Web-Application Vulnerability Scanners:

An Analysis, Exploration and Development

Joseph Avanzato, Ashutosh Bhave

Rochester Institute of Technology

Professor Rob Olson

CSEC 731 – Web Server and Application Audits

Abstract

This project consists of a comparison between ten separate web-application scanners which are utilized in an automated fashion to target three separate web-applications which exist as part of the OWASP Broken Web Application project (Druin, 2011). This included scans against ***Mutillidae II*¸ *Bricks*** and ***ZAPWAVE***. These applications are intentionally vulnerable to various types of attacks and are utilized in order to understand the efficiency and performance differences among the various scanners utilized throughout this work. Additionally, we developed a custom basic vulnerability scanner for web applications which seeks to detect Reflected XSS, Error-based MySQL injection and Local File Inclusion vulnerabilities. Basic data analysis is performed over the collected data in order to determine a weighted vulnerability per minute rate which may help to allow organizations to determine the most cost effective application scanner to use for their specific needs. Developing a custom vulnerability scanner helped give our team insight into the design and implementation phase of such a utility as well as the complexity and difficulty of writing efficient code for such a purpose.

Introduction

The increasing pervasiveness of complex attacks on modern web applications is of great concern to security researchers and developers seeking to harden their public-facing applications. As time marches forward, the complexity of threat actors as well as potential attack surfaces has only increased; this has the potential to lead towards even more nefarious and malicious eventualities should web-based applications simply ignore defensive security tactics and established best practices. Understanding how and through what mechanisms an application is vulnerable is of paramount importance when attempting to design defensive practices seeking to mitigate potential risks. The knowledge a vulnerability scanner can impart to security professionals can greatly assist administrator's and developer's ability to harden defenses focused on web services or applications as well as grant insight into where and how their applications are failing to be properly protected.

This work seeks to explore modern, mostly open source, web application vulnerability scanners in order to compare and contrast performance and efficiency. Additionally, these scanners will be evaluated on their ability to detect 'cutting-edge' attacks which may be relatively new or simply more obscure than other, more well-known types of malicious activity. The selection of scanners utilized will attempt to represent a broad portion of those available to average users but may also include those focused on enterprise-organization usage. No paid tools will be utilized in this work but trial versions may be implemented on a case by case basis depending upon the popularity.

The overall goal of this paper is intended as an analysis of certain available vulnerability scanners as well as the creation of a simple utility designed to detect exploits which may be overlooked by certain other utilities. Our analysis will encompass the usage of various tools against a pre-determined set of web applications contained within the OWASP BWA as a means to gather metrics related to detection percentage for known vulnerabilities and stack utility metrics against one another in order to determine which tools perform the best. Additionally, we will attempt to determine which vulnerabilities are commonly not picked up by said tools and then design a script which will test for these specific vulnerabilities.

<u>**Background**</u>

Modern ‘vulnerability scanners’ range in scope from full hardware analysis to HTTP form fuzzing to brute-force techniques designed to query web-servers for potentially hidden content or exposed vulnerabilities (Fonseca, Vieira, Madeira; 2007). In particular, the usage of vulnerability scanners as a means to detect exploits within web-based services or applications is a field which has seen enormous growth over the past decade. New tools are introduced at a rapid rate with many seeking to encompass as many features as possible rather than focusing on a particular vulnerability domain. As such, modern enterprises are moving towards software which functions as ‘all-in-one’ with respect to scanning, analysis, mitigation recommendations and auditing features which allow penetration testers to efficiently perform wide-scope assessments of enterprise assets. Some of the more popular scanners and frameworks in usage around the world today include *Nessus*, *Nexpose*, *OpenVAS*, *ZAP*, *Vega*, *W3af*, *Skipfish*, and *Arachni* to name only a few from the large selection of available utilities (Rapid7 Toolkit, 2012; Rogers, 2011; Developers, 2012; Makino, Klyuev, 2015; Bau, Bursztein, Gupta et al, 2010).

Not all vulnerability scanners are made equal; many lack full-feature sets or detection capabilities that more full-fledged enterprise-oriented utilities encompass. This work will examine a varied selection of mostly open-source ‘black-box’ styled scanners; some of these utilities exist at a very basic command-line level and others implement full-scale graphical user interfaces with feature-rich functionalities not as easily possible in Command-Line-Interface tools. As web applications have become more complex, it has become necessary for penetration testers to develop more sophisticated applications; this has led to an increase in encompassed functionality and production of larger tools containing varied tool-kits necessary for the modern security researcher. Understanding the mechanisms through which these utilities conduct vulnerability detection is important to glean how they are different and to understanding the strengths and weaknesses of a particular piece of software. Additionally, scanners may include similar functionality in the front-end but perhaps utilize different implementation sources or techniques on the back-end, leading to differences in ability and performance. Understanding the differences, similarities, strengths and weaknesses of these utilities is critical in being able to assess and evaluate the results they produce.

Typical Functionality for Web-App Vulnerability Scanners

Application scanners should attempt to conform with the Web Application Security Consortium's (WASC) *Web Application Security Scanner Evaluation Criteria* which specifies the different features and mechanisms a well-designed scanner should possess (WASC, 2009). In terms of transport protocol support, HTTP1.1/1.0, SSL, TLS and other HTTP features such as Keep-Alive, Compression and User Agents should be present and configurable in a modern scanner in order to support the full variety of web servers and applications existing in the wild (WASC, 2009). Additionally, proxy support should be fully implemented for the mentioned HTTP versions as well as for Socks 4/5 and auto-configuration abilities to properly direct web-based requests (WASC, 2009).

In regards to authentication scanning support, a variety of schemes should be testable including Basic, Digest, HTML Forms and Negotiation, Single Sign On mechanisms such as OpenID as well as SSL certificates or custom designs built within HTTP (WASC, 2009). Testing authentication can become particularly tricky for application scanners but at the most basic, the previously listed mechanisms should have some sort of implemented analysis ability within an examined scanner. Testing authentication for a web application is particularly important because flawed or weak designs and implementations can lead to catastrophic consequences with respect to the potential risks and impacts of unauthorized users becoming maliciously authenticated.

Additionally, it is typically understood that modern scanners should attempt to maintain an 'alive' or active session with the application or service being scanned for vulnerabilities. This is in order to facilitate both session analysis and directory crawling mechanisms which may not function properly when using 'one-off' HTTP requests to attempt detection (WASC, 2009). Management capabilities should typically include the handling of session management such as expiration or refresh as well as receiving and processing HTTP Cookies, Parameters and URL-based tokens (WASC, 2009). Additionally, capable scanners should be able to examine and analyze the session refresh policy of a particular application or web-server.

Further, modern vulnerability scanners will typically include some form of 'web-spider' or crawling component which enables the parsing and discovery of additional URI/pages/files located on a particular web-server (WASC, 2009). This is important because advanced web pages typically consist of a vast amount of resources linked together rather than existing within one location or file. Being able to 'crawl' through pages in an automatic fashion assists penetration testers by automating the manual discovery of additional pages and assists in the detection of potentially hidden vulnerabilities. At a minimum, a crawler should be able to start at a given URL, hostname, IP or other resource and take input as to potentially excluded files/domains/types; finally it should be able to ID new hosts, discover form submissions, detect errors and other HTTP status codes, support the usage of cookies and AJAX and also support proper redirection for additional analysis (WASC, 2009).

In congruence with fully-implemented crawling abilities, most scanners should have the capability to receive and properly parse various types of web-content, including but not limited to HTML, JavaScript, XML, CSS and numerous other languages typically encountered in real-world web-applications (WASC, 2009). This may include the handling of various encoding mechanisms in addition to separating static from dynamic content. In regards to user configuration, control and reporting capabilities, features such as exclusions, scan customization including time, number and status and report generation should be robust and able to fit the needs of the user in question (WASC, 2009). Together, these features should comprise the core abilities of modern web application vulnerability scanners and represent an ideal development design.

As stated previously, not all scanners are created equal. As such, testing them against one another as will be performed in this project is important to understand where one utility may have weaknesses compared to another. Previous work such as (Bau, 2010) help to demonstrate this fact in assessing particular scanners against one another and how some work more efficiently than others. Core features of studied scanners will be assessed and compared in the results section.

