Escrow: A Large-Scale Web Vulnerability Assessment Tool

Baden Delamore
Cyber Security Lab
University of Waikato
Hamilton, New Zealand
Email: delamore@outlook.com

Ryan K. L. Ko
Cyber Security Lab
University of Waikato
Hamilton, New Zealand
Email: ryan@waikato.ac.nz

***Abstract*—The reliance on Web applications has increased rapidly over the years. At the same time, the quantity and impact of application security vulnerabilities have grown as well. Amongst these vulnerabilities, SQL Injection has been classified as the most common, dangerous and prevalent web application flaw. In this paper, we propose Escrow, a large-scale SQL Injection detection tool with an exploitation module that is light-weight, fast and platform-independent. Escrow uses a custom search implementation together with a static code analysis module to find potential target web applications. Additionally, it provides a simple to use graphical user interface (GUI) to navigate through a vulnerable remote database. Escrow is implementation-agnostic, i.e. it can perform analysis on any web application regardless of the server-side implementation (PHP, ASP, etc.). Using our tool, we discovered that it is indeed possible to identify and exploit at least 100 databases per 100 minutes, without prior knowledge of their underlying implementation. We observed that for each query sent, we can scan and detect dozens of vulnerable web applications in a short space of time, while providing a means for exploitation. Finally, we provide recommendations for developers to defend against SQL injection and emphasise the need for proactive assessment and defensive coding practices.**

I. Introduction

In recent years, a large number of software systems are being ported to the Web, and platforms providing new kinds of services over the Internet are becoming increasingly more popular [1]: e-health, e-commerce, e-government. At the same time, however, such services, which take the form of web applications are subject to attacks by hackers with the objective of gaining unauthorised access to the system, accessing private information, or simply causing a denial of service.

According to the Open Web Application Security Project (OWASP) [2], Structured Query Language (SQL) [3] Injection is presently the most dangerous and prevalent vulnerability pertaining to web applications. Due to the nature of SQL injection, successful attacks can lead to leakage of administrator credentials, credit card information, social security numbers, email data and so forth. Additionally, the Common Weaknesses Enumeration (CWE) [4] has classified SQL Injection as the most widespread and most critical software error at the time of writing this paper. Contrary to popular belief, SQL injection reports to the CWE as a percentage of total reported vulnerabilities have almost doubled from 2013 to 2014 [5], indicating that there is a real need to address the ubiquity of such a vulnerability. In spite of their continuous evolution, vulnerability scanners that scan for SQL injection still have their faults with regards to the high number of undetected vulnerabilities and high percentage of false positives [6]. With that said, we propose to build a large-scale detection and exploitation tool to achieve two goals. First, to identify how widespread SQL Injection is today. And secondly, to illustrate how using search can aid in the discovery for potential vulnerabilities by using simple search parameters. In addition, our proposed tool will also provide a proof of concept for the vulnerabilities it claims to detect, thereby reducing the false positive rate.

The remainder of the paper is organised as follows. In Section II we study existing security tools and compare and contrast their capabilities. In Section III we describe our main contributions to this paper, namely Escrow, a large-scale web vulnerability assessment tool. Section IV compares and evaluates our tool in terms of speed and efficacy. In Section V we outline a number of recommendations about mitigation and defensive techniques to defend against SQL injection. Finally we conclude the paper in Section VI.

II. Comparison / Related Work

Escrow shares commonalities with both scanners and exploitation tools. Two programs that lend themselves to comparison with Escrow, FImap [7] and SQLMap [8], will be examined and contrasted in this section. In many cases, Escrow is technically and/or administratively easier to deploy than other open source or commercially available tools.

