gFuzz: An instrumented web application fuzzing environment

Ezequiel D. Gutesman
Corelabs, Core Security Technologies.
Humboldt 1967 2do piso
Buenos Aires, Argentina

Abstract

Web application fuzzers have traditionally been used by security experts as a first step in a security assessment. They typically produce false positive alerts and all the vulnerability reports must be carefully studied. We introduce a new fuzzing solution for PHP web applications that improves the detection accuracy and enriches the information provided in vulnerability reports. We use dynamic character-grained taint analysis and grammar-based analysis in order to analyze the anatomy of each executed SQL query and determine which resulted in successful attacks. A vulnerability report is then accompanied by the offending lines of source code and the fuzz vector (with attacker-controlled characters individualized). As a result, the usage of the tool is not restricted to security experts, but the tool becomes usable for developers. The prototype is available as open source software.

1 Introduction

Web Applications are widely used by organizations and companies. They can provide a vast variety of services but most of the times, they interact with databases, handle sensitive information and sometimes deliver critical services. It is in the interests of the owner and users of a web application that it is secure so that its availability cannot be compromised, the private data that it hosts is not stolen, and its integrity and security are not damaged. To do this, the development cycle must address security issues which may represent a threat to the aforementioned concerns.

Different types of vulnerabilities might be present in a web application, for example cross-site scripting and SQL-injection attacks [Opec]. We will focus our work on injection vulnerabilities [Oped], more precisely SQL-injection vulnerabilities [Opeb]. SQL-injection vulnerabilities are exploited, whenever specially crafted user-supplied data reaches the web application execution environment (i.e., code interpreter or virtual machine) as part of a command or query, tricking this environment into executing commands which the web application was not designed to execute. This behavior is triggered when user-supplied data is used to query the back-end database management system (DBMS for short) without taking the proper precautions (e.g., through a sanitization procedure).

As a result, an attacker might be able to execute arbitrary SQL queries and therefore steal or modify the data in the database.

Last year numerous SQL injection attacks were reported [web07]. These attacks have resulted in data theft and unavailability, and considerable money losses (depending on the severity of the attacked vulnerability). To prevent attackers from exploiting these vulnerabilities one can use ad hoc protection systems (e.g., using an application firewall) or detect them and fix the code so they are no longer a threat (e.g., before they are deployed); see [SSS06].

Web application firewalls [Imp] stand in front of the web application, but since each custom-built application might have unique vulnerabilities and attack signatures must be built for each of these, signature-based detection remains imprecise. Further, for this same reason statistical detection also fails to stop all the vulnerabilities.

Static code analysis tools [HYH+04, LL05] intend to detect, in the source code and before deploying the web application, common programming errors that allow injection attacks. Unfortunately static analysis tools fail to detect complex attack vectors (more on this in Section 2.4) since run-time information is required to detect some of these attacks. Dynamic code analysis tools [NTGG+05, FGW07, Ven06, HO06, BK04] and scanners [Ins] deal with run-time vulnerability prevention and detection. Dynamic code analysis tools can detect attacks on-the-fly but must have a good balance between performance penalty and accuracy since they form a last line of defense before the attack takes place and do impact on the perceived performance of the application. There are some tools which combine both static and dynamic methods such as [LMLW08] which require the developer to specify meta-information or define security policies over the application being analyzed.

Notwithstanding, a commonly adopted practice is to perform black-box (or gray-box) security audit against the application in the staging phase based on a fuzzing process, which consists in sending the web application specially-crafted data through valid application input vectors (e.g., form input fields, URL parameters, cookies) and analyzing those that trigger exceptions, since they could lead to vulnerabilities [SGA07].

The security vulnerabilities found should be fixed by successfully sanitizing input data, e.g., detecting single quotes, `<script` tags and combinations with other characters which are commonly used to trigger some error in SQL queries or HTML output respectively.