Strengths and Weaknesses

Typically, application vulnerability scanners will offer various operating techniques to researchers in order to test different facets of a web application. As one example, the Zed Attack Proxy (Bennetts, Neumann; 2013) offers penetration testers four separate testing modes; these include Safe, Protected, Standard and ATTACK, with each mode defining a separate scope for the scanner. In Safe Mode, any potentially dangerous actions are automatically disallowed with passive listening and analysis of HTTP requests and responses existing as the main analysis utility. Protected Mode allows for the simulation of *potentially* dangerous vulnerability, with harmful actions only allowed in URL modification rather than direct attacks or requests. Standard Mode allows the user to perform any action required while ATTACK mode allows for automated scanning of newly discovered hosts which exist within the defined scope as soon as they are discovered (Bennetts, 2013). Together, these offer the experienced tester the ability to control exactly how much damage may be done by a particular test and remain in scope for the assigned project.

More complex utilities such as *Nessus* or *Nexpose* offer greatly enhanced feature sets compared to open-source software such as *ZAP*, *W3af* or *Vega* (Toolkit, 2012; Rogers, 2011; Suteva, Zlatkovski, Mileva, 2013). Geared towards major enterprises, scanners such as these encompass a vast amount of capabilities not possible to break down into an easily digestible format. In particular, *Nessus* comes equipped with multiple scanning templates which can be further customized and are designed to quickly deploy focused scans for problems such as Shell-shock Detection or vulnerable services as well as more advanced hardware extrapolation and analysis capabilities not present in *ZAP* or smaller utilities. This makes it an ideal tool for full scanning results but perhaps not the best software for web-application vulnerabilities in particular, contrasted with *Nessus* which seems to better integrate with web-application traffic for a more focused analysis when necessary (Fonseca, 2007; Rogers, 2011). Enterprise applications such as the mentioned *Nessus* and *Nexpose* typically represent the ‘best of the best’ in regards to vulnerability scanners and as such this project will focus on those typically used by smaller businesses or developers.

With respect to that, open-source scanners are by no means worse off in terms of detection capabilities for most common application vulnerabilities (Suteva, Zlatkovski, Mileva, 2013; Saeed, Elgabar, 2014). In fact, many come fully equipped to handle the same vulnerabilities as enterprise software but typically are far less robust when it comes to configuration, report generation or general usage technique availability. Some of the more common and efficient open-source scanners available to the public include *Grabber, Vega, ZAP, W3af, Skipfish* and *Arachni* (Suteva, 2013; Saeed, Elgabar, 2014). A sub-selection of certain open-source scanners will be utilized in determining comparison metrics later on in this work.

Unfortunately, many web application scanners on the market today suffer from an inability to detect vulnerabilities which require advanced manual interaction or multi-step complex transactions in order to discover. One example of this is the existence of second-order XSS attacks which are only typically revealed through some form of source-code analysis as demonstrated in (Dahse, Holtz, 2014). In particular, open source scanners which may not have the same development resources as enterprise-level software tends to lag behind with regards to 'cutting-edge' detection or implementation techniques.

With respect to common weaknesses seen in web-applications, this can depend on the specific sort of service being provided. Typically these will include Cross Site Scripting (XSS), SQL Injection, Directory Traversal/Path Disclosures, Denial of Services, Arbitrary Code Execution or Cross Site Request Forgeries (Suteva, 2013; Saeed, Elgabar, 2014; Daud, 2014). A selection of these will be utilized in determining various scanners detection capabilities with respect to each type of exploit. Results from these tools will be compared and assessed in order to make determinations as to tool efficiency, capability and adaptability when evaluating potentially dynamic web-applications and content generated from client requests. This analysis should provide interesting results relating to open-source vulnerability scanners and their potential usefulness in real-world applications.

Of particular initial interest to our team initially was a form of Cross Site Scripting (XSS) known as 'persistent', 'stored' or 'second-order' (Bau, 2010; Halfond, 2006). In this style of attack, a threat actor will identify a vulnerable web-application first by examining how a user may provide input to the site; this may consist of profiles, file management, settings, forums, server logs or other objects which in some form rely on input provided by the end-user. Further, associated HTML may require analysis to determine formation and storage location for various data objects, with separate injection techniques required depending upon the nature of the data storage. Upon identification of vulnerable data stores and un-sanitized user input uploads, an attacker may be able to inject malicious scripts which remain persistent on the web-server containing the web-application. As such, it is possible to effect visitors to the site who have not previously interacted with the attacker due to the web-application potentially serving up the previously specified malicious script to the new end-user.

Methodology

For this project, we utilized a typical LAMP stack configuration for hosting the Damn Vulnerable Web Application (*DVWA*) as well as *Mutillidae II*, *Bricks* and *ZAPWAVE*, all part of the OWASP BWA which was developed and designed for web-application security testing purposes (Druin, 2011; Dewhurst, 2012). These applications serve as vulnerability scanner test-beds by utilizing source-code containing known flaws pertinent to web-applications which may then be used to test the performance of individual application scanners. In particular, various SQL injections, XSS, CSRF, Directory Traversal, File Inclusion and Code Execution vulnerabilities are found throughout these applications which allows for a broad scope when testing various scanners (Druin, 2011; Dewhurst, 2012).

Additionally, multiple open-source scanners were used in an automated fashion in order to determine rate of detection metrics for the vulnerabilities discussed above. The scanners which were utilized in this study included ***Vega, ZAP, Skipfish, Arachni, Nikto, Acunetix, NetSparker, Watobo W3af*** and ***Wapiti.*** Others such as *Nexpose*, *Nessus* and *WebInspect* were also initially included but resource constraints and trial versions limited their usefulness (Suteva, 2013; Saeed, Elgabar, 2014; Toolkit, 2012; Bennetts, 2013). Most of these applications have feature-rich sets which enable security researchers to perform a variety of tasks both through manual and automated analysis mechanisms. In general, we will be utilizing automated analysis techniques provided by these in order to determine how effective their results are when compared to one another in scanning identical installations of vulnerable web-applications.

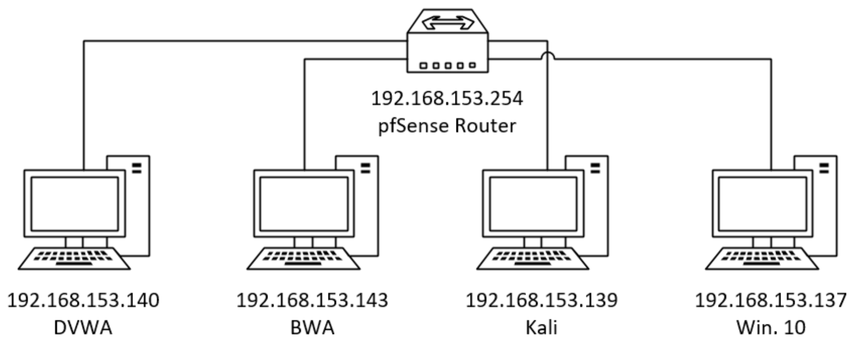
Data related to the detection of known vulnerabilities in each web-application as performed by individual scanners will be analyzed in order to make assessments as to the internal capabilities of each utility, giving insight into how accurate its evaluation of the application in question is. This data will be aggregated and used to make comparisons to other open-source scanner research in order to determine how well our data agrees with that of other security researchers.

Additionally, we sought to attempt the development of a basic utility seeking to implement detection of said vulnerabilities. We initially sought out any potential methods to detect second-order cross site scripting (XSS) attacks, otherwise referred to as 'persistent' or 'stored' XSS attacks (Fonseca, 2007; Halfond, Viegas, Orso. 2006). This is due to their relatively undetected nature in many 'black-box' testing scenarios, making them one of the more commonly missed vulnerabilities in these types of analysis (Makino, 2015; Daud, 2014). Over the course of the project, it was determined that this may be slightly unfeasible and a shift to focus on detecting basic SQL injection, Reflected XSS and Local File Inclusion was instead taken. The developed utility is written in Python 3+ and is extremely limited in scope and capacity when compared to actual vulnerability scanners existing in the wild due to time and resource constraints for this report. Seeking stored-XSS attacks can be extremely difficult due to a lack of investigative feedback and as such many attempts at detecting this sort of maneuver are geared towards code-analysis as seen in (Dahse, Holz, 2014). Attempting to port this functionality to a live application scanner rather than static code analysis was one goal of this work beyond metric gathering for existing open-source vulnerability scanners. Unfortunately this turned out to be more difficult than initially anticipated and as such our focus and scope shifted over the course of the project.