Perhaps the best comparisons of Escrow is FImap and SQLMap. FImap is a command line based python tool which scans for Local and Remote File Inclusion (LFI, RFI) vulnerabilities. SQLMap, however, is a program that automates the process of detecting and exploiting SQL injection flaws, also built in python. A detailed analysis of SQLMap is covered in Section IV. Both tools have the capability to use search modules for information gathering on remote sites. This feature is essential in any large-scale vulnerability program. Presently, Escrow does not detect for LFI or RFI, but future versions will implement this capability.

One advantage of Escrow is that it provides a users with scanning and proof of concept modules while still being a lightweight, cross-platform tool that doesn't require installation. Lightweight security tools are small, powerful and flexible enough to be used as permanent elements in security assessment. Escrow is well suited to fill these roles, weighing in at roughly 1200 Kilobytes. Compare this with many commercial

security assessment tools, which require dedicated platforms and user training to deploy in a meaningful way.

Another advantage of Escrow is that it provides a user friendly Graphical User Interface (GUI). The GUI has search modules with pre-defined parameters that make it easy for users to search for areas within a web application that take part in a database query. We believe it is better to generate only those pages that are of interest to us, rather than searching through the entire web application. In addition, Escrow provides a simple to use database traversal module. Once SQL injection has been exploited with Escrow, users need only click on the database of interest to them in order view the corresponding tables and column data.

One powerful feature that Escrow has is the ability to send HTTP requests in parallel. For example, when a search query has returned pages that are of interest to us, we can systematically scan multiple pages for vulnerabilities at the same time. This significantly reduces the time taken to process requests. We cover this more in detail in Section III.

III. Escrow Implementation / Usage

In this section we describe our main contribution for this paper, namely Escrow, a large-scale vulnerability assessment tool with exploitation capability. Built in Java, Escrow's architecture is focused on performance, simplicity and flexibility. The tool is designed to be used by information security specialist, penetration testers, researchers and IT practitioners.

The proposed approach for large-scale SQL injection analysis can be broken down into four phases described below.

Step I - Link Gathering. This phase aims at gathering URLs for further analysis. How this is achieved is by using a search module that can query within Escrow using pre-defined parameters.

Step II - Analysing URLs. After receiving URLs from the search module, we then move onto the analysis phase. This is carried out by sending a HTTP request back to the URL along with a specific query concatenated to the value element of the parameter field. At this point we compare the response source code of the page returned against a list of known database errors.

Step III - Identifying Injection Type. When URLs have been identified as being susceptible to SQL injection, we then send specially crafted requests to the web application to identify the type of SQL injection that might yield information from the database system.

Step IV - Database Exploration. In this phase we examine the data that is extracted from the application database. This is achieved by a GUI-based exploitation module within Escrow that allows simple table, column and database traversal. When required, it is possible to store the database data to disk in HTML format for further analysis.

The implementation and usage details are described in the forthcoming subsections:

A. Search

Our approach for finding vulnerable injection points in a web application is to use search. For this, we have implemented a Google search module that uses the Google API [9], allowing users to systematically send requests directly from Escrow and have the URLs returned as output. In addition, we have constructed a Bing module that is built by directly analysing the source code of an HTTP request sent to the Bing search engine, as illustrated in Figure 1. Our rationale for doing this is based on the limitations of the Google API, and the number of results capped by the API which is 64. Due to the type of requests that are sent to Google's servers, there is a limit of around half a dozen search requests imposed by using the API. As such, for this paper, we have chosen to focus on using the Bing module for sending queries.

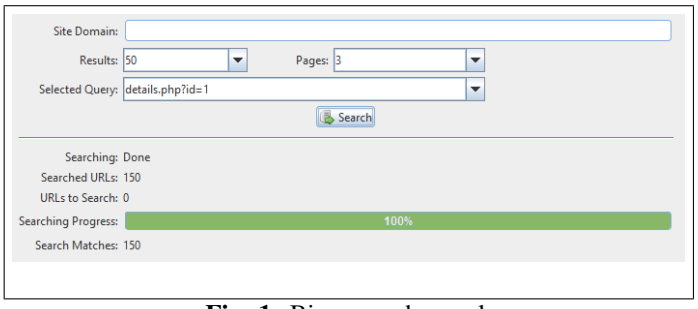

Fig. 1: Bing search panel.

