy-Backed Query	Active SQL Injection	' ; drop table temp --	MySQL/MS SQL
Stored Procedures	Active SQL Injection Oracle SQL Injection	exec sp_addsrvrolemember 'name' , 'sysadmin' ' || myappadmin.adduser('admin', 'newpass') || '	MySQL/MS SQL Oracle SQL
Inference	MS SQL Ninja Injection (Blind) MySQL/MS SQL Common Injection	'; if not(substring((select @@version),25,1) <> 0) waitfor delay '0:0:2' -- 1 and ascii(lower(substring((select top 1 name from sysobjects where xtype='u'), 1, 1))) > 116	MS SQL MySQL/MS SQL
Alternate Encodings	Passive SQL Injection	'||(elt(-3+5,bin(15),ord(10),hex(char(45))))	Generic

Overall, ZAP is comprehensive if the Fuzzer is used in a SQLi-vulnerable location. The default scripts themselves unevenly (from a comprehensiveness perspective) support SQL-variant syntaxes.

The following table summarizes ZAP's success detecting XSS attacks.

Type of XSS Attack	**Detection using Active Scan mode**	**Detection using Manual Explore mode**
Stored	No	No
Reflected	Yes	Yes
DOM Based	No	No

Overall, ZAP successfully detected one of the XSS attack vulnerabilities and is therefore *not* comprehensive. ZAP features fuzzing scripts and advertises support for each XSS type, but these are only useful if the vulnerabilities are detected in the first place.

Vega

Vega is a free and open source web security scanner and web security testing platform. Vega can help find and validate SQL Injection, Cross-Site Scripting (XSS), inadvertently disclosed sensitive information, and other vulnerabilities. It is written in Java, GUI based, and runs on Linux, OS X, and Windows [16].

Vega SQLi Detection
Vega's automated scan failed to detect *OWASP Juice Shop*'s SQLi vulnerability. Vega doesn't allow for the transparency ZAP does with respect to fuzzing scripts. Instead one can only infer based on the names of their "Injection Modules". These include: Blind SQLi, Time-based SQLi, and Blind SQLi Arithmetic Evaluation Differential check.

Vega XSS Detection
Vega did not detect *any* of the XSS attack vulnerabilities.

Discussion of Issues
Overestimating a reasonable project scope was a continued issue throughout the process of collecting research for the project. Because of the incredible amount of time each attack vector takes and the strong learning curve for each WV, I had to reduce the total vectors from ten to two and total WVs from four to two. I believe the scope I ended up with is reasonable for an eight week period.

Conclusion
Categorizing additional attack vectors and comprehensively testing WVs for them would be a natural extension for the research done in this paper. Additional research should be also done regarding WV detection comprehensibility across attack mechanisms. For example, W.G. Halfond et al. classified the following injection mechanisms [17]: injection through user input (Example: HTTP GET/POST requests), injection through cookies, injection through server variables (example: HTTP, network headers, environmental variables) and second order injection (seeding a malicious input into a system or database to indirectly trigger an SQLi when the input is used at a later time). Additional mechanisms may exist for different types of attack vectors.

ZAP is a wide-ranging and useful tool for vulnerability detection in software and ZAP's robust SQLi scripts make it a solid choice for SQLi testing. I would recommend a developer or small team going this route be sure to make sure that the SQL dialectical variant they're testing is covered comprehensively by the ZAP SQLi scripts. While ZAP does provide some level of support for all three types of XSS attack vulnerabilities, it did not catch the vulnerabilities I tested for which does not make me feel confident recommending one to use ZAP for XSS vulnerabilities.

Compared to ZAP, Vega is not a tool I would recommend. Vega failed to detect all of the tests and did not provide transparent comprehensibility on the level of ZAP.

Lessons Learned
In a real-world setting, a WV is not a catch-all for discovering vulnerabilities. It is a useful tool to help an already well-educated (with respect to security) developer uncover accidental blind spots. While a solid WV does help, it's much more important to be educated on the attacks and vulnerabilities you're defending against. Furthermore, success of a WV is deeply connected to whether a developer has invested the necessary time to learn the tool. I would recommend learning one good WV deeply rather than using many WVs in the pursuit of a perfect detection rate.