Detecting second-order XSS vulnerabilities on live web applications is problematic if not given access to the source application code or allowed to inspect data storage formatting and conditions. Having some first-hand knowledge of the application in question while testing is often referred to as 'grey-box' testing; this is most likely the type of penetration testing we will hope to achieve in our basic utility. Utilizing partial knowledge about the web-applications in question will help to determine stored-XSS vulnerabilities as well as potentially allow for extrapolation to external applications. We attempted to build a basic analysis engine which may determine available forms and fields for a specific web application page which may be coupled with a fuzzing engine seeking to provide randomized inputs to identified sections. Attempts will be made to perform a more sophisticated analysis in regards to assessing the state of data storage and returns or uniquely fuzzing based upon environmental factors but these may be limited due to known constraints and developer ability.

In general, the above process describes two key objectives for this work; the first being gathering vulnerability scanner metrics in regards to detection accuracy for the previously mentioned scanners and web-applications. The second being the design and development of a basic vulnerability scanner seeking to expose basic reflected XSS, MySQL injections and Local File Inclusions. Using this two-pronged approach, it was possible to determine how accurate a set of 'black-box' scanners are when compared to one another and using basic data analysis to uncover a 'vulnerability per minute' metric which was used to compare the results from all scanners.

Specifically, five virtual machines were utilized in this project; one for a PfSense Router, one hosting Kali VM, one hosting DVWA, one hosting the OWASP BWA and a fifth utilizing Windows 10 for applications not available on Linux. Together, these virtual machines formed the basis for our test environments with most scanners being utilized from the Kali virtual machine to perform application scans across the virtual machines hosting target web applications. A basic environment image of our setup is shown below.



Basic Environment Details

In general, each of the previously introduced scanners were run in as automated as possible manner in order to gain relevant and effective results which were used to form data analysis comparisons between the applications. Understanding how different application scanners perform when compared to one another is important in order to allow individuals and enterprises to make informed decisions when it comes to deciding what type of scanner to use for particular assessments. Unfortunately, although many scanners do include the option to include cookies and other data, scanning DVWA with the various applications returned less than favorable results due to a number of factors. Mainly, fuzzing of the logout form, the security cookie and a failure to maintain session states for proper crawling led many of the utilized scanners to fail at performing effective scans of the web-application. This led us to focus on the OWASP BWA set of applications rather than any results gained from DVWA scans due to their varying and unpredictable nature. The commercial scanners handled session state maintenance much better than the open-source ones but since this project contained a main focus on open-source application scanners, we will keep the scope of the project focused on results generated from scans of the OWASP BWA applications.

Results

As mentioned in the methodology, our team has setup environments hosting both the DVWA and OWASP Broken Web Application (BWA) project, with specific testing being performed on *Mutillidae II, Bricks* and *ZAPWAVE* from within the BWA project. The scanners utilized consisted of *Vega, Skipfish, Nikto, Watobo, Wapiti, Arachni, Acunetiux, W3af, ZAP* and *Netsparker.* Unfortunately, our results regarding scans over the Damn Vulnerable Web Application (DVWA) were less than effective for a variety of reasons. As mentioned previously, session-state maintenance, refresh and creation seemed to prove troublesome for most of the open-source scanners and as such results from scanning DVWA among the various software were varying and unpredictable. Instead, we shifted to focus solely on results from scanning the Broken Web Application project as those were much more reliable and did not rely on the passage and maintenance of session states when scanning or performing requests.

Direct results from our various scans and our analysis of the generated data are presented immediately below followed by a brief discussion on our personal experience with each scanner, its strengths, weaknesses and ideas surrounding its usage. Following this, a discussion regarding our creation of a custom vulnerability scanner is introduced along with potential future work, the difficulties in creating such a utility and other general thoughts about the process as a whole.

Data Results and Discussion

The scanners listed previously were used to perform thorough web-application scans against certain web-apps in the previously introduced OWASP Broken Web Application project, namely *Mutillidae II*, *Bricks* and *ZAPWAVE.* The results gathered were categorized and organized by the type of vulnerability, with main attention paid to potentially critical vulnerabilities such as Cross Site Scripting (XSS), SQL Injection, Command Injection, Remote or Local File Inclusion and Arbitrary File Uploading. Other detected vulnerabilities such as passwords over plain-text or Cookies passed without the HTTPONLY header were sorted under an ‘Other’ category while all other low-priority or informational disclosures were categorized within a ‘Low-Priority’ category. This included items such as missing certain headers, blank or unspecified character sets or version information being leaked.

The results presented below represent the generated reports of utilizing the ten application scanners in a black-box fashion against the listed application. The categories are based upon the most detected and critical vulnerabilities for easy comparison between the various application scanner with respect to these specific weaknesses. The ‘time taken’ metric indicates the amount of time a scan took to run to completion, given in minutes, and helps to understand the time differences between the various open source and commercial scanners when related to the number of issues detected among each.

Scanner Name	XSS	SQLi	CMDi	LFI/RFI	Other	Low Priority	Time Taken (Minutes)
Vega	5	6	7	2	39	135	55
Skipfish	17	2	2	9	72	262	36
Nikto	0	3	1	8	37	25	24
Watobo	16	6	0	2	21	65	5
Wapiti	23	15	0	0	0	0	210
Arachni	18	4	2	5	17	22	255

Acunetix	14	11	1	2	37	64	439
ZAP	25	23	1	7	64	362	155
W3af	3	0	2	1	8	5	4
Netsparker	6	1	1	1	26	195	185

Scan Results against *Mutillidae II*

Scanner Name	XSS	SQLi	CMDi	LFI/RFI	Other	Low Priority	Time Taken (Minutes)
Vega	3	4	0	0	16	89	4
Skipfish	3	1	0	0	21	45	1
Nikto	0	0	0	0	3	7	2
Watobo	8	2	0	0	3	20	4
Wapiti	3	1	0	0	2	2	21
Arachni	5	2	0	0	6	34	6
Acunetix	3	2	0	0	5	18	3
ZAP	15	3	1	4	17	102	25
W3af	5	5	0	0	7	18	3
Netsparker	3	2	0	0	16	93	12

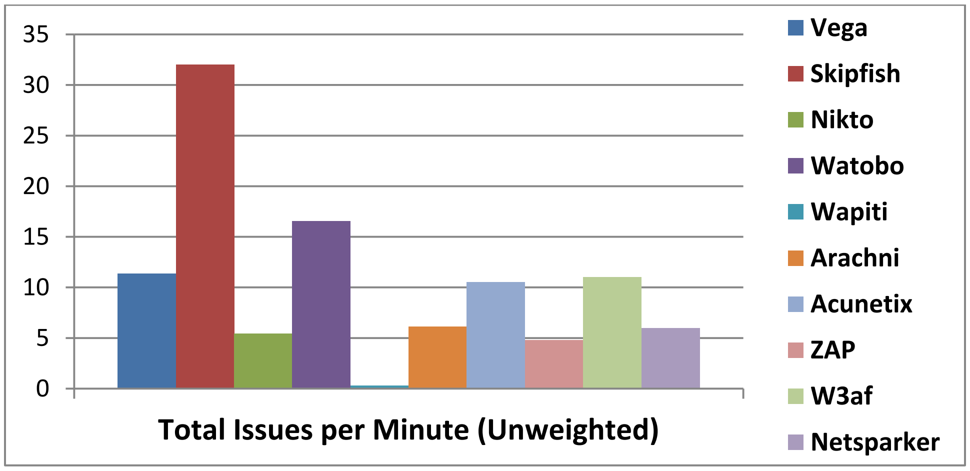
Scan Results against *ZAPWAVE*

Scanner Name	XSS	SQLi	CMDi	LFI/RFI	Other	Low Priority	Time Taken (Minutes)
Vega	12	19	0	1	13	36	32
Skipfish	23	2	1	12	32	248	20
Nikto	2	1	0	0	5	25	4
Watobo	12	14	0	2	24	45	5
Wapiti	13	13	0	0	0	0	74
Arachni	21	11	1	1	43	97	17
Acunetix	16	32	1	7	28	26	5
ZAP	15	43	0	2	54	125	43
W3af	2	1	4	7	17	19	3
Netsparker	12	27	4	4	28	192	37