B. Search Heuristics

Finding vulnerable web applications using search requires the use of specific pre-defined queries. Our approach for this is to search specifically for a parameter with a corresponding value in a URL. For example, consider the following URL - http://www.site.com/shop.php?productid=20. In this example, productid is the parameter and its corresponding value 20 are the elements that were interested in. We can observe from this pattern that productid takes part in generation of an SQL query which may take the form of SELECT * FROM products WHERE productid = 20. The parameter element is always fixed; however, an adversary can freely alter the value element. As such, we have chosen to craft our pre-defined query list searching for such parameters which are illustrated in Figure 2. For example, a request containing page.php?id= returns hundreds of pages within the Bing module that have a similar pattern in their URL. Variations for this kind of query do in fact yield URL's that are of high interest to both malicious and benign users. This includes government, education, banking and finance related web applications.

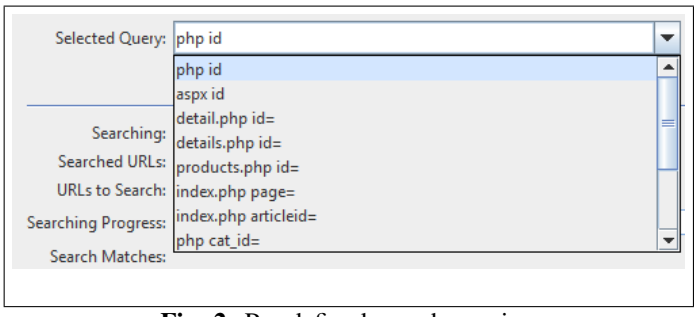

Fig. 2: Pre-defined search queries.

C. Search Modules

In Escrow there are two separate search modules built for the purpose of finding pages that contain input parameter elements, but for the purposes of this paper we will be covering the Bing module. We have implemented a custom Bing class in Escrow that queries the Bing search engine. Our rationale for this is due to the lack of restrictions imposed on using Bings' search service, and to reduce the reliance on an external API. To do this, we first reverse engineered a typical query sent to the search engine. We found that within the URL, one can craft a request by populating the *query* parameter, and specify the maximum number of URLs to retrieve by use of the *count* parameter. Therefore, we can systematically send queries to the Bing search engine without the need for an external API. An illustration of typical search query results is given in Figure 3. Further, if a user wishes to restrict their search to a particular region or country, one can constrain their search by appending the *site* operator within their query. Both Google and Microsoft support the use of such operators.

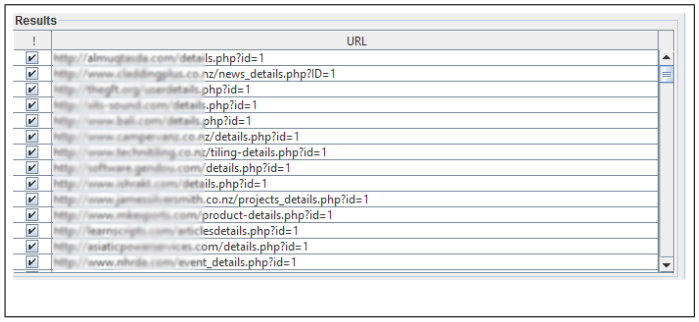

Fig. 3: Bing search results.

