Global Information Assurance Certification Paper

Testing Web Application Security Scanners against a Web 2.0 Vulnerable Web Application

GIAC (GSEC) Gold Certification

Author: Edmund Forster, e336712@gmail.com

Advisor: Sally Vandeven

Accepted: September 26, 2018

Abstract

Web application security scanners are used to perform proactive security testing of web applications. Their effectiveness is far from certain, and few studies have tested them against modern 'Web 2.0' technologies which present significant challenges to scanners. In this study three web application security scanners are tested in 'point-and-shoot' mode against a Web 2.0 vulnerable web application with AJAX and HTML use cases. Significant variations in performance were observed and almost three-quarters of vulnerabilities went undetected. The web application security scanners did not identify Stored XSS, OS Command, Remote File Inclusion, and Integer Overflow vulnerabilities. This study supports the recommendation to combine multiple web application security scanners and use them in conjunction with a specific scanning strategy.

1. Introduction

1.1. The Importance of Web Applications

Critical services across financial, healthcare, defense, energy, and other sectors rely on secure web applications. Web applications enable users to share and manipulate information in a platform-independent manner (Berbiche et al. 2017). They underpin ubiquitous products and services that range from social media to e-commerce, e-government, banking, and many more. Most businesses now have a significant e-commerce component and many rely on web applications to connect with customers. The confidentiality, integrity, and availability of these services can depend on the security of web applications (Ferreira & Kleppe 2011).

The Web Application Security Consortium (WASC) defines web applications as software applications executed by a web server that respond to dynamic requests over HTTP (WASC 2009). They consist of scripts that reside on a web server and interact with databases or other sources of dynamic content (Berbiche et al. 2017). Typical deployments are comprised of a client browser, web server, application server(s), and database server(s) (Berbiche et al. 2017). As the complexity and connectivity of web applications increases, the challenge of securing them grows exponentially (Berbiche et al. 2017). These applications are susceptible to commonly occurring security vulnerabilities including SQL injection, Cross-Site Scripting (XSS), insecure direct object references, Cross-Site Request Forgery (CSRF), security misconfiguration, and failure to restrict URL access (Ferreira & Kleppe 2011). As web applications have grown in complexity it has become increasingly difficult to perform security tests against them. Testing methods have had to evolve to accommodate the diversity of new technologies and the increased attack surfaces they bring.

1.2. Web 2.0

The term ‘Web 2.0’ describes the shift in web services and technologies to the “network as a platform” that spans interconnected devices and delivers software that is continuously updated. This software provides a mash-up of data from multiple sources including individual users (O’Reilly 2007). With the emergence of Web 2.0,

 Author Richard Forster, e336712@gmail.com

the complexity of websites and the resources they draw on increased dramatically, changing the way resources accessible via HTTP are presented and accessed. Micro-services written in node.js and Spring Boot are replacing traditional monolithic applications. Single page applications built with JavaScript (JS) frameworks now enable the creation of feature-rich front ends that are highly modularized. JS is the principal language of the web, which includes node.js running server-side and frameworks such as Angular, Bootstrap, Electron, and React running on the client (OWASP 2017).

With increases in the accessibility and ease of use of web applications come greater attack surfaces (Berbiche et al. 2017). Butkiewicz et al. (2011) identified that the top 20,000 websites loaded an average of 40 resources. Kumar et al. (2017) found that 33% of the top million sites loaded content indirectly through third parties. Many users access resources over which site administrators have little control. Of the sites studied, 87% executed active content from external domains (Kumar et al. 2017). Kumar et al. (2017) describe this as the "tangled attack landscape."

1.3. AJAX

Asynchronous JavaScript and XML (AJAX) is a collection of technologies used by web application developers to create a user experience that mimics non-web applications. AJAX technologies are used to build robust web applications that support data-driven websites, increase usability, interactivity, and speed (OWASP 2013). Due to their positive impact on functionality and ease of use, AJAX technologies have become popular with web application developers (Orloff 2012).

AJAX technologies include the scripting language JS, JS Object Notification (JSON) or XML for the exchange of data. Document Object Model (DOM) for the dynamic display of data enables dynamic representation and interaction (Orloff 2012). HTML (or XHTML) and Cascading Style Sheets (CSS) set the standards for presentation of content to the user. XML and XSLT provide the formats for server-client data exchange and manipulation. The XML HTTP Request facilitates asynchronous data retrieval and ensures that full-page reloads are not necessary each time the user makes

requests (Acunetix 2018). Instead of refreshing the web document after each event, AJAX performs server calls and document updates in the background without necessitating a full reload.

1.4. AJAX Security Issues

There is a common misconception that AJAX applications provide increased security by obscuring server-side scripts. However, XML HTTP uses the same HTTP protocol as non-AJAX applications and is therefore vulnerable to many traditional attacks (Acunetix 2018). AJAX technologies offer an increased attack surface due to the multitude of inputs to be secured. Internal functions of the application can be exposed, and clients may access third-party resources with limited security and encoding mechanisms. Lines between client and server-side can become blurred, and authentication information and sessions require additional protection due to an increase in session management vulnerabilities (OWASP 2013, Acunetix 2018).