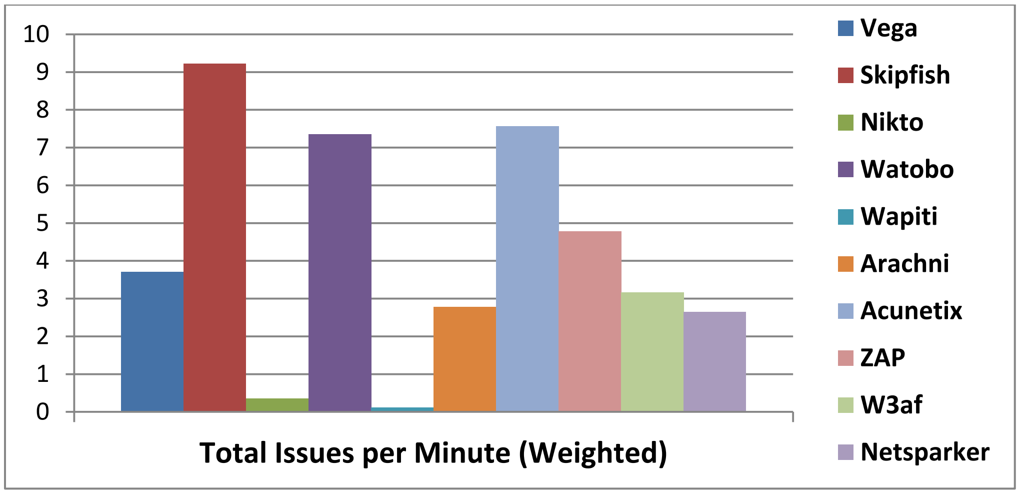
Scan Results against *Bricks*

The results presented above demonstrate the raw output from the various application scanners when used against the web-application listed in the caption. XSS includes all detected Cross Site Scripting injections, SQL includes all detected SQL injections including both blind and error based, CMDi represents all detected potential command injections, LFI/RFI represents any local or remote file inclusion possibility and other includes issues such as cookies transmitted without HTTPONLY, detected directory listings or traversals and other potential vulnerabilities which do not fall under the main four categories. Low-Priority indicates any detected issue which is not necessarily a critical vulnerability but could potentially be utilized by an attacker for further pivoting or knowledge gain such as unspecified or blank character sets, meta-tagging issues or missing headers such as 'X-Frame-Content'. As is immediately obvious, certain scanners tend to perform far better than others with respect to detecting the variety of vulnerabilities presented above, both in terms of detection rates and time taken to perform a thorough application scan.

We then utilized this data in order to derive a 'issues detected per minute' rating using averaged values from the above data. This was performed in order to give some sense of how well the application scanners perform from an efficiency point of view. Large enterprises view time spent as equivalent to money spent and as such having this type of metric available may help to make determinations about which application may be most useful to the organization as a whole. The general results of this analysis are shown immediately below.

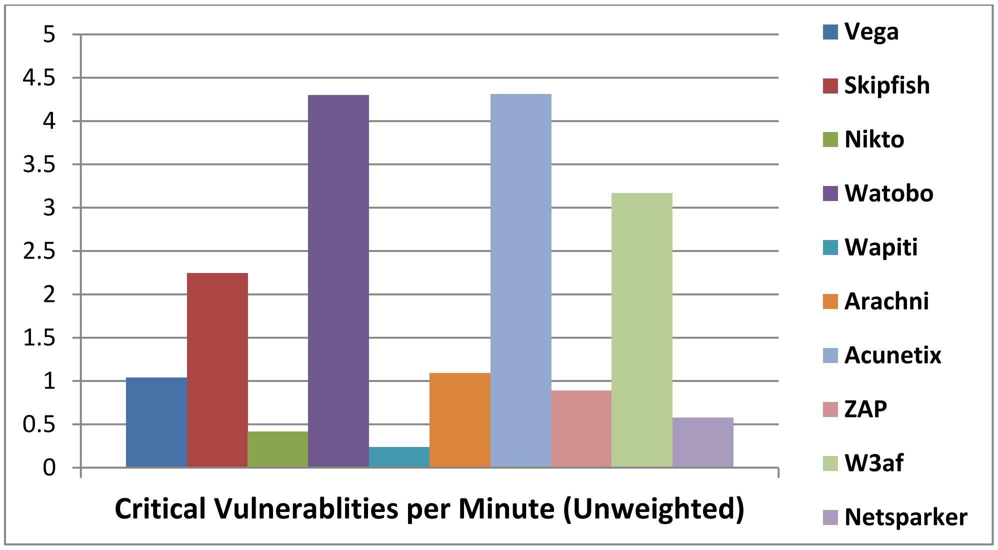


As shown in the graph above, ***Skipfish, Watobo*** and ***Vega*** seem to possess the highest effective ratings when compared in this manner, although ***Acunetix*** and ***W3af*** are in a very similar range. Unfortunately, taking this type of approach could lead to an issue where-in one scanner may detect ten issues in one minute but another might detect 100 issues in ten minutes, leading to an effective rate of ten issues per minute for each application. That number does not take into account the fact that one of the scanners detected a far greater amount of issues overall, making it potentially have a greater effectiveness overall that is not represented in the visualization of the ‘issues per minute’ metric. To correct this type of behavior, we decided to perform a basic weighted analysis based upon the number of total issues a given scanner detected, pulling down the metric for scanners which detect very few issues compared to those that detect great amounts. Each metric was multiplied by a ratio compiled using the scanner which detected the greatest amount of issues with results shown below.

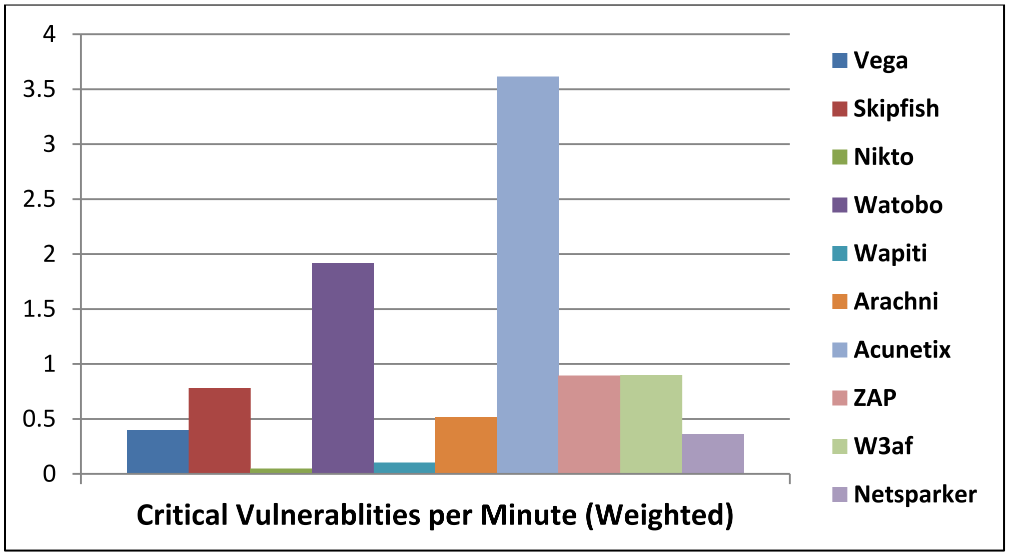


From the weighted values given above, we see that ***Skipfish, Watobo*** and ***Acunetix*** seem to detect the highest amount of issues per minute when weighted across all scanners. We also observe that ***W3af*** and ***Vega*** have fallen significantly lower than ***Acunetix*** after performing this weighted analysis, indicating that ***Acunetix*** has a much higher effective issues per minute detection rate due to detecting a much larger amount, even if it did take quite a bit longer to perform its complete scan. Using this type of weighted analysis is important in order to give researchers a better understanding of how scans are actually performing in the real world.

In addition to a ‘total issues per minute’ rate, we also examined the ‘critical vulnerability per minute’ metric using only issues detected in the first four categories of XSS, SQLi, CMDi and LFI/RFI. Many application scanners will return generated issues for minute problems or things that are not problems at all and as such it is important to understand exactly how these applications stack up when observing only critical vulnerabilities. The initial un-weighted analysis is presented immediately below.



Similarly to before, we can observe from these metrics that ***Watobo, Acunetix*** and ***W3af*** seem to be the obvious leaders when considering only critical vulnerability per minute rates. We performed the same type of weighted analysis over these metrics as for the 'total issues per minute' rates and generated data in order to truly understand whether these three application scanners are the best for this scenario or whether the data was obscured by a low overall time for the scanner operations. The weighted visualization is given immediately below.