D. Code Analysis

To determine whether a web application is susceptible to SQL Injection, first we must identify vulnerable injection points within an application. To do this, we use the search engine modules provided in Escrow and gather a list of potentially vulnerable applications. From here, the URLs retrieved from our search are then passed to a static code analysis module to be scanned for potential errors. Static code analysis in Escrow is performed by analysing the page source code of a web application against a list of database errors (This occurs after sending a request back to the original URL with a modified value element in the parameter field). When an application has been identified as having a potential SQL Injection, we add this to our table of potential vulnerable URLs. Using this method is orders of magnitude faster than exhaustively scanning a site for potential SQL vulnerabilities. This is due to the high number of requests that scanners perform when doing vulnerability assessment. Thus, using pre-defined search parameters eliminates the need for such exhaustive scanning and significantly reduces the number of requests sent to a target server.

E. Multi-Threading

Due to the number of HTTP requests that are sent to remote servers with Escrow, we decided to make use of multi-threading. In doing so, we specify a fixed amount of threads to handle our concurrent outgoing requests. Our rationale for this is due to high wait times when using a single thread. We first identified that a significant amount of time was spent waiting for the response from a remote server, as every outbound request had to be acknowledged and returned before moving on to the next request. This indicated to us that the bottle-neck was at the network layer. Thus, using a multi-threaded approach, and sending our requests in parallel, we significantly reduced the time required to process multiple HTTP requests. The results for this experiment are illustrated in Section IV.

F. Database Errors

Dynamic content that web applications generate is in many cases provided by means of SQL queries to a database. This holds true for a variety of web content technologies including PHP, ASP, ASP.NET, JSP in which web applications are written in. Queries to such web applications are generally benign. That said, it is quite possible that some queries might be mal-formed and executed in the database. When sending a mal-formed query to a web application, and input validation and/or error handling are overlooked, it is possible that the web application may produce an error, which is then echoed back to the user by the application.

Our approach is to scan the web pages source code and compare it against a list of known database errors. These errors are given in Table I. When we find a successful match, we can say to a certain extent that SQL Injection is possible. One advantage of scanning in this way is that we do not require rendering a page in a browser, thus eliminating significant overhead. In addition, many of the errors that we scan for exist only in the page source of the site and not in the rendered version. Therefore, systematically scanning in this way is more effective and eliminates the need for manual analysis of a rendered web page.

TABLE I. DATABASE ERROR LIST

No	Error
1	mysql_num_rows()
2	mysql_fetch_array()
3	FetchRow()
4	GetArray()
5	mysql_numrows()
6	mysql_fetch_object()
7	mysql_fetch_assoc()
8	include()
9	Syntax error
10	mysql_fetch_row()
11	Invalid Querystring
12	error in your SQL syntax
13	Microsoft OLE DB Provider for ODBC Drivers error
14	Server Error in '/' Application

G. Vulnerable Sites

After receiving the search results back from our query, users can choose individually what links in the table they are interested in analysing further, or alternatively, select all the results from the table via the check-all selection in the pop-up menu. From here a user ought to send selected links to the analysis module via the *send to injection crawler* option in the pop-up menu, as illustrated in Figure 4. Within a matter of seconds, courtesy of our multi-threaded HTTP requests, Escrow handles the processing of these sites in parallel and populates the vulnerable links table (Figure 5) for sites that

inadvertently disclose error information. Now that we have established which pages are susceptible to SQL injection, in the next subsection we will demonstrate how from this panel we can traverse the back-end database of the remote server.

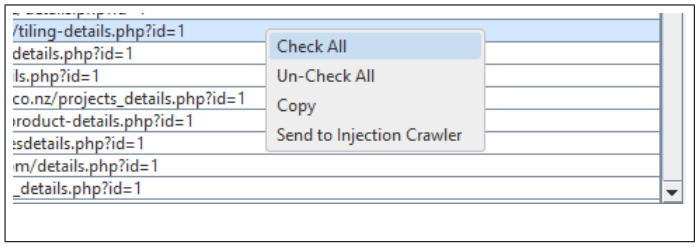

Fig. 4: Pop-up menu.