In the context of AJAX, there are several common security vulnerabilities (Orloff 2012). Browser-based attacks can exploit security weaknesses in JS. SQL injections can extract valuable data from the server side of the web application. Cross-site scripting (XSS) attacks can exploit browser-side scripts. Also, attackers can compromise the AJAX service bridge that enables mash-ups to draw on third-party websites and data sources. Web applications using AJAX technologies can also be vulnerable to Cross-Site Request Forgery (CSRF).

1.5. Web Application Security Testing

Given the increased attack surface and potential for security vulnerabilities, modern web applications require proactive security testing. Testing should take place throughout the Software Development Life Cycle (SDLC) from development through to deployment and beyond (Gioria 2009, OWASP 2014). The penetrate-and-patch model, which emphasizes penetration testing and responsive software patching, was popular in the 1990s. The tester adopted the role of the attacker with limited insight into the inner workings of the application (Ferreira & Kleppe 2011). Penetration

testing was often considered the primary or only security testing technique (OWASP 2014). As a more holistic view of software development has emerged, the limitations of the penetrate-and-patch approach have become more widely accepted. Penetration testing is now regarded as an assurance method rather than as a primary tool for vulnerability detection (NCSC 2017). Vulnerability studies have shown that attackers can respond quickly to inhibit the usefulness of patch installation (Symantec Threat Reports, 2018) highlighting the need for a more strategic approach to security testing.

Software development brings together a combination of people, process, and technology all of which require testing (OWASP 2014). Comprehensive and high-quality education, proper policies and standards, and the correct implementation of technologies can all impact the security of an application. An effective security-testing regime uses manual inspections and reviews to test the security implications of people, policies, and processes. Threat modeling helps developers to consider the security threats their systems and applications may face. Code review is a way of manually checking the source code for security issues. Finally, penetration testing can be used to test a running application remotely to identify vulnerabilities for remediation (OWASP 2014).

1.6. Web application security scanners

Due to the rapid iteration cycle employed in web application development and maintenance, web application security scanners are used to identify exploitable vulnerabilities (Ferreira & Kleppe 2011, Berbiche et al. 2017). Examples of web application security scanners include OWASP ZAP, Arachni Scanner, Burp Proxy, w3af, and Subgraph Vega. Known as "black-box vulnerability scanners" they are often marketed as point-and-shoot (PaS) penetration testing tools that automate the assessment of web application security (Doupé et al. 2010). These tools can speed up and simplify many routine security tasks (OWASP 2014) and automate the process by performing thousands of otherwise manual tests. They generate vulnerability reports and offer remediation advice for security testers. When used wisely and correctly, they can complement a well-balanced security program (Keary 2013 in OWASP 2014).

Despite their benefits, web application security scanners attract criticism for their limitations. Keary (2013 in OWASP 2014) argues that web application security scanners are both generic and seductive. They are designed to assess applications in general, rather than custom code, and can quickly and easily identify large numbers of security issues. Configuring web application security scanners can be a complex undertaking for unfamiliar users, and there is a significant risk of false positive results (Orozco et al. 2017). Denim Group (2014), an application security firm, argue that security scanners identify roughly 30% of severe vulnerabilities and often fail to detect design flaws. Security scanners tend to offer little or no insight into the internal state of the application. They can provide a useful first look for easily identifiable vulnerabilities but are unable to deliver in-depth, sophisticated assessments (OWASP 2014). While these tools do not make software more secure, they can help to enforce policy and scale the assessment process (Howard 2006 in OWASP 2014). A risk-based approach that considers the system architecture and the attacker's perspective is the best way to deploy web application security scanners (Zhu 2017).

1.7. How effective are web application security scanners?

Many studies have attempted to assess the effectiveness of both open-source and proprietary web application security scanners available to security professionals. It can be difficult to determine the relative efficacy of these tools (Berbiche et al. 2017) due to inconsistent standards and technologies. To help security professionals evaluate web application scanners, the Web Application Security Consortium (WASC) developed the web application security scanner Evaluation Criteria (WASSEC) (WASC 2009). WASSEC was a document created to provide a vendor-neutral to help guide security professionals in selecting the most appropriate tool. It offers a comprehensive list of features to consider when evaluating a web application security scanner and covers factors such as crawling, parsing, session handling, testing, and reporting.