The above visualization of the weighted 'critical vulnerabilities per minute' metric for each application scanner makes it clear that ***Acunetix*** far out-performs ***Watob*** as well as all other scanners, an expected outcome due to its existence as an expensive commercial utility. In regards to open source software, it seems that ***Watobo***, ***ZAP*** and ***W3af*** perform the best with ***Watobo*** seemingly performing twice as good as other scanners in the same category. This metric provides an interesting insight into how these applications stack up against one another when performing similar tests on the same hardware against the same web-application configurations.

While not necessarily as direct a metric as 'detection rate' or 'false-positive rate,' this type of weighted analysis on a 'critical vulnerability per minute' measurement can help give insight as to how well application scanners fare against one another when examining the same web application on the same hardware. There are lots of variables involved in this experiment that were not as tightly controlled as would be ideal such as the amount of processing power available at any given time, the memory and network usage of each application scanner, the potential inherent differences in scope, breadth or depth and, among other factors, the fact that certain scanners have a built-in ability to adjust the number of threads and concurrent running processes, leading them to naturally perform better than other utilities.

In conclusion of this data, it seems obvious that certain application scanners tend to perform far better than others, with ***Acunetix***, as expected, far outperforming the open-source scanners when it comes to the efficiency of detecting critical vulnerabilities per minute. While this type of metric but not be universally useful, we believe it may help enterprises to make informed decisions about what type of scanner to utilize with respect to the idea that 'time is money' and saving time by detecting larger amounts of vulnerabilities over perhaps a slightly longer time frame is more useful than detecting small amounts in very small time-frames.

A brief discussion of each vulnerability scanner along with its strengths and weaknesses is presented immediately below. The discussion focuses on our personal experience regarding the strengths and weaknesses of each scanner when considering its proposed feature list. This is meant as a high level overview to each application with specifics given when deemed important as uncovered by our testing methodology. Each scanner is briefly examined and any highlights from our interaction with it our discussed as well as any extremely strong positive or negative aspects which were determined to hinder or help our utilization process.

Web-Application Scanner Discussions

The chart given below gives a brief overview as to the general capabilities of each scanner and helps to quickly summarize the differences between the utilized software. It is not meant as a comprehensive review but rather a generic examination with more details given on the discussion pertinent to each application scanner.

Scanner Name	**Cookie Inclusion**	**Authentication Support**	**Confidence Metrics**	**Vulnerability Prioritization**	**CLI**	**GUI**	**Open-source**	**Report Exporting**
Vega	Yes	Yes	No	Yes	No	Yes	Yes	No
Skipfish	Yes	Yes	No	Yes	Yes	No	Yes	Yes
Nikto	Yes	Yes	No	No	Yes	No	Yes	Yes
Watobo	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Wapiti	Yes	Yes	No	No	Yes	No	Yes	Yes
Arachni	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Acunetix	Yes	Yes	No	Yes	No	Yes	No	Yes
ZAP	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
W3af	Yes	Yes	No	No	Yes	No	Yes	Yes
Netsparker	Yes	Yes	Yes	Yes	No	Yes	No	Yes

Skipfish

Skipfish is an intuitive command-line utility developed in an attempt to try to address some common problems with previously existing apiplication scanners including a high rate of performance, a relatively simple ease of use curve and relatively well-formed security checks which seek to provide accurate results to its users (Zalewski, Heinen. Roschke; 2011). Unfortunately, a lack of updates or maintenance in recent years has meant that it will potentially begin to fall to the wayside in favor of more modern or maintained utilities that are in existence. Having said that, it’s extremely capable reporting and prioritization system makes it easy to understand where and how an application is failing or in learning potential weaknesses that may exist through items such as stored XSS, blind SQL/XML and blind command injection, making it quite robust for scanning even modern web-applications. As a whole, although reporting can generate a mess of files, it is an extremely capable application scanner which should be considered to provide accurate results to its users. Although not recently updated, ***Skipfish*** still seems to remain strong as an application scanner, especially when comparing total issue detection.

Vega

Vega is an automated GUI-based web-application scanner containing typical features seen in such an application such as thorough crawling, dynamic attack payloads and additionally the ability to proxy web traffic through its interface for analysis or modifications (Suteva, et al; 2013). It is also possible to include custom built modules in JavaScript to form a more complete scan or analysis depending upon the organizations requirements. Vulnerabilities are prioritized cleanly in separate categories and highly specific results are given in the built-in report. It also attempts to prevent the reporting of essentially duplicate vulnerabilities while still providing relatively accurate information regarding those that it does detect.

Unfortunately, reports are not able to be exported which is an extremely negative aspect as most other application scanners contain the ability to export detailed and complete reports covering the aspects of the previously performed scan. This detail alone makes it unfeasible for usage in enterprise environments where sharing reports of application scans is critical to improving security. Additionally, some of our tests seemed to indicate a failure to properly crawl to available web pages with no reasonable explanation and a lack of well-formed authentication and cookie support left it unable to properly scan applications such as DVWA. The lack of a Command Line Interface will also deter many users seeking to perform batch or automated scripting use of this application, leaving it to fill a more niche role as software for individual developers rather than enterprise environments.

Nikto

Nikto is a relatively old and no longer maintained or updated vulnerability scanner which tends to focus on the analysis of obtained version information in order to reference vulnerability databases and retrieve appropriate vulnerability information regarding detected versions (Daud, 2014). As such, it is not as dynamic or robust as most of the other application scanners utilized in this project but it may still be useful for certain security researchers as a starting point in order to understand details about the underlying web-server or specific vulnerabilities that may be detected in the available web-application. Unfortunately, it seemed to perform the worst out of any scanner utilized, likely because it is geared more towards analyzing web-servers than specifically web-applications but in either case it is good to get a baseline for poor performing tools as well as well performing tools in order to have a more well-rounded and complete analysis of the gathered data.

W3af

The ***Web Application Attack and Audit Framework (W3af)*** is a general web-application auditing tool which started development in 2007 and was sponsored by Rapid7 of ***Nexpose*** fame in 2010 previous to inclusion in Kali Linux (Doupe, et al; 2010). It is frequently known as the 'Metasploit' of web applications due to its incredibly broad scope and high levels of effectiveness as well as the ease of use it provides in both auditing and attacking web-based applications. It scans for most, if not all, of the common OWASP top 10 vulnerabilities and is developed in Python with both a GUI and CLI available to users. Some of the major included plugins involve an extensive auditing module allowing for custom vulnerability selection, multiple configurable crawling extensions that allow for the introduction of specific scope criteria, integration with *grep* for simple information retrieval functionality and the ability to output reports into a variety of formats such as HTML and CSV.

The main drawback with this application is that it is not included in Kali anymore as of this project and installing or attempting to utilize it can bring with it a variety of headaches that require extensive manual intervention in the form of the downloading and extraction of certain third-party software and the modification of source repository listings to handle dependencies correctly. Additionally, the crawling plugins require very specific scope and domains and have the tendency to enter certain looping conditions which may lead to extremely long hang times as the program attempts to maneuver infinite loops, as discovered through our interaction with the utility.

Wapiti

Wapiti is another completely automated vulnerability scanning platform intended for the black-box analysis of web-applications (Surribas, 2006). It exists only as a CLI utility and has an extremely configurable amount of options available for ease of use and the ability to setup and automate batch scans via remote calls. It is regularly maintained and updated with the latest release occurring as recently as February of 2018. A variety of modules exist within the base version, each performing a separate task and able to be called independently of the others. This makes ***Wapiti*** interesting in that it can be used to test specific vulnerabilities within web applications rather than requiring extensive scanning using all included modules and additionally each module can be customized with various options in order to suit the environment being tested.

<u>**Watobo**</u>

Watobo is a relatively modern take on application scanners, standing for the ***Web Application Security Auditing Toolbox*** and existing as both a passive and active scanner developed in Ruby (Saeed, et al; 2014). It contains an assortment of features that, as the name suggests, are extremely useful for performing security audits on web applications suspected to be vulnerable to a variety of attacks. It solely utilizes a GUI without any CLI but is appended with a detailed console which provides detailed logs on all activities being carried out, providing feedback to the user on queries, results and other useful information occurring in the background. As learned throughout our experience with the software, results seem to be relatively accurate and it also seems to possess a relatively high rate of detection as determined through our previous analysis and comparisons to other application scanners. The main drawback for this tool is the partial reliance on manual intervention in the case of setting up specific scope boundaries for the crawling component.