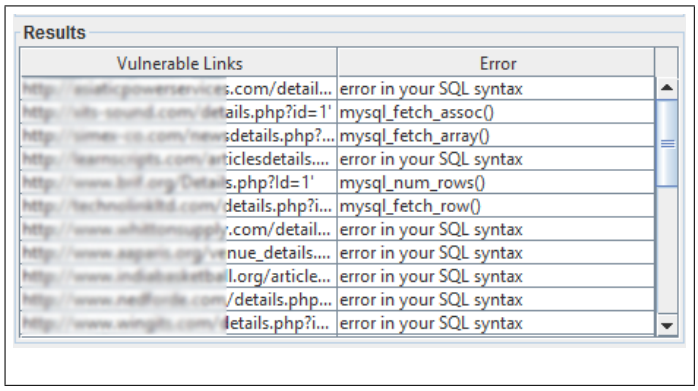

Fig. 5: Vulnerable links table.

H. Database Exploration

After successful identification of a vulnerable injection point within a site, users can then pass the URL on to the exploit module. The exploit module built in Escrow is an extension of jSQLs [10] injection class which supports the following SQL injection types:

- Time Based
- Error Based
- Blind
- Normal

In this section we illustrate how to retrieve data from a database within Escrow. Figures 6 and 7 have been captured when exploiting a local web server so as to not publicly disclose information that does not belong to us. However, links that are extracted from the vulnerable injection panel could simply be loaded into the exploit module in the same way; for security reasons, this will not be demonstrated in this paper. Instead, we will be targeting the vulnerable web application WackoPicko [11] on our local machine.

Figure 6 illustrates the database interface provided in Escrow in which users can select databases and their corresponding tables and display them in a readable format. Certainly many cases that we have encountered have an administrator table that consists of admin credentials and their corresponding passwords.

Figure 7 illustrates the *save-as* option provided in Escrow in which users can save database information for later analysis.

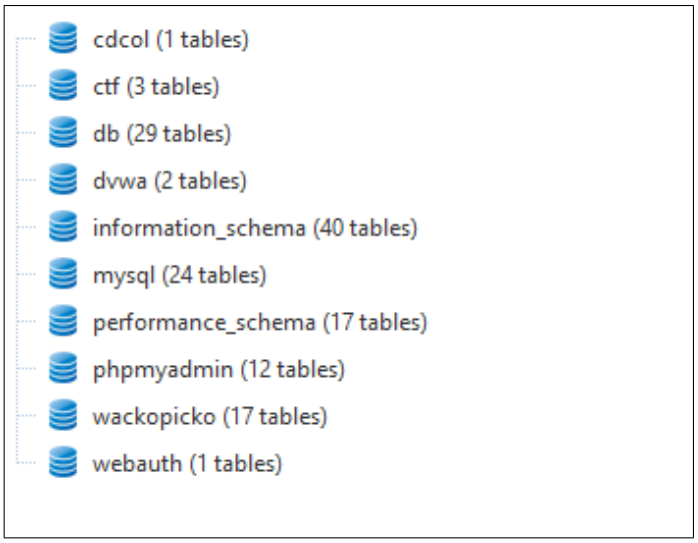

Fig. 6: Database exploration module.

We implemented this function for when providing a proof of concept is required for penetration testers and application security practitioners. The information is saved in HTML format.

id	login	password
1	admin	d033e22ae348aeb5660fc2140aec35850c4da997
2	adamd	c533607326f2b815a7c23701be52989dac8bdbb1
3	admin	d033e22ae348aeb5660fc2140aec35850c4da997
4	adam	0ace61762d02afdf98f793d98c37edf696b675b2
5	bob	42a9037223cdbfe0c49ef0032f0a1f3392af3fe3

Fig. 7: HTML saved data.

As we have seen in this section, it is certainly possible to search for potential input parameters in most web applications. Our software successfully retrieves over 150 results per query using simple search heuristics. Such results can then be passed to a static code analysis module which rapidly scans these sites in quick succession and provides the sites URL and error data back to the GUI. At this point, users can visit the link via the pop-up menu provided, or alternatively pass the link on to an exploitation module where data can be extracted from the remote database and saved to disk.