Several automated fuzzing tools have been created over the last years, both open source projects and commercial products [Imm, Rog, Por, Nic, HP, Com]. Actually, commercial products integrate fuzzing to other vulnerability detection functionalities. The first fuzzers have grown to include authentication, session handling, interactivity and some post-exploitation functionalities, but only a few address exception analysis towards precision.

A reported exception can indicate a vast spectrum of possibilities: a real vulnerability, a real bug inside the code (with no security impact) or a false positive exception which can arise in the form of any of the aforementioned. The security tester usually has to manually review the reported exceptions in order to assert their severity, requiring deep experience in code review and some level of understanding of the analyzed web application [Jod03].

The main goal of this work is to give the security tester enough information to accurately distinguish real security vulnerabilities (which could or could not

be possible to exploit) from the rest of the alerts, integrating accurate exception analysis with fuzzing tools. This will help to solve a yet open problem: assert at run-time and from the attacker's point of view, when an executed SQL query is the product of a SQL-injection attack.

Our work is based on fuzz-based web application testing. We introduce an accurate technique for detecting and reporting exceptions in PHP applications (although it can be generalized to other programming languages [CW08]). For each reported exception we are able to determine: which commands led to the successful execution of each query inside the web application and how they were triggered from a fuzzing standpoint, reflecting which parts of each query are controlled by the attacker. By analyzing the anatomy of the executed queries, the system is able to classify fuzz vectors that exploit SQL-injection vulnerabilities with high accuracy, allowing the tester to refine the attacks performed and determine the risk they represent.

The proposed technique uses classical fuzzing and crawling techniques combined with precise information about the queries executed inside the tested web application (i.e., a mixture between server-side instrumentation and developer-side testing) and a grammar-based analysis. No modification of the application is needed since the test is performed from the attacker's point of view. Although, the application must run in an instrumented execution environment [FGW07] which provides precise information allowing the security tester to have real-time feedback available. Then, the accuracy of the fuzzing process and the test as a whole is significantly increased.

The solution allows us to combine a fuzzing tool with additional information that will help to better categorize exceptions and distinguish between legitimate SQL queries and attacks. The precise information used will be delivered by the instrumented execution environment and will be explained in Section 2.1. On the fuzzing environment side, the solution uses a SQL grammar-based analysis similar to [HO06] that analyzes the structure of the query using also the precise security information.

1.1 Anatomy of a SQL injection attack

As a particular case of injection attacks, a SQL-injection attack occurs when input sent by an attacker reaches the back-end DBMS without adequate checks (e.g., adequate string escaping). The attacker is then able to modify the structure of a predefined query in order to execute queries that fall outside the scope of the designed functionalities.

Nowadays, most web applications are developed using so called scripting languages (e.g., PHP, Python, ASP). The usage of scripting languages makes interacting with DBMS easier. The web user makes a request to the web server and this causes a language-specific interpreter or virtual machine (in languages such as Java or ASP.NET) to execute a script. For example, a script will accept an input from a web user and use this input to perform a query to the database creating a web page in response to the request processed.

Consider this PHP command , which in turn contains a SQL command, where `$user` and `$pass` represent the text variables entered by a user:

Unless the user input is verified (and sanitized), an attacker might be able to execute any SQL command he wishes.

```
$result = mysql_query(‘‘SELECT name,email FROM users
                        WHERE username=’$user’ and password=’$pass’ ’’);
```

Example 1: Execution of a SQL command using user-supplied data

Continuing with the above example, if the login script allowed a user to submit arbitrary values (e.g., with no sanitization), a web user without valid login credentials might be able to modify the structure of the query in order to log into the website. When user *bob* wants to log in with password *foo*, the web application produces the following query:

```
SELECT name,email FROM users
WHERE username=’bob’ and password=’foo’
```

On the other hand, an attacker using user *noname* and password *nopass' or 1=1* can cause the web application to submit the following query to the DBMS:

```
SELECT name,email FROM users
WHERE username=’noname’ and password=’nopass’ or 1=1;--’
```

Example 2: An attacked SQL query

This query would result in the DBMS evaluating the tautological condition `1=1` and returning all the records in the *users* table hence, he would be able to log in without having valid credentials. It is evident how the original query was modified in order to execute a "slightly" different one.