<u>**Zed Attack Proxy (ZAP)**</u>

The ***Zed Attack Proxy***, otherwise known as ***ZAP***, is a highly modular utility developed and published by OWASP for the purpose of automated vulnerability scanning performed through a variety of mechanisms and features (Bennetts, et al; 2013). As with most of the previously discussed applications, ZAP is able to thoroughly crawl and attack with a vast amount of configuration options related to scope boundaries, threading, encoding or decoding techniques, custom spidering/crawling mechanisms and a variety of other attack options that allow users to handle a wide range of situations and tailor the tool to their specific task. Specifically, certain options that helped when scanning the tested web applications included the configuration of HTTP Sessions, SSL Certificates, global exclusions and manipulating the configuration of active scanning to ensure it was performed as thoroughly as possible. Additionally, reports may be generated in a range of formats such as JSON, XML or HTML as well as the fact that they may be customized to a wide degree with respect to the details included. This tool seemed to work very effectively at catching total issues but with respect to critical vulnerabilities it did not seem to perform as efficiently as certain other open source scanners such as ***Watobo***.

Arachni

Arachni is an open-source scanner utilizing both a web-interface and command-line interaction method (Laskos, 2011). In particular, it is one of the easiest tools to begin using from our perspective and through the interactions our team has had with the various application scanners. It supports a dynamic amount of threads and processes specified by the user in order to perform more efficient scanning should the host system and targeted system be able to support the number of specified requests. The web-interface system is nice, especially for enterprise environments, as it allows for multiple users to be created with a variety of permissions and allowances meaning multiple users can simultaneously utilize it for a variety of scanning and reporting purposes. The main drawback we experienced in utilizing this tool was its extensive crawling time required for complete operations.

Acunetix

Acunetix is one of the only commercial scanners tested in this project and as expected it seemed to perform as one of the most efficient scanners with respect to critical vulnerabilities (Attack, 2014). It is available to be used only through a web-based interface that must be started on the host system and requires logging on in order to utilize all proposed features. The edition we evaluated was a full trial account which had all scanning features available but limited the source requests and specific details about detected vulnerabilities. Additionally, scans took an extremely long time when compared to the other open-source application scanners utilized. Even so, this application seemed to be the most effective when it came to detecting critical vulnerabilities as shown in the data presented in the previous section. This is expected due to its existence as an expensive commercial utility and so the results it generated were not extremely surprising. It seems as if the long scanning, crawling and attacking times were utilized well by the software and helped it to detect an assortment of vulnerabilities which were undetected by other scanners used in the project..

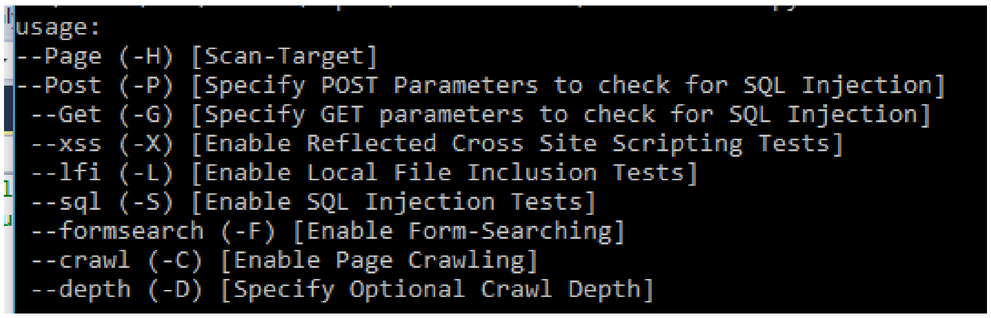
Netsparker

Netsparker is the second and only other commercial application scanner which was tested against the web-applications in this project's scenario (Suteva, et al, 2013). It is used through a local GUI which is packed full of features, configuration and information regarding its capabilities, reporting techniques and allowances for the user to customize almost every aspect of the scan. One of the more interesting features is the ability to perform a dynamic login through the software and have it track all actions, allowing for the login to be replicated at a later time for the purpose of session refresh or management. This can make it easier for security researchers to perform application scans against dynamic, modern web applications which require extensive session requirements. Much like ***Acunetix***, scans with ***Netsparker*** are extremely thorough and extensive depending upon the options which have been initially configured. This means they can potentially take hours upon hours if not longer to run to completion and must be allowed long periods of time in order to gain effective results.

Custom Scanner Development Discussion

In addition to the above results, our team also developed a basic web application vulnerability scanner which seeks to expose MySQL injection points, Reflected Cross Site Scripting (XSS) and Local File Inclusion vulnerabilities. Over the course of development, we have learned much about how vulnerability scanners function, the mechanisms they utilize to perform assessments and the difficulty inherent in building a strong, dynamic and robust application scanner. Much respect has been gained for developers of complex application scanners due to now having gained some internal knowledge of a scanner's business logic.

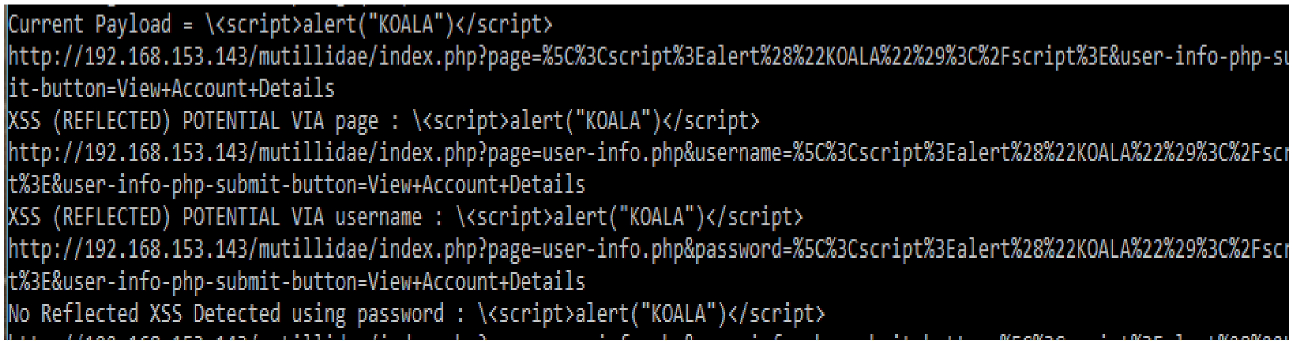
The utility is written in Python 3.6 and utilizes both the HTTP-Requests and BeautifulSoup4 modules to enhance its capabilities (Mitchell, 2014). Requests is implemented in order to easily make dynamically formed GET and POST requests to web-servers and receive the contents without having to manually create and utilize sockets while BeautifulSoup4 is used to simplify the parsing of HTTP responses and extract relevant and pertinent data. Without these it would be much more difficult to perform the required tasks and greatly increase the already 'spaghetti'd' nature of the source code. The scanner features basic crawling functionality which allows for the specification of a base domain and depth level between 1-5, extracting links from the initial response and making additional requests to determine the existence of other pages found on the initial links. This crawling capability is limited in scope by the base domain of the provided URI and will not attempt to crawl or make requests to URIs deemed to not exist within the base domain. This feature attempts to limit the crawler and not allow it to automatically extend to pages which may be linked on the base domain but not necessarily exist within the allowed attack scope. The crawler itself will attempt to identify any HTML forms within the HTTP response in order to inject dynamic payloads into the form parameters and submit the form through the appropriate form action. An example image shown below demonstrates the help page and various commands which are allowed when running the script.



```
usage:
--Page (-H) [Scan-Target]
--Post (-P) [Specify POST Parameters to check for SQL Injection]
 --Get (-G) [Specify GET parameters to check for SQL Injection]
 --xss (-X) [Enable Reflected Cross Site Scripting Tests]
 --lfi (-L) [Enable Local File Inclusion Tests]
 --sql (-S) [Enable SQL Injection Tests]
 --formsearch (-F) [Enable Form-Searching]
 --crawl (-C) [Enable Page Crawling]
 --depth (-D) [Specify Optional Crawl Depth]
```