IV. Escrow Analysis

In this section we discuss how our static code analysis module combined with a multi-threaded approach is proven to be effective in analysing remote sites for SQL Injection vulnerabilities. We compare our tool with the current state of the art, SQLMap, in terms of detection speed and number of requests sent.

To measure the effectiveness of our software we compare it against another tool that uses search in a similar way to Escrow, namely SQLMap. SQLMap provides a search module that makes use of Google's search engine. In order to quantify the performance of our tool we send a single query (asp?id=) and set the maximum number of pages to search at 50. We

measure the time taken to successfully crawl and scan 50 pages for SQL injection vulnerabilities.

TABLE II. SPEED COMPARISON OF ESCROW & SQLMAP

Parameter	Escrow (1 Thread)	Escrow (30 Threads)	SQLMap
Average Time (seconds)	90	6	7700
Number of Pages	50	50	50
Search Query	asp?id=	asp?id=	asp?id=

Unfortunately, due to the high number of requests sent using SQLMaps search module to a remote server, the time taken to analyse one web application for SQL injection is around 154 seconds. A total time of 2 hours and 28 minutes to analyse 50 remote pages.

Table II illustrates the performance difference using multiple threads (in this case we are using 30). The time taken to scan 50 pages is significantly lower compared with using a single thread. In fact, multi-threaded HTTP requests, in this case, is proven to be 15 times faster as opposed to using a single thread and visiting each page in a sequential manner.

In order to quantify how many requests are sent within each program we setup a local testbed with a vulnerable SQL injection point. We then test the capabilities of each tool to first identify whether the site page is susceptible to SQL injection, and secondly, to display the list of databases that reside in the application. Both tools were capable of doing so. We were able to capture packets on the localhost web application by using RawCap [12]. RawCap is a command line network sniffer for Microsoft Windows that uses raw sockets. We then analyse the captured packets using Wireshark [13].

Figure 8 shows that both tools do comparatively well for the number of requests sent to our testbed. SQLMap, however, only requires sending a total of 37 requests to detect for SQL injection and output a list of the databases that reside on the server, back to the program. While Escrow requires a total of 51 requests to achieve the same goal; A difference of 14 requests in total. Our assumption for these results is that when SQLMap detects SQL injection, it then returns back to the main program. However, our tool will check for all injection types before returning back to the program, regardless of whether it has succeeded along the way or not.

In Figure 9 we compare each tool in terms of elapsed time to detect for SQL injection. We note that although SQLMap requires sending less amount of HTTP requests, the time required to identify the vulnerability is significantly higher. SQLMap takes approximately 33 more seconds to detect for a known SQL injection, with a total time of 43 seconds. Escrow, however, only needs 10 seconds for the injection type to be identified. These results show that Escrow far outperforms SQLMap in both the time taken to scan numerous remote sites, and the elapsed time for identifying SQL vulnerabilities.

What we have observed in this section is that both tools we tested are capable of achieving large-scale assessment on remote servers, however, they differ in their implementation details. The main differences with respect to Escrow are given below:

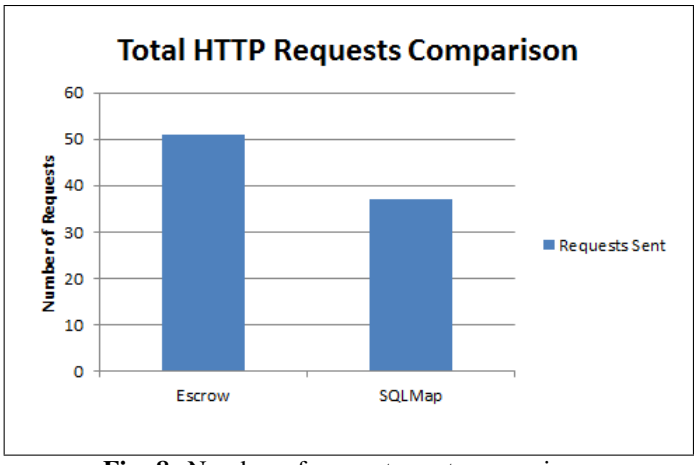

Fig. 8: Number of requests sent comparison.