Previous studies have evaluated web application security scanners by testing them against web applications with known security vulnerabilities. Results are typically mixed, with many scanners missing all but fundamental issues. Bau et al. (2010) tested eight commercially available web application security scanners against popular applications. The majority of scanners in this study detected SQLI and Reflected XSS vulnerabilities, but identified other issues at a low rate. Doupé et al. (2010) tested eleven web application security scanners, both commercial and open- source, using a realistic web application. The tools overlooked many classes of vulnerability. Suto (2010) tested three web application security scanners against each vendor's test web applications. Results were highly variable, particularly between 'trained' and 'point-and-shoot' (PaS) mode. Suto noted that training these tools required expert input and was time intensive.

Shelly et al. (2010) assessed the limitations of web application security scanners using both a secure and an insecure custom web application. The aim of the study was to identify scanner weaknesses, improve scanner performance, and reduce false reports. Scanners performed well against simple reflected XSS and SQL injection vulnerabilities but struggled to detect less traditional variants. Multiple false positives resulted from tests against the insecure version of the web application. The study did not explore Web 2.0 technologies such as AJAX. Ferreira and Kleppe (2011) tested web application security scanners against a custom application. The tools did not detect reflected XSS and SQL injection but could detect stored XSS and CSRF. Both of these studies demonstrate that web application security scanners do not detect all vulnerability types in a consistent manner.

A range of studies highlights significant differences in the performance of web application security scanners, as well as the lack of standardized methods for testing them. Alassmi et al. (2012) focused on the detection of stored XSS and identified limitations of various scanners. Saeed (2014) compared thirty two open-source web application security scanners using a selection of the WASSEC criteria. The best tool that met four of six criteria was W3AF. Bakar et al. (2014) tested three web application security scanners (Nessus, Acunetix, and OWASP ZAP). They were assessed using two custom test applications using a two-stage methodology that tested

the speed and accuracy of each tool. Alnabulsi et al. (2014) and Alsmadi et al. (2013) used SNORT to detect SQL attacks on web applications, including the DVWA. Use of custom SNORT rules showed improved detection rates. Fakhreldeen and Eltyeb (2014) assessed open-source scanners according to the OWASP Top 10-2013. Detection performance was compared using the average metric. Makino & Klyvev (2015) compare the OWASP ZAP and Skipfish web application security scanners. The tools were used to evaluate vulnerabilities in the Damn Vulnerable Web Application (DVWA) and Web Application Vulnerability Scanner Evaluation Project (WAVSEP). Reports were analyzed, and tool characteristics were compared. Results were in favor of the OWASP ZAP tool.

Zhu (2017) performed a case study test of a web application called Virtual Application Manager using two web application security scanners. Orozco et al. (2017) used an IDS to obtain the attack signatures of various web application security scanners (OWASP ZAP, Acunetix, HP WebInspect, Arachni Scanner) and compared the requests with the reports generated. Berbiche et al. (2017) assessed the effectiveness of eleven web application security scanners against WAVSEP. Each of the scanners produced different outcomes. All tools performed better on SQLI and XSS than on Local and Remote File Inclusion. With so many products tested under such a range of experimental conditions it is difficult to determine how one scanner performs relative to another. Within this body of research, traditional (non-dynamic) web applications have been evaluated. Much of this work either pre-dates the emergence of Web 2.0 technologies including AJAX.

Web 2.0 technologies present new challenges for web application security scanners. These include traps for crawlers as JS and AJAX employ dynamic links and pages. Web application security scanners were built around HTML name and value pairs, and not the newer formats. Form input validation often requires valid user data. Further research will increase security professionals' knowledge of the capabilities and relevance of web application security scanners in the modern development environment.

2. Method

2.1. Procedure

Three web application security scanners, one open-source, and two proprietary, were tested against a modern web application with known traditional and Web 2.0 (including AJAX-related) vulnerabilities. The web application security scanners were installed on a testing machine and deployed against the Hackazon application. The tools were run in automated 'Point-and-Shoot' (PaS) scan mode with minimal configuration. When necessary, the scanners were configured with test user credentials to permit access to restricted areas of the site. AJAX options were selected where available. The number and nature of the vulnerabilities detected were recorded and analyzed in order to understand how these particular web application scanners perform against a modern Web 2.0 application with AJAX and HTML use cases.

The web application security scanners tested were:

OWASP Zed Attack Proxy (ZAP) (www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project) (OWASP 2018) is a free, open-source penetration testing tool maintained by the Open Web Application Security Project (OWASP). ZAP is an "intercepting proxy" designed for testing web applications.

BurpSuite Pro (https://portswigger.net/burp) **(PortSwigger 2018)** is a fully featured web application scanner and intercepting proxy and claims "coverage of over 100 generic vulnerabilities, such as SQL injection and cross-site scripting (XSS)" and with "great performance against all vulnerabilities in the OWASP top 10". It claims to advance crawling capabilities (including coverage of the latest web technologies such as REST, JSON, AJAX, and SOAP)". This study tested the professional version.