Basic Arguments used for Script Input

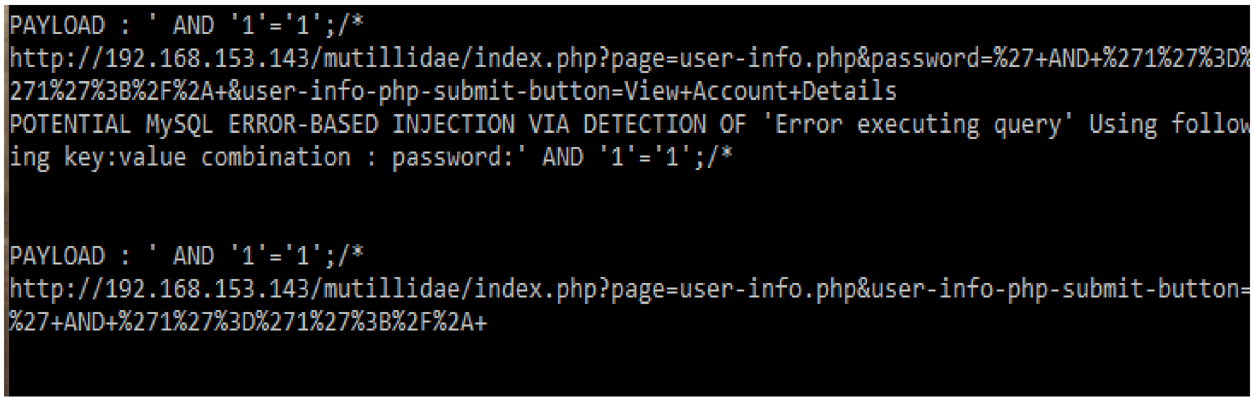
The first basic function is the ability to inject and test for the presence of Reflected XSS via the usage of a static payload list provided in 'xss.txt'. When provided a base URI and with XSS testing enabled via '-X', the script will first load the page and parse the response to determine the format and number of HTML forms which exist in the specified page. Then, for each form, the script will learn all form parameters and the required action and will iterate through the payload list, submitting each payload to each form parameter via the proper form action and parsing the received response to check if the payload exists, indicating the presence of a potential reflected XSS vulnerability. If the payload is detected in the response, a dynamic list is updated and upon script completion this list is written to a report file with the URI that triggered the response and payload given in the report. The reporting system is relatively simple and was not focused upon during development of the script as a whole and could be vastly improved. As it stands, base payloads for XSS are loaded through the included ‘xss.txt’ file and can be modified at will to include whatever payloads the user requires. An assortment of sample payloads representing typical reflected XSS injections are included but using the text list makes it modular and easy to add additional payloads or modify the list to what a user requires. An example screenshot of the script attempting to detect Reflected XSS with partial success through certain form parameters is shown below.



```
Current Payload = \<script>alert("KOALA")</script>
http://192.168.153.143/mutillidae/index.php?page=%5C%3Cscript%3Ealert%28%22KOALA%22%29%3C%2Fscript%3E&user-info-php-su
it-button=View+Account+Details
XSS (REFLECTED) POTENTIAL VIA page : \<script>alert("KOALA")</script>
http://192.168.153.143/mutillidae/index.php?page=user-info.php&username=%5C%3Cscript%3Ealert%28%22KOALA%22%29%3C%2Fscr
t%3E&user-info-php-submit-button=View+Account+Details
XSS (REFLECTED) POTENTIAL VIA username : \<script>alert("KOALA")</script>
http://192.168.153.143/mutillidae/index.php?page=user-info.php&password=%5C%3Cscript%3Ealert%28%22KOALA%22%29%3C%2Fscr
t%3E&user-info-php-submit-button=View+Account+Details
No Reflected XSS Detected using password : \<script>alert("KOALA")</script>
```

Example of Reflected XSS payload success and failure on different form parameters using identical payloads

The second function of the script is to detect the presence of error-based MySQL injection. This is also done through a static payload list given in ‘sql.txt’ but these payloads are dynamically combined with pre-specified ‘special characters’ which are appended as the suffix and prefix to each payload in an iterated fashion. The same type of automatic form scraping and crawling occurs as in the XSS mode with the script attempting to insert and submit the payload via each detected parameter of each detected form and analyzing the results to look for common MySQL error messages. The presence of a detected error indicates the fact that SQL injection may be possible and the URI/payload which triggered the error is appended to a memory list and written to a basic report upon completion of the current scan. Currently only error-based MySQL injection is detected although it would be possible to append other back-end error messages and types of injections through additional work in the source code. Blind-SQL injection would be a logical next step as well as supporting back-end errors and injections for database schemas other than MySQL. This would require slight additional work but is planned for future commits on this project along with a myriad of other potential features. An example image demonstrating partial SQL injection detection via password parameter injection and failure for another is shown below.

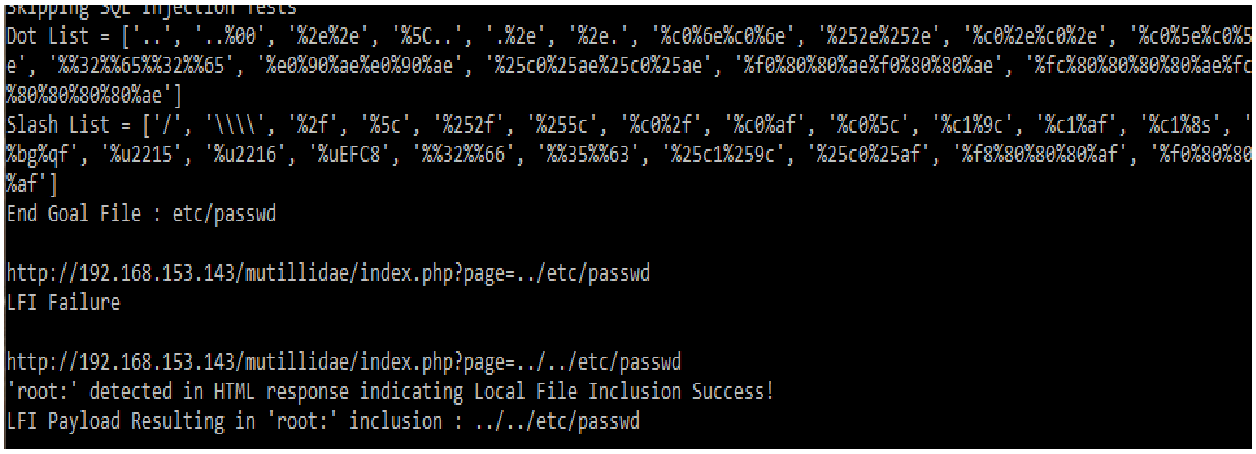


```
PAYLOAD : ' AND '1'='1';/*
http://192.168.153.143/mutillidae/index.php?page=user-info.php&password=%27+AND+%271%27%3D%
271%27%3B%2F%2A+&user-info-php-submit-button=View+Account+Details
POTENTIAL MySQL ERROR-BASED INJECTION VIA DETECTION OF 'Error executing query' Using follow
ing key:value combination : password:' AND '1'='1';/*


PAYLOAD : ' AND '1'='1';/*
http://192.168.153.143/mutillidae/index.php?page=user-info.php&user-info-php-submit-button=
%27+AND+%271%27%3D%271%27%3B%2F%2A+
```

Example SQL Injection success and failure indicated by lack of success message

The last included function in the script is the ability to search for local file inclusion when given a URI to be used in GET requests as well as the pre-pended parameter upon which to test for LFI, such as “http://IP/mutillidae/index.php?page=”. Upon being given this as input with the ‘-L’ flag, the script will attempt to make GET requests using a combination of known dots and slashes given in various encoding styles within the ‘lfidot.txt’ and ‘lfislashe.txt’ included with the script. The payloads within these files are dynamically combined with one another to test a variety of attempts to gain local file inclusion while responses to each attempt are parsed for the string ‘root:’, the presence of which would indicate that the world-readable ‘/etc/passwd’ file was included in the HTTP response, meaning file inclusion was successful. This is demonstrated in the image given below taken during script operations.



```
Skipping SQL Injection Tests
Dot List = ['..', '..%00', '%2e%2e', '%5C..', '.%2e', '%2e.', '%c0%6e%c0%6e', '%252e%252e', '%c0%2e%c0%2e', '%c0%5e%c0%5
e', '%%32%%65%%32%%65', '%e0%90%ae%e0%90%ae', '%25c0%25ae%25c0%25ae', '%f0%80%80%ae%f0%80%80%ae', '%fc%80%80%80%80%ae%fc
%80%80%80%80%ae']
Slash List = ['/', '\\\\', '%2f', '%5c', '%252f', '%255c', '%c0%2f', '%c0%af', '%c0%5c', '%c1%9c', '%c1%af', '%c1%8s', '
%bg%qf', '%u2215', '%u2216', '%uEFC8', '%%32%%66', '%%35%%63', '%25c1%259c', '%25c0%25af', '%f8%80%80%80%af', '%f0%80%80
%af']
End Goal File : etc/passwd

http://192.168.153.143/mutillidae/index.php?page=../etc/passwd
LFI Failure

http://192.168.153.143/mutillidae/index.php?page=../../etc/passwd
'root:' detected in HTML response indicating Local File Inclusion Success!
LFI Payload Resulting in 'root:' inclusion : ../../etc/passwd
```