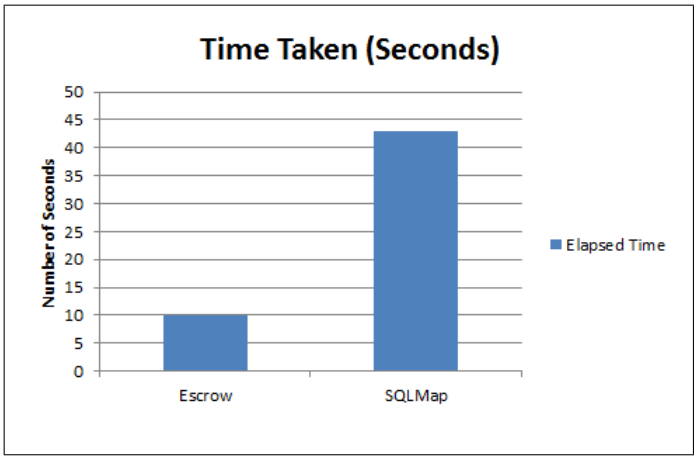

Fig. 9: Time taken to detect SQL injection.

User Interaction: Escrow is completely automatic with its testing implementation. That is to say it does not require any interaction with the tester, who just needs to specify the web application URL. SQLMap on the other hand requires a continuous interaction to better address attacks.

Detection Performance: The time taken to scan 50 remote pages retrieved from SQLMaps search module takes over 2 hours to complete. While Escrows multi-threaded approach can achieve the same result in under 10 seconds.

User Interface: Escrow provides a simple to use GUI which takes care of all the heavy lifting while abstracting away the unnecessary intricacies of the program. SQLMap, however, is a command line tool which shares similar functionality but requires more user interaction.

Search Module: Both tools provide a Google search module. Additionally, Escrow also provides a Bing module that enables users to send the same queries but without the restrictions imposed by using an API - specifically the number of requests retrievable and the type of queries sent.

SQLMap's reliance on the Google API which has imposed restrictions for the type of queries it allows within a certain time frame, is certainly one of the major drawbacks of the tool. That said, when passed a vulnerable URL, SQLMap requires less amount of HTTP requests compared with Escrow.

However, the difference in this case is marginal. Escrow on the other hand is equipped with two search modules and predefined heuristics which makes it a simple to use tool and gives users the option for which search engine they want to use. Moreover, the time taken to detect for SQL injection in Escrow on a large scale is orders of magnitude faster than that of the current state of the art. This approach, coupled with a multi-threaded static code analysis module, highlights the speed and effectiveness of Escrow compared with that of its predecessors.

V. Recommendations

Researchers have proposed a wide range of techniques to address the problem of SQL injection [6][14][15]. These techniques range from development best practices to fully automated frameworks for detection and prevention of SQL injection attacks.

The root cause of SQL injection vulnerabilities is insufficient input validation. In this section we give a brief overview of some of the defensive coding practices proposed in the literature for SQL injection prevention.

- *Input type Checking*: SQL injection attacks can be performed by injecting commands into either a string or a numeric parameter. Therefore, we propose that developers should reject input that isn't consistent with the data type they're expecting. For example, in the case of numeric input, developers ought to reject any input that are not digits. This principle also applies for string data.
- *Input Encoding*: Certainly many injection attempts are often executed through the use of meta characters that trick the SQL parser into interpreting user input as SQL tokens. It is possible to restrict the use of meta characters usage, however, doing so would result in non malicious users ability to specify input that contains such characters. A better approach is to use functions that encode a string in such a way that they are interpreted by the database as normal characters.
- *Positive Pattern Matching*: Developers can validate their input using a white-list. This approach is labeled *positive validation*. Different from *negative validation* which filters out known bad input, positive validation allows only input that is explicitly defined as valid.