While the above attack exemplifies the anatomy of a SQL-injection attack, it is simple and easy to prevent. Yet, SQL-injection vulnerabilities do get more complicated as web applications grow in size and complexity. Sometimes a vulnerable web application will use insufficient sanitization that has to be analyzed carefully in order to circumvent it, sometimes the exceptions will provide little information or the web application will execute the attacker's queries but will not return the data explicitly. For example, blind SQL-injection vulnerabilities [Opea] require more effort in order to be detected since no visible output can be obtained inside the response sent by the web server.

An important feature of the work presented is that it does not require special treatment for blind SQL-injection vulnerabilities since it does not rely on the application's output to detect vulnerabilities. Detecting anomalies (or exceptions) in the web application's behavior is as important as refining attack vectors in order to perform a more accurate tests, and there's a close relation between the exception detection task and the fuzzing process.

2 Detecting SQL anomalies with gFuzz

Detection within gFuzz is done by combining taint analysis of the characters that form a query with a grammatical analysis of the query.

2.1 Character-grained taint analysis

Taint analysis is an information-flow analysis technique commonly used for detecting injection vulnerabilities during run-time [FGW07] or development [Hur04, Ven06]. Functionally, a taint analysis is done over a running script by adding taint marks to all the user-provided data that enters the scripting language interpreter, propagating taint marks during execution and producing alarms when data with taint marks reaches (predefined) sensitive APIs or functions.

We have chosen to assign taint marks per character. This is implemented in the Core Grasp [Cor07] open source solution.

Our implementation of the technique [Cor07] is a modification of the PHP interpreter which automatically tracks security information about the data entering the web application through all the input vectors (e.g., GET, POST, COOKIES). It can run in any platform PHP can run. Every PHP script runs inside the modified interpreter allowing dynamic security taint marks tracking in run time. The technique used is based on character-grained taint analysis [FGW07, NTGG$^+$05]. Per-character security information is propagated during execution among string operations.

The modified interpreter monitors SQL queries sent to MySQL databases and sends gFuzz an entry per query to be executed with security information attached. For Example 1 the entry produced by Grasp is:

```
<GRASP_FUZZ_ENTRY>
  <GRASP_QUERY_ID>
    /location/of/the/executed/file/userlogin.php:40
  </GRASP_QUERY_ID>
  <GRASP_FUZZ_IS_ATTACK>0</GRASP_FUZZ_IS_ATTACK>
  <GRASP_FUZZ_QUERY>
   SELECT name,email FROM users WHERE username='bob' and password='foo'
  </GRASP_FUZZ_QUERY>
  <GRASP_FUZZ_QUERY_MARK>
   ........................................XXX.............XXX.
  </GRASP_FUZZ_QUERY_MARK>
</GRASP_FUZZ_ENTRY>
```

Example 3: Entry for one SQL query executed inside the instrumented execution environment

As the implemented character-grained analysis is performed from within the execution environment it can supply detailed information: file location and line where the SQL query was executed, Grasp information of whether it could represent an attack (this is not a grammatical but a lexical analysis), query text and query security mark that gFuzz will use to categorize exceptions and increase the information provided to the tester.

Inside the instrumented execution environment all the executed queries are buffered. When the application flushes all its output (i.e., HTTP headers and content) all the buffered entries are sent together with the response, as shown in Example 3.

2.2 Grammatical analysis of SQL queries

Grammar-based analysis is a widely used technique, specially in program analysis [Thi05] and for test generation [BM83]. It is also used for static program analysis focused on security [GKL08, LMLW08, LL05]. It consists in specifying well-formed strings or constructions, sometimes depending on the analysis context, which describe a given language. SQL queries have a grammar which defines the valid structure a query has to be compliant with in order to be executed (defined in the SQL-92 standard), some DBMS extend this grammar in order to give additional capabilities.