Acunetix Vulnerability Scanner https://www.acunetix.com/vulnerability-scanner/) (proprietary demo version) (Acunetix 2018) claims to detect over 4500 web application vulnerabilities and critical vulnerabilities with 100% accuracy. It offers

"DeepScan Technology – for crawling of AJAX-heavy client-side Single Page Applications (SPAs)" and the "industry's most advanced SQL Injection and Cross-site Scripting testing – includes advanced detection of DOM-based XSS."

2.2. Vulnerable Web Application

The Hackazon vulnerable web application (www.github.com/rapid7/hackazon) created by Dan Kuykendall (Kuykendall 2014) was the target web application for this study. Hackazon offers a 'fake app' test site that replicates an online storefront (Kuykendall 2014). Unlike 'traditional' vulnerable web applications tested in previous studies, it incorporates a realistic e-commerce workflow as well as frameworks such as the Google Web Toolkit and JSON. 'Traditional' vulnerable web applications such as Web Goat (published in 2002 and written in Java) and the Damn Vulnerable Web Application (published in 2008 and composed in PHP) are valuable teaching tools and can effectively test Web 1.0 scanners. However, they do not pose the challenges represented by Web 2.0.

Hackazon enables the user to configure the application to customize the vulnerability landscape. It therefore reduces the risk that scanners have 'pre-learned' the vulnerabilities. Hackazon includes (Kuykendall 2014): both AJAX and standard HTML use cases; AJAX interfaces using RESTful backends, mostly XML and JSON with portions using GWT; web services for mobile-client; Flash and AMF support for entering coupon codes; and strict workflow sequences with vulnerabilities. The Hackazon frontpage (below) includes branding, product descriptions and images, and links to typical e-commerce sign-in and contact pages .

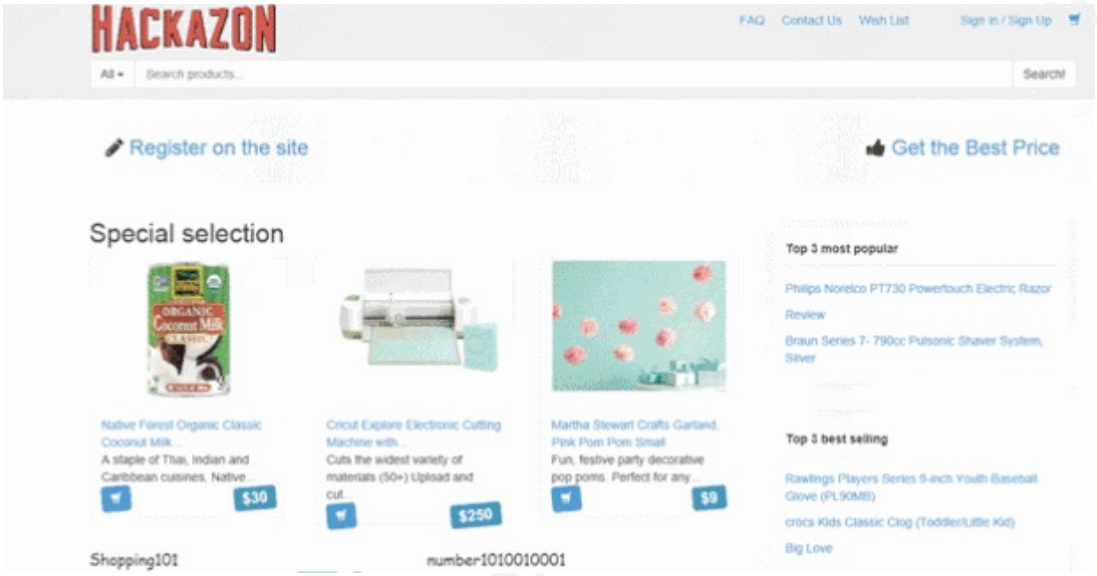

Figure 1: Hackazon (Rapid7 2018)

2.3. Installation and Configuration

Hackazon is a PHP web application and requires a PHP framework, an Apache server, and a MySQL database (Rapid7 2018). Hackazon was downloaded from https://github.com/rapid7/hackazon and was installed on a Windows 10 VM by following the installation guide (Rapid7 2018). WampServer 2 was installed and the appropriate DocumentRoot and Directory modifications were made. The MySQL database was created, and user credentials were set. The Hackazon Installation Wizard was used to set up the application. Difficulties logging into the Administrator Interface were overcome by adding the @hackazon.com stem to the username. The following vulnerabilities were set (further detail at Annex A):

Vulnerability	Type	Location URL
1	SQL	/
2	OSCommand	/account/documents
3	RemoteFileInclude	/account/help_articles
4	XSS	/account/orders[id]
5	Stored XSS	/account/profile/edit
6	ArbitraryFileUpload	/account/profile/edit
7	BlindSQL	/amf
8	SQL	/api/category [GET]
9	BlindSQL	/api/category [GET]
10	Stored XSS	/api/category/_id_ [GET]
11	XMLExternalEntity	/api/user/_id_ [PUT]
12	BlindSQL	/api/user/_id_ [PUT]
13	BlindSQL	/category/view
14	SQL	/checkout/billing
15	Stored XSS	/checkout/shipping
16	CSRF	/contact
17	SQL	/contact
18	Stored XSS	/faq