Example demonstrating initial LFI failure then success using different payloads

Overall, this script performs the basic functions of crawling, parsing and analyzing HTTP responses, automatically detecting HTML forms and attempting basic vulnerability scanning using Reflected XSS, MySQL Error-Based Injection and Local File Inclusion against the detected form parameters. It is nowhere near as robust or advanced as any pre-existing application and mainly serves as an initial starting point in learning how to design and program vulnerability scanners, especially how difficult such a task can be. As 'fun' and interesting as it was to create such a scanner, we have learned the hard way that reinventing the wheel is not easy and creating a dynamic, complex application vulnerability scanner is quite a bit more difficult than it may sound. While we would like to continue work on this we believe a thorough re-working of the built-in algorithms will be necessary in order to improve overall performance and efficiency with respect to the networked programming aspect as well as the dynamic payload generation functions.

Future Work

Future work in this area could encompass many different directions as this project only begins to scratch the surface of effective comparisons and analysis of web-application scanners and their counter-parts. An analysis using 'total issues' and 'critical vulnerabilities per minute' was conducted and attempts were made at weighting the results to favor those scanners which detected greater amounts of issues in longer time periods but a more thorough analysis could be achieved through the usage of additional weighting factors, stricter environment control and more care taken to align the processing power and threading of each application scanner. Additional weighting factors could include the criticality of individual issues detected, a more detailed breakdown of the categorization, whether or not remediation or mitigation strategies were included, suggestions of why the vulnerability exists or other more quantifiable items such as the size of the scanner and the amount of processing power or RAM required to efficiently see scans through to completion against complex web-applications.

Additionally, with respect to the custom vulnerability scanner developed, it would be possible to greatly improve a number of aspects. First and foremost is the performance and efficiency of the created algorithms, which currently are relatively poor and complex, requiring high amounts of time to run through the payload database as well as to generate dynamically formed payload combinations. Improving and reducing algorithm complexity is always a challenge, especially when attempting to tackle a difficult problem such as web-application vulnerability scanning. Other features or items that could use improvement include the robustness or dynamic nature of form-detection, parsing and processing via the inclusion of additional algorithms to handle outlying HTTP forms or other methods of user input. In general, the parsing as a whole could be improved to be ‘less strict’ and attempt to catch and process URIs which may currently be skipped due to rigid parsing features.

Conclusion

This work examined multiple open source and commercial web-application scanners and utilized their features against a subset of web-applications contained within the OWASP Broken Web Application project as a means to gather data and infer performance related metrics between the featured scanners. As demonstrated through the data results section, it seems that certain application scanners such as ***Watobo*** and ***Acunetix*** perform far better with respect to ‘critical vulnerabilities per minute’ detection rates but ***Skipfish*** performs the best when considering all detected issues and not only potentially serious vulnerabilities. This type of application scanner analysis is useful to corporations who wish to retain high efficiency with respect to time spent scanning when considering tests against the same application on the same host platform.

In addition, we attempted to develop a basic and naïve vulnerability scanner which sought out error-based MySQL injections, Reflected Cross Site Scripting (XSS) injections and Local File Inclusion (LFI) vulnerabilities. Throughout this process we learned a great deal on application scanner design as well as gained a deep appreciation for the developers who actively maintain and update modern, efficient and accurate web-application scanners. Our scanner performs relatively poorly, has a potentially high false positive rate, generates messy reports and is not very robust nor dynamic but does offer limited capabilities related to detection of the previously mentioned vulnerabilities and represents a good starting point for learning about programming custom vulnerability scanners. This project helped teach us about the design, application and usage of vulnerability scanners as a whole through the utilization of multiple scanners as well as the creation of our own script attempting to emulate the basic abilities of these applications.

References

[1] Fonseca, J., Vieira, M., & Madeira, H. (2007, December). Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on* (pp. 365-372). IEEE.

[2] Toolkit, W. E. (2012). Rapid7™ Nexpose™ Vulnerability Management and Penetration Testing System V5. 1.

[3] Rogers, R. (Ed.). (2011). Nessus network auditing. Elsevier.

[4] Developers, O. (2012). The Open Vulnerability Assessment System (OpenVAS).

[5] Makino, Y., & Klyuev, V. (2015, September). Evaluation of web vulnerability scanners. In *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), 2015 IEEE 8th International Conference on* (Vol. 1, pp. 399-402). IEEE.

[6] Bau, J., Bursztein, E., Gupta, D., & Mitchell, J. (2010, May). State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on* (pp. 332-345). IEEE.

[7] Web Application Security Consortium (WASC). (2009). Web application security scanner evaluation criteria. *Version*, *1*, 1-26.

[8] Halfond, W. G., Viegas, J., & Orso, A. (2006, March). A classification of SQL-injection attacks and countermeasures. In Proceedings of the IEEE International Symposium on Secure Software Engineering (Vol. 1, pp. 13-15). IEEE.

[9] Bennetts, S., & Neumann, A. (2013). Owasp zed attack proxy project. Retrieved July, 6, 2013.

[10] Dahse, J., & Holz, T. (2014, August). Static Detection of Second-Order Vulnerabilities in Web Applications. In USENIX Security Symposium (pp. 989-1003).

[11] Daud, N. I., Bakar, K. A. A., & Hasan, M. S. M. (2014, August). A case study on web application vulnerability scanning tools. In Science and Information Conference (SAI), 2014 (pp. 595-600). IEEE.

[12] Saeed, F. A. (2014). Using wassec to evaluate commercial web application security scanners. International Journal of Soft Computing and Engineering (IJSCE), 4(1), 177-181.

[13] Doupé, A., Cova, M., & Vigna, G. (2010, July). Why Johnny can’t pentest: An analysis of black-box web vulnerability scanners. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (pp. 111-131). Springer, Berlin, Heidelberg.

[14] Khoury, N., Zavarsky, P., Lindskog, D., & Ruhl, R. (2011, October). An analysis of black-box web application security scanners against stored SQL injection. In Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on (pp. 1095-1101). IEEE.

[15] Druin, J. (2011). OWASP Mutillidae II Web Pen-Test Practice Application. Retrieved June, 25, 2013.

[16] Dewhurst, R. (2012). Damn Vulnerable Web Application (DVWA).

[17] Suteva, N., Zlatkovski, D., & Mileva, A. (2013). Evaluation and testing of several free/open source web vulnerability scanners.

[18] SAEED, F. A., & ELGABAR, E. A. (2014). Assessment of Open Source Web Application Security Scanners. Journal of Theoretical and Applied Information Technology, 61(2), 281-287.

[19] Faghani, M. R., & Saidi, H. (2009, August). Social networks' XSS worms. In Computational Science and Engineering, 2009. CSE'09. International Conference on (Vol. 4, pp. 1137-1141). IEEE.

[20] Zalewski, M., Heinen, N., & Roschke, S. (2011). Skipfish-web application security scanner.

[21] Laskos, T. Z. (2011). Arachni—Web Application Security Scanner Framework, Retrieved Date, 10/2011.

[22] Attack, C. S. S. (2014). Audit your website security with Acunetix Web Vulnerability Scanner. online], https://www. acunetix. com/websitesecurity/cros s-site-scripting/(Accessed: 28 June 2014).

[23] Surribas, N. (2006). Wapiti, web application vulnerability scanner/security auditor. URL: http://wapiti. sourceforge. net.

[24] Mitchell, R. (2015). Web scraping with Python: collecting data from the modern web. " O'Reilly Media, Inc.".