Our work uses a SQL grammar in order to analyze an determine if a given SQL query, together with its taint marks, is valid from a security perspective.

Figure 1 describes the structure of the query from the examples in Section 1.1 in a tree diagram. By comparing the diagrams for Example 1 (legitimate SQL query) and Example 2 (attacked SQL query) we notice that new nodes are added in the latter. The terminal nodes, that is nodes without childs, that are controlled by the attacker altered the grammatical structure of the query. This modification leaves the rest of the query conditions below the *WHERE* node superfluous, since all the rows match the tautological condition `1 = 1`. The shadowed nodes represent the information controlled by the attacker which lead to the successful attack.

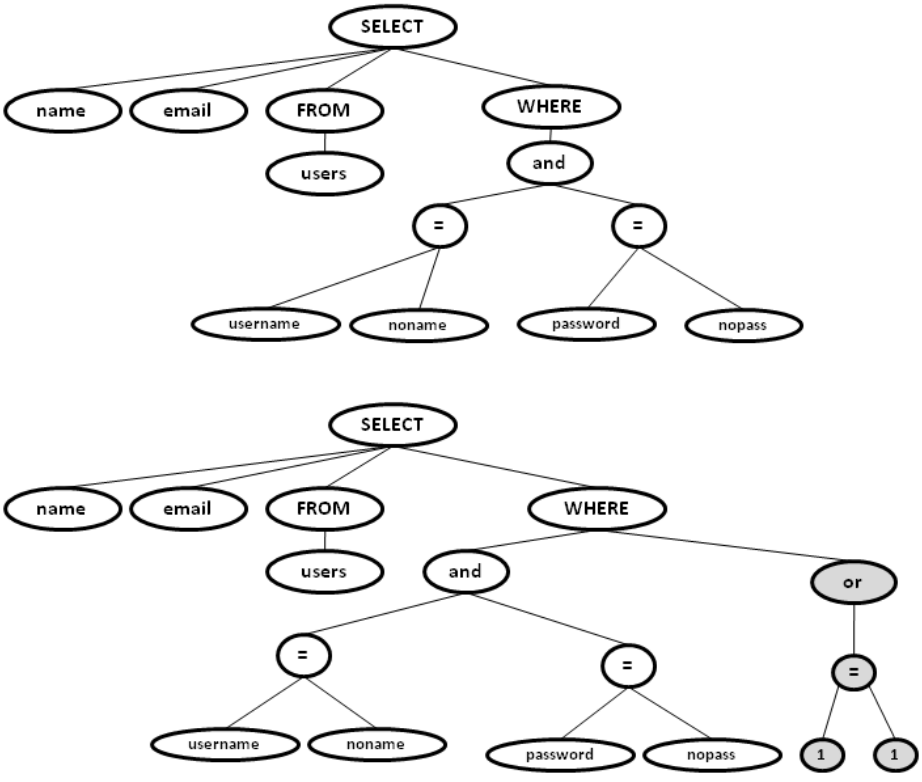

Figure 1: Syntax tree of a valid SQL query and the attacked version.

Notice that in Example 2 the query produced is valid according to the

MySQL parser. An attack will only work if the altered query is also valid, hence executable by the DBMS, as opposed to an example where the attacker adds a single quote, which is not escaped and results in an invalid query, not executed by the DBMS.

2.3 Combining grammar-based with character-grained taint analysis

gFuzz uses a fuzzer, which can be replaced by the fuzzer of the tester's choice, to provide the fuzz vectors to the web application under analysis, and uses the SQL-grammar analysis combined with taint-analysis information to classify SQL-queries and detect attacks. Explicitly we analyze every SQL query improving over other fuzzing technologies which only analyze exceptions that can be inferred from explicit output. The architecture of the solution is summarized in Figure 2.