 Leonard Forster, e336712@gmail.com

19	BlindSQL	/helpdesk
20	SQL	/helpdesk
21	IntegerOverflow	/product/view
22	SQL	/product/view
23	XSS	/search
24	SQL	/user/login
25	BlindSQL	/wishlist/add-product/_id_
26	CSRF	/wishlist/new
27	XSS	/wishlist/new
28	SQL	/wishlist/remove-product/_id_

Ferruh Forster, e336712@gmail.com

3. Results

3.1. Acunetix (Trial Version)

A significant limitation of the Acunetix Vulnerability Scanner Trial Version is that it did not provide details of the specific location of the vulnerabilities detected. The Acunetix Vulnerability Scanner reported 45 total alerts: four high, 30 medium, eight low, and three informational. It performed well on XSS with an assessed 100% detection rate. It detected one of six Blind SQL vulnerabilities and one of two CSRF vulnerabilities. It did not identify SQL, Stored XSS, Integer Overflow, or File Upload vulnerabilities. The report is as follows:

Alerts Raised
45 total alerts, four high, 30 medium, eight low, three informational. **High:** XSS (3) BlindSQL (1) **Medium:** User Credentials sent in clear text (19) Application error messages (6) Vulnerable JS library (3) HTML form without CSRF protection (1) Insecure crossdomain.xml file (1) **Low:** Hidden form input found (1) Apache mod_negotiation filename brute-forcing Clickjacking X-Frame-Options header missing Cookie without HttpOnly flag set Cookie without secure flag set

 Leonard Forster, e336712@gmail.com

Login-page password guessing attack TRACE method enabled

3.2. OWASP Zed Attack Proxy (ZAP)

Automated Active Scan and the AJAX Spider modes were used. OWASP ZAP reported a high number of alerts across five vulnerability categories. ZAP highlighted site-wide issues such as ‘X-Frame-Options Header Not Set,' ‘Web Browser XSS Protection Not Enabled,' and ‘Path Traversal.' It did not flag any of the specific pre-configured vulnerabilities.

Alerts Raised
High: Path Traversal. Allows the attacker access to files, directories, and commands that may reside outside the web document root directory. **Medium:** X-Frame-Options Header Not Set (290). Not included in HTTP response to protect against ‘clickjacking' attacks. **Low:** Cookie No HttpOnly Flag (212) Web Browser XSS Protection Not Enabled (318) X-Content-Type-Options Header Missing (761)

3.3. BurpSuite Pro

BurpSuite Pro correctly identified several of the preconfigured vulnerabilities: one BlindSQL, two SQL, one CSRF, and two reflected XSS. It also identified a potential additional Python code injection vulnerability, as well as issues such as data from input returned in the application's response, and the HTTP TRACE method.

 Leonard Forster, e336712@gmail.com

Vulnerabilities Detected
High: Cross-site scripting (reflected) (5) Flash cross-domain policy Cleartext submission of the password (38) SQL Injection (3) Python code injection **Low:** Password field with autocomplete enabled (3) Unencrypted communications Cookie without HttpOnly flag set (2) **Other:** Input returned in response (403) HTTP TRACE enabled Email addresses disclosed Frameable response (potential Clickjacking) (39) Cross-site request forgery (4)

3.4. Analysis

Acunetix reported 40 total alerts, Burp Suite 499 (across 13 alert categories), and OWASP ZAP 1676 (across five alert categories). Differences in the classification of severity between the scanners complicated the analysis. For example, Burp Suite classified vulnerabilities as either high severity, low severity, or informational, whereas Acunetix and OWASP ZAP rated them as high, medium, and low.

Scanner	High	Medium	Low	Other	Total Alerts
Acunetix	4	30	6	-	40
Burp Suite	45	-	450	4	499
OWASP ZAP	1	313	1363	-	1676

Table 1: Total alerts by scanner and severity

The 2215 total alerts across the three scanners covered a spectrum of 22 vulnerability types. There were high numbers of generic items identified such as 'Input Reflected in Response,' 'No Brower XSS Protection,' 'Clickjacking X-Frame-Options,' and 'Cookie HTTPOnly not set' vulnerabilities. A significant number of these were related alerts; a single issue (for example, XSS) reported across multiple parameters. Overall, there were significant inconsistencies between each of the scanners. High consequence issues had to be picked out from the 'noise' of multiple alerts relating to more generic issues. The total alerts were analyzed to identify the number of unique vulnerability reports (figures 2 and 3 refer). Both BurpSuite and OWASP ZAP reported high numbers of duplicate alerts.