In addition to the defensive coding practices listed above, we recommend users employ proactive assessment, including traditional black box testing on their servers. This generally involves crawling the entire application and exhaustively scanning for all points where SQL statements are executed. In conjunction with black box testing, we also recommend using search heuristics, like the ones described in this paper, to efficiently test areas of their web application that search engines have previously crawled.

VI. Conclusion

In this paper we assessed two vulnerability scanners and exploitation tools and gave a brief overview of their capabilities for detecting and exploiting SQL injection. Our observations of the gaps in these tools motivated us to develop Escrow - a large-scale vulnerability assessment tool. We observed that for each query sent within our software, we were able to identify dozens of web applications susceptible to SQL injection including government, banking and education TLDs (Top Level Domains). Using this approach we can say with confidence, it is indeed possible for adversaries to discover and exploit at least 100 databases per 100 minutes. To our best knowledge, this is the first GUI-based large-scale SQL injection detection and exploitation tool available at the time of writing this paper. Finally, we state our recommendations for improving web application security and emphasise the need for defensive coding and proactive assessment.

Future work for this tool aims at building Cross Site Scripting (XSS), Local File Inclusion (LFI) and Remote File Inclusion (RFI) detection modules. Cross Site Scripting, at the time of writing this paper, is currently the most reported vulnerability according the MITRE CWE.

VII. Acknowledgments

The authors would like to thank Raja N Akram for his comments on this paper.

References

[1] A. Ciampa, C. A. Visaggio, and M. Di Penta, "A Heuristic-based Approach for Detecting SQL-injection Vulnerabilities in Web Applications," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*, ser. SESS '10. New York, NY, USA: ACM, 2010, pp. 43–49. [Online]. Available: http://doi.acm.org/10.1145/1809100.1809107

[2] "OWASP - Open Web Application Security Project." [Online]. Available: https://www.owasp.org/index.php/

[3] "Structured Query Language - SQL."

[4] "Common weaknesses enumeration." [Online]. Available: https://cwe.mitre.org/

[5] "2014 sql injection statistics." [Online]. Available: http://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&cwe_id=CWE-89/

[6] S. Roy, A. K. Singh, and A. S. Sairam, "Detecting and defeating SQL injection attacks," *International Journal of Information and Electronics Engineering*, vol. 1, no. 1, pp. 38–46, 2011.

[7] "Fimap." [Online]. Available: https://code.google.com/p/fimap/

[8] "Sqlmap." [Online]. Available: http://sqlmap.org/

[9] "Googleapi." [Online]. Available: https://code.google.com/p/google-api-java-client/

[10] "jsql injection." [Online]. Available: http://jsql-injection.googlecode.com/

[11] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: An analysis of black-box web vulnerability scanners," in *Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 111–131. [Online]. Available: http://dl.acm.org/citation.cfm?id=1884848.1884858

[12] "Rawcap network sniffer." [Online]. Available: http://www.netresec.com/?page=RawCap

[13] "Wireshark - a packet capture and analysis tool." [Online]. Available: http://www.wireshark.org/

[14] S. Boyd and A. Keromytis, "Sqlrand: Preventing sql injection attacks," in *Applied Cryptography and Network Security*, ser. Lecture Notes in Computer Science, M. Jakobsson, M. Yung, and J. Zhou, Eds. Springer Berlin Heidelberg, 2004, vol. 3089, pp. 292–302. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24852-1_21

[15] W. G. J. Halfond, J. Viegas, and R. Orso, "Abstract a classification of sql injection attacks and countermeasures."