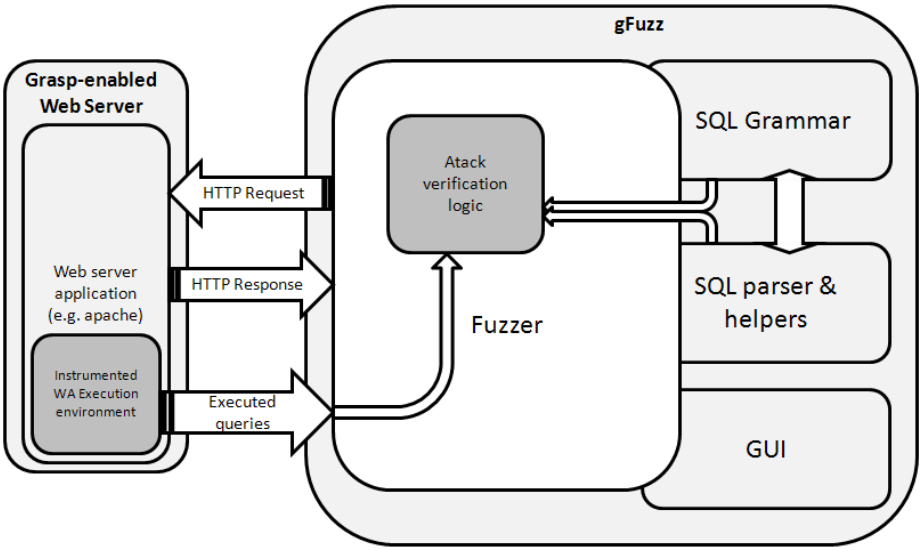

Figure 2: gFuzz Architecture layout

A fuzzing round with gFuzz can be divided into three steps:

Fuzzing targets The fuzzer receives a list of target web pages on a web application and a list of *fuzz strings* (i.e., chains of specially-crafted characters which try to break some functionality). It then starts testing each target (e.g., submitting form entries and URL parameters) storing the submitted *fuzz string* for each target/input vector. In particular, a set of predefined **harmless** input values are included to fuzz each entry point. These values serve as **witness** fuzz strings used to **profile** the structure of the query as intended by the developer.

Web application monitoring On the server side, the web application runs inside the instrumented fuzzing environment. Each query sent to the database

is sent back to gFuzz through standard HTTP responses with all the information described in Section 2.1. This information exchange is done with caution so that any headers sent by the original web application logic are preserved. To accomplish this, Grasp buffers each `FUZZ_ENTRY` and sends all the entries together once the script flushed all the content it needs.

Analysis We assume that **profiling** is successful, in other words, the **witness** fuzz strings trigger all the possible SQL queries generated at this data entry point[1]. gFuzz detects if the profiling is unsuccessful, since it recognizes which line of the PHP script triggered each query, knowing if there is a witness query available to compare with, and performs a different analysis in each case.

While receiving the response for each fuzz vector submitted, gFuzz builds, for each query sent by the execution environment, a tree-like structure (as described in Section 1.1) representing each query executed. A grammar-based analysis is then performed combined with the security information provided and, if the profiling was successful, with the witness queries. gFuzz then classifies each query in an alert category as follows:

A given query is classified as **harmless** if it is a valid query (i.e., it could be successfully parsed by the grammatical analyzer) and no terminal node is controlled by the attacker.

A query is classified as an **attack**, if the attacker is able to fully control a terminal node, its parent and all of its brothers of the SQL grammatical tree. This means the original structure of the query was successfully modified in order to change the final outcome (as shown in Figure 1). The grammar-based analysis detects this anomaly by constructively trying to build a valid SQL query. Finally, when the executed query was successfully parsed it can be classified as an attack.