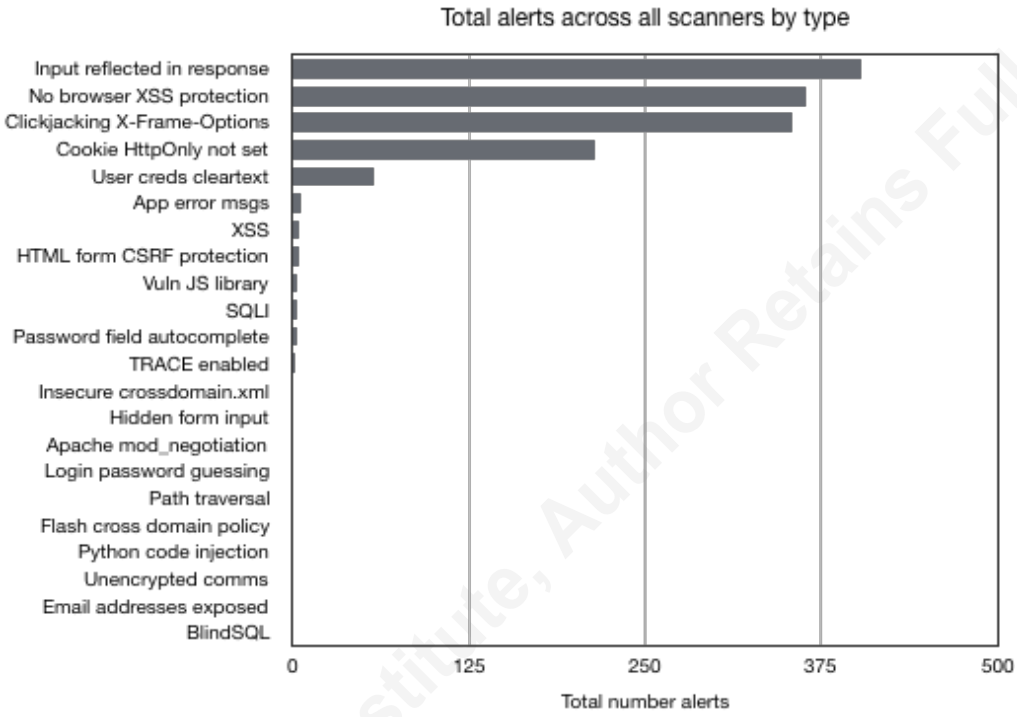

Figure 2: Total alerts across all scanners by type

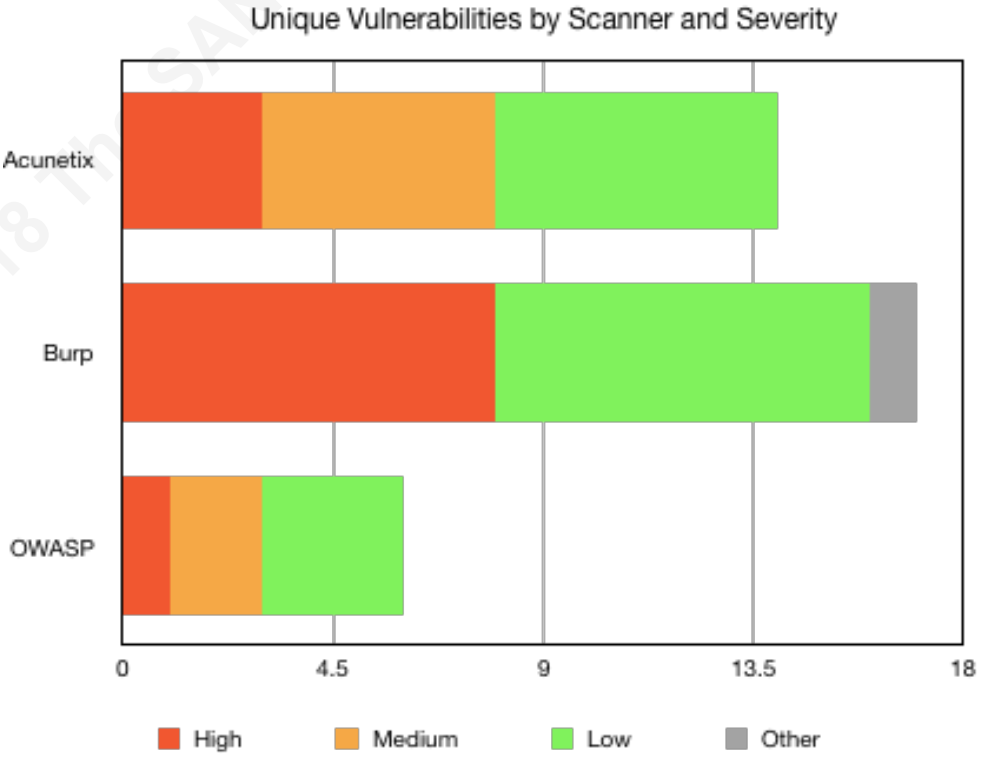

Figure 3: Unique vulnerabilities by scanner and severity

The scanners identified SQLI and XSS vulnerabilities. Both Acunetix and Burp Suite identified SQLI including BlindSQL issues, although coverage was far from complete. Similarly, reflected XSS was detected, and stored XSS were not detected, a finding consistent with previous studies by Bau et al. (2010) and Berbiche et al. (2017). This suggests that web application security scanners may frequently encounter problems when attempting to identify stored XSS vulnerabilities, perhaps due to limitations in their detection methods. For example, Burpsuite identified two of the preconfigured Reflected XSS vulnerabilities but did not flag any of the Stored XSS issues. This observation points to the challenges involved in detecting these attacks, and the importance of performing manual code security reviews. Burp Suite did flag that the HTTP TRACE method was enabled, a potential risk factor for XSS that could permit an attacker to steal cookie data via JS. Both Acunetix and Burp Suite identified CSRF-related issues. Acunetix identified an HTML form with no apparent CSRF protection enabled, and Burpsuite highlighted a similar feature that was vulnerable to attacks against unauthenticated functionality. Although encouraging that a range of vulnerabilities were successfully detected by the three web application security scanners, the successful detections were sufficiently inconsistent to indicate that a manual review of the vulnerable application would be required in addition to automated testing.

Furthermore, all three scanners tested overlooked many classes of vulnerability. These included OS Command Injection, Remote File Inclusion, and Integer Overflow vulnerabilities. This is broadly consistent with the findings of Doupé et al. (2010) in which half of the vulnerabilities were not detected by the scanners tested. In that study Stored SQL Injection, directory traversal, multi-step XSS, and logic flaw vulnerabilities were among those missed. OS Command injection uses a web interface to execute OS commands on the web server. Manual review and URL modification are used to detect it— a difficult challenge for an automated scanner. Remote file inclusion involves the exploitation of vulnerable inclusion procedures such as when a page receives a file path as input that is not correctly sanitized, allowing injection of an external URL (OWASP 2014). Testing should focus on scripts that use filenames as parameters and prevention must ensure disabling of the

remote inclusion feature in the relevant programming language. Integer overflow occurs when arithmetic operations cause a number to grow too large to be represented by the allocated bits (OWASP 2013). Manual review and testing methods can be used to detect and repair this type of vulnerability.