Comprehensiveness of attacks is important for defense against attacks, but equally important is comprehensiveness of attack *mechanisms*. One needs to be well educated on all the ways an attacker can penetrate a system in order to effectively defend against attacks. Knowing how to use a WV such that one can test these different mechanisms is an essential skill for good defense.

The industry has a good awareness of XSS attacks and most literature I saw on the subject knew about all the different types of attacks. This is less-so regarding SQLi attacks. There is a potential blind spot here and comprehensiveness as a methodology can help one be aware of this blind spot. Also, developers should make sure their WV supports the syntax of SQL they're using on a project.

References

[1] Chen, S. (2017, November 10). WAVSEP 2017/2018 - Evaluating DAST against PT/SDL Challenges. Security Tools Benchmarking. http://sectooladdict.blogspot.com/2017/11/wavsep-2017-evaluating-dast-against.html.

[2] Doupé, Adam & Cova, Marco & Vigna, Giovanni. (2010). Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. Proc. DIMVA 2010. 6201. 111-131. 10.1007/978-3-642-14215-4_7.

[3] ENISA Threat Landscape 2020 - Web application attacks. ENISA. (2020, October 20). https://www.enisa.europa.eu/publications/web-application-attacks.

[4] Fonseca, José & Vieira, Marco & Madeira, Henrique. (2008). Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks. Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium On. IEEE. 365 - 372. 10.1109/PRDC.2007.55

[5] *IBM AppScan Pricing*. Npm. https://www.ndm.net/sast/appscan-pricing.

[6] Khoury, Nidal & Zavarsky, Pavol & Lindskog, Dale & Ruhl, Ron. (2011). An Analysis of Black-Box Web Application Security Scanners against Stored SQL Injection. 1095-1101. 10.1109/PASSAT/SocialCom.2011.199.

[7] kingthorin, Blankenship, H., Grossman, J., adamczi, & mootooki. (2020, August 27). *Types of XSS*. OWASP. https://owasp.org/www-community/Types_of_Cross-Site_Scripting.

[8] KirstenS, Manico, J., Williams, J., Wichers, D., Weidman, A., Roman, … kingthorin. (2020, September 20). Cross Site Scripting (XSS). Cross Site Scripting (XSS) Software Attack | OWASP Foundation. https://owasp.org/www-community/attacks/xss/.

[9] K. McQuade, "Open source web vulnerability scanners: the cost effective choice?" in Proceedings of the Conference for Information Systems Applied Research, vol. 2167, p. 1508, 2014.

[10] Kuppan, Lavakumar (2013 September 20). *IronWasp* [electronic source code]. https://github.com/Lavakumar/IronWASP/tree/master/IronWASP

[11] Laskos, Tasos (2020, March 6). Arachni Is No Longer Maintained. Arachni. www.arachni-scanner.com/blog/arachni-is-no-longer-maintained

[12] Mansour Alsaleh, Noura Alomar, Monirah Alshreef, Abdulrahman Alarifi, AbdulMalik Al-Salman, "Performance-Based Comparative Assessment of Open Source Web Vulnerability Scanners", Security and Communication Networks, vol. 2017, Article ID 6158107, 14 pages, 2017. https://doi.org/10.1155/2017/6158107

[13] Mburano, Balume & Si, Weisheng. (2018). Evaluation of Web Vulnerability Scanners Based on OWASP Benchmark. 1-6. 10.1109/ICSENG.2018.8638176.

[14] Open Web Application Security Project. (2017). OWASP Top Ten. Retrieved from https://owasp.org/www-project-top-ten/

[15] Suteva, Natasa & Zlatkovski, Dragi & Mileva, Aleksandra. (2013). Evaluation and Testing of Several Free/Open Source Web Vulnerability Scanners..

[16] Subgraph. “Vega Vulnerability Scanner.” Subgraph, subgraph.com/vega/.

[17] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL-Injection Attacks and Countermeasures. In Proc. of the Intl. Symposium on Secure Software Engineering, Mar. 2006