A query will be classified as a **warning** if it is not a valid SQL query. In other words, it could not be successfully parsed by the grammatical parser, thus not being able to determine whether a terminal node is controlled by the attacker or not. A query classified as a warning might result in a successful attack or not. This class of notifications are reported so the tester can perform a deeper analysis.

gFuzz adds to each alarm the information provided by the instrumented execution environment about the security check performed. This provides more information to the tester at the time of analyzing warnings. Table 1 shows four queries logged and analyzed by gFuzz. The queries reported as successful attacks, are marked with the ATTACK label on the *gFuzz* column. For queries which are not well formed, and hence not executable, the WARNING label is attached. For valid queries the label NO INFO is applied.

Also Table 1 displays in *Grasp* column the classification made by Core Grasp. Where "A" denotes an attack and "OK" a safe query. *Query* column shows the query text where the characters controlled by the attacker are underlined. The last column labeled *Fuzz vector*, shows the fuzz string which triggered the execution of the query.

[1]Sometimes this is not an easy task. For example, if the system sends numeric values in all the inputs of a form the web application logic may cause a sequence S_1 of queries to be executed. If however sending other *fuzz strings* triggers a different sequence S_2 of queries could be executed, the profiling is not successful.

gFuzz	Grasp	Query	Fuzz vector
ATTACK	A	SELECT title FROM books WHERE id = 5 or 1=1 or id = 5 or 1=1	*5 or 1=1*
WARNING	A	SELECT title FROM books WHERE id = ' or 1=1	' or 1=1
ATTACK	A	SELECT title FROM books WHERE id = ' or 1=1 or id = ' or 1=1	' or 1=1
NOINFO	OK	SELECT title FROM books WHERE id = 12345	12345 (witness)

Table 1: Queries with fuzz keywords reported by gFuzz.

Input	Form

`<input type='text' name='id'/>`	/graspExamples/7.php:
	`<form action="index.php" method="post">`
	`<input type="text" name="id" size="40">`
	`<input type="text" name="name" size="40">`
	`<input type="text" name="surname" size="40">`
	`<input type="text" name="age" size="40">`
	`<input type="submit" value="Submit Query">`
	`</form>`

Table 2: Fuzz point informed per-entry

Note that sometimes our taint analysis system accuracy in determining a successful attack is lower than the one provided by gFuzz. This is because gFuzz has, at the time of the test, the grammatical analysis that can be combined with a witness query to compare with.

gFuzz completes the information for each query with references to the source code where the offensive request was handled and the input provided. This allows the tester to associate the fuzz vector with the query executed and the characters controlled. This is depicted in Table 2.

With all this information, the tester is able to focus on refining the reported tests and only work on potentially dangerous fuzz vectors, because he can filter out the queries reported as valid.

2.4 Analysis of the solution

We proposed a new approach to perform web application security audits which consists in a fuzz-based gray-box testing. The technique combines dynamic security information using character-grained taint analysis with a grammar-based analysis. We developed a prototype which handles real-life web applications. It can be combined with another fuzzer implementation different from the simple fuzzer bundled with it.

The proposed solution differentiates itself from static code analysis tools (e.g., [HYH+04, LL05] and [HO06]) since with gFuzz coverage depends on the user-provided fuzzing vectors (and therefore can be constantly improved) and

with static analysis tools coverage depends on the technique itself. Additionally, static analysis tools have suffered from aliasing and other problems ([LR91]) and require a full analysis of the sanitization processes (e.g., regular expressions or escaping functions) and individualizing the sensitive sink functions.

Regarding the use of character-grained taint information, it allows the system to have accurate information without using heuristics to find error codes [Opee].

While in black-box tests (i.e, no source code available) once the attacker is able to execute a modified version of the original query, he might not be able to verify the success of the attack. Heuristic methods are used in order to retrieve data from an attacked SQL database; one possibility is to adopt covert information retrieval techniques such as [RT07b, RT07a, Hot04] so the flaws found can be validated. This problem is not present in the analysis gFuzz performs in run-time.