Overall, approximately 75% of the preconfigured vulnerabilities went undetected. There are two principal explanations for this: the limitations of web application security scanners and of 'Point-and-Shoot' (PaS) mode. In their 2010 study, Shelly et al. suggested possible explanations for these shortfalls. One may be that the overload of requests made to the server may lead to the server failing to produce proper response pages. In turn, the scanners may then fail to adequately handle the server responses. Consequently opportunities may be missed to test login pages that require human interaction and user authentication. Failings in the spidering techniques may cause scanners to overlook parameters or links. Doupé et al. (2010) found that modern web applications present crawling challenges to scanners. At the time, scanners were limited in their abilities to handle multimedia data, by incomplete or incorrect HTML parsers, and by lack of support for JS and Flash.

An additional explanation for the uneven performance is the use of PaS mode. Each of the security scanners was deployed against the vulnerable web application in this 'automated' or 'point-and-shoot' setting, a widely criticized methodology with significant limitations (Suto 2010). For example, the BurpSuite Pro documentation (Portswigger 2018) highlights the limitations of fully automated scanning of web applications. The developers warn that this approach to scanning will provide limited coverage. They attribute this to the rapid pace of change in client-side technologies, highly stateful application functionality, and the complexities of session handling (Portswigger 2018). The developers acknowledge that human insight is required to locate many critical vulnerabilities. They remind the user that scanners are designed to be deployed within a "user-driven testing workflow." The use of PaS mode undoubtedly reduced the number and range of vulnerabilities identified.

4. Conclusion

This study tested three web application security scanners against a realistic Web 2.0 web application. The web application, Hackazon, replicated an online storefront (Kuykendall 2014). It incorporated an e-commerce workflow as well as frameworks such as the Google Web Toolkit and JSON, as well as both AJAX and standard HTML use cases. All of the scanners identified potential vulnerabilities, but there was significant variation in the number and type detected. Overall 75% of configured vulnerabilities went undetected; OS Command Injection, Remote File Inclusion, Stored XSS, and Integer Overflow vulnerabilities were overlooked. Web application security scanners may have limited utility when tackling Web 2.0 applications with dynamic links, crawler traps, and form validation challenges. The lack of detection may also be explained by the limitations of PaS mode. Given the limited overlap in the vulnerability coverage achieved by the three scanners tested, this study supports the recommendation to use multiple web application security scanners together in conjunction with a specific scanning strategy to achieve greater coverage and accuracy (McQuade 2014).

4.1. Related Work

Due to limitations of time and resources, a pre-existing vulnerable web application tested three web application security scanners. Due to cost limitations, the Acunetix scanner was deployed in demonstration mode and therefore did not provide granular insights.