The combination of a grammar-based analysis, character-grained taint information and the concrete instantiation of one of the possible execution paths allow gFuzz to present a useful and accurate information set to aid and help the tester (developer or security auditor) in the task of performing security audits with information. This would not be available under a static code analysis or a standard dynamic protection by itself. gFuzz allows combination of both approaches and reports in real-time all the database activity the web application is performing.

3 Experimental results and future work

Current implementation has limited fuzzing features and a basic SQL grammar to build a proof of concept. The SQL grammar can be extended to cover the entire SQL language and the fuzzer can be replaced by any other implementation. Besides the instrumented PHP interpreter, which is implemented in C language, the prototype is entirely developed as open source software in Python language.

On a first accuracy test we tested popular web applications with published vulnerabilities, exploited these vulnerabilities manually and then tested them with gFuzz. gFuzz found all known vulnerabilities and reported no false positives.

On a second test, we compared gFuzz accuracy and performance with other open source web application security tools such as [Rog] with the same set of fuzz strings, and Paros proxy [Com], with its own fuzz strings. In all cases using the same custom web application, with SQL-injection vulnerabilities intentionally inserted, as a target. It is important to mention that both tools perform tests for different types of vulnerabilities (e.g., cross-site scripting) and provides proxying functionalities while gFuzz does not. We chose them since they are useful tools we are familiar with. Table 3 summarizes the results. It compares a run of WebScarab, Paros, gFuzz and a process where the fuzz vectors used by both gFuzz and WebScarab were sent in requests, but responses were not analyzed.

WebScarab only reported a *possible injection*. Paros found the SQL-injection vulnerability we had inserted but also reported false positives. gFuzz reported the concrete injection vulnerability with precise information of the triggering fuzz input and query executed, as mentioned in Section 2.3.

Tested tool	Measured run time	Result
WebScarab	0m 59.321s	Possible injection
Paros Proxy	0m 15.321s	SQL-injection found and false positives reported
gFuzz	1m 23.279s	SQL-injection found and warnings informed
Stress test	0m 12.727s	No Vulnerability found

Table 3: Run time comparison observed between different web fuzzing tools.

Although gFuzz's run time is .5 times higher than WebScarab's and 5 times higher than Paros', we notice that this is the first version of a prototype and has much space for improvement. On the other hand, it is more accurate.

As said in Section 2.3, gFuzz can be extended to test for other types of injection vulnerabilities, such as cross-site scripting, but this would require additional support from the instrumented execution environment (hopefully soon to come). A better fuzzer/crawler can be included in default code so it can be more reliable and easier to use without modifications, this would also allow to attack more complex web applications.

References

[BK04] S. Boyd and A. Keromytis. Sqlrand: Preventing sql injection attacks. In *"Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference. Volume 3089 of Lecture Notes in Computer Science., Springer-Verlag (2004) 292–304."*, 2004.

[BM83] David L Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM Syst. J.*, 22(3):229–245, 1983.

[Com] Chinotec Technologies Company. Paros proxy - web application security. URL: `http://www.parosproxy.org/`.

[Cor07] Core Security Technologies. Core grasp for PHP, 2007. URL: `http://grasp.coresecurity.com/`.

[CW08] Briand Chess and Jacob West. Learn to stop fuzzing and find more bugs. In *RSA Conference, San Francisco, USA.*, 2008. URL: `http://www.rsaconference.com`.

[FGW07] Ariel Futoransky, Ezequiel D. Gutesman, and Ariel Waissbein. A dynamic technique for enhancing the security and privacy of web applications. In *Black Hat USA, August 1-2, 2007, Las Vegas, NV. Proceedings*, 2007. URL: `http://www.coresecurity.com/index.php5?module=ContentMod&action=item&id=1884`.

[GKL08] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, 2008.

[HO06] William G. J. Halfond and Alessandro Orso. Preventing sql injection attacks using amnesia. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 795–798, New York, NY, USA, 2006. ACM.

[Hot04] Cameron Hotchkies. Blind SQL injection automation techniques. In *Black Hat USA, Las Vegas, NV. Proceedings*, 2004. URL: `http://www.blackhat.com/html/bh-usa-04/bh-usa-04-speakers.html#hotchkies`.

[HP] HP. HP webinspect software. URL: `http://www.spidynamics.com/products/webinspect/index.html`.

[Hur04] Andrew Hurst. Analysis of perls taint mode, June 2004. URL: `http://hurstdog.org/papers/hurst04taint.pdf`.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW*, pages 40–52, 2004.

[Imm] Immunity, Inc. Spike proxy. URL: `http://www.immunitysec.com/resources-freesoftware.shtml`.

[Imp] Imperva. Imperva securesphere web application firewall. URL: `http://www.imperva.com/products/waf.html`.

[Ins] Insecure.org. Top 10 web vulnerability scanners. URL: `http://sectools.org/web-scanners.html`.

[Jod03] Jody Melbourne and David Jorm. Penetration testing for web applications, 2003. URL: `http://www.governmentsecurity.org/articles/PenetrationTestingforWebApplications.php`.

[LL05] V. Livshits and M. Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, 2005.

[LMLW08] Monica S. Lam, Michael Martin, Benjamin Livshits, and John Whaley. Securing web applications with static and dynamic information flow tracking. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 3–12, New York, NY, USA, 2008. ACM.

[LR91] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *POPL*, pages 93–103, 1991.

[Nic] Nicolas Surribas. Wapiti - web application vulnerability scanner / security auditor. URL: `http://wapiti.sourceforge.net/`.

[NTGG+05] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. In *SEC*, pages 295–308, 2005. URL: `http://dependability.cs.virginia.edu/publications/2005/sec2005.pdf`.

[Opea] Open Web Application Security Project. Blind SQL injection attacks. URL: `http://www.owasp.org/index.php/Blind_SQL_Injection`.

[Opeb] Open Web Application Security Project. OWASP - sql injection. URL: `http://www.owasp.org/index.php/SQL_Injection`.

[Opec] Open Web Application Security Project. OWASP top 10 - the ten most critical web application security vulnerabilities. URL: `http://www.owasp.org/index.php/Top_10_2007`.

[Oped] Open Web Application Security Project. OWASP top 10 2007 - injection flaws. URL: `http://www.owasp.org/index.php/Top_10_2007-A2`.

[Opee] Open Web Application Security Project - OWASP guide v2. Testing for error code. URL: `http://www.owasp.org/index.php/Testing_for_Error_Code`.

[Por] PortSwigger.net. Burp intruder. URL: `http://www.portswigger.net/intruder/`.

[Rog] Rogan Dawes of Aspect Security. OWASP webscarab project. URL: `http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project`.

[RT07a] Fernando Russ and Diego Bartolomé Tiscornia. Agent-oriented sql abuse. Pacific Security Conference (PacSec '07). Tokyo, Japan. November 28-29, 2007, 2007.

[RT07b] Fernando Russ and Diego Bartolomé Tiscornia. Zombie 2.0. Hack.lu '07. Luxembourg. October 19-20, 2007, 2007.

[SGA07] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing. Brute Force Vulnerability Discovery.* Addison Wesley, 2007.

[SSS06] Joel Scambray, Mike Shema, and Caleb Sima. *Hacking Web Applications Exposed. Web Application Security Secrets and Solutions.* McGraw-Hill, California, second edition, 2006.

[Thi05] Peter Thiemann. Grammar-based analysis of string expressions. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 59–70, New York, NY, USA, 2005. ACM.

[Ven06] Wietse Venema. PHP internals mailing list, Dec 2006. URL: `http://www.mail-archive.com/internals@lists.php.net/msg25405.html`.

[web07] webappsec.org. List of incidents for year 2007, 2007. URL: `http://www.webappsec.org/projects/whid/byclass_class_attack_method_value_sql_injection.shtml`.