Several areas for potential future research were identified:

a) A significant number of studies have tested a wide range of scanners against multiple targets. An overarching analysis of these studies will identify general conclusions and trends

b) There are relatively few vulnerable web applications that incorporate modern web technologies. Testing a sample of scanners against multiple vulnerable

applications would allow for a cross-comparison of results and identification of strengths and weaknesses.

c) Suto (2010) found high variability in results between automated scanning and manual testing following a user-driven testing workflow. The low success rate in the present study points to the need for further research into the limitations of the automated approach, including behavioral considerations when undertaking manual testing.

4.2. Implications

Automated web application security scanners offer a useful tool to aid application security testing and education. This study and previous body of literature suggest that this tactic is not fruitful in isolation. Excessive reliance on automation could lead to a false sense of security and a reduction in coverage.

This study adds weight to the consensus that web application security scanners have significant limitations. The results support the OWASP recommendation that web application security testers adopt a holistic approach and go beyond the narrow conception of the researcher as the attacker (Ferreira & Kleppe 2011). Productive web application security-testing regimes comprise a blend of manual assessments and reviews, threat modeling, code review, and penetration testing. In relation to penetration testing, web application security scanners have a limited role to play. They are indeed generic and seductive (Keary 2013 in OWASP 2014) and automation is not the silver bullet it appears to be.

References

Acunetix (2017) What is Remote File Inclusion (RFI)?, accessed at https://www.acunetix.com/blog/articles/remote-file-inclusion-rfi/.

Acunetix (2018) Acunetix Vulnerability Scanner, accessed at https://www.acunetix.com/vulnerability-scanner/.

Acunetix (2018) AJAX security: Are AJAX Applications Vulnerable to Hack Attacks?, accessed at https://www.acunetix.com/websitesecurity/ajax/.

Alassmi, S., Zavarsky, P., Lindskog, D., Ruhl, R., Alasiri, A., & Alzaidi, M. (2012). An analysis of the Effectiveness of Black-box Web Application Scanners in Detection of Stored XSSI Vulnerabilities. International Journal of Information Technology and Computer Science, 4(1).

Alnabulsi, H., Islam, M. R., & Mamun, Q. (2014, November). Detecting SQL injection attacks using SNORT IDS. In Computer Science and Engineering (APWC on CSE), 2014 Asia-Pacific World Congress on (pp. 1-7). IEEE.

Alsmadi, I., Alsukhni, E., & Dabbour, M. (2013). Efficient assessment and evaluation for websites vulnerabilities using SNORT. International Journal of Security and Its Applications, 7(1), 7-16.

Arachni (2018). Arachni Web Application Scanner Framework, accessed at http://www.arachni-scanner.com.

Bakar, K. A. A., Daud, N. I., & 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.

Berbiche, N., El Idrissi, S., Guerouate, F., & Sbihi, M. (2017). Performance Evaluation of web application security scanners for Prevention and Protection against Vulnerabilities. International Journal of Applied Engineering Research, 12(21), 11068-11076.)

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.

Butkiewicz, M., Madhyastha, H. V., & Sekar, V. (2011, November). Understanding website complexity: measurements, metrics, and implications. In Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference (pp. 313-328). ACM.

Denim Group (2014) Limitations of Automated Tools for Dynamic Web Application Security Scanning, www.denimgroup.com/resources/blog/2014/05/limitations-automated-application/

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.

Fakhreldeen, A., & Eltyeb, E. (2014). Assessment of Open Source web application security scanners. College of Computer Science and Information Technology, KAU, Khulais, Saudi Arabia.

Ferreira, A. M., & Kleppe, H. (2011). Effectiveness of automated application penetration testing tools, 6 February 2011.

Giora, S (2009) Web Application Security, Clusif, pp.1-20.

Kumar, D., Ma, Z., Durumeric, Z., Mirian, A., Mason, J., Halderman, J. A., & Bailey, M. (2017, April). Security Challenges in an Increasingly Tangled Web. In Proceedings of the 26th International Conference on World Wide Web (pp. 677-684). International World Wide Web Conferences Steering Committee.

Kuykendall (2014) 'Hackazon: Stop hacking like it's 1999', presented at OWASP APP Sec USA 2014, available at https://youtu.be/Yekzm0Olc3Y.

Makino, Y., & Klyuev, V. Evaluation of web vulnerability scanners. In 2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS).

McQuade, K. (2014). Open source web vulnerability scanners: the cost-effective choice. In Proceedings of the Conference for Information Systems Applied Research ISSN (Vol. 2167, p. 1508).

National Cyber Security Centre (NCSC). (2017) Penetration testing guidance, accessed at www.ncsc.gov.uk/penetration-testing.

Orozco, A. L. S., Vega, E. A. A., & Villalba, L. J. G. (2017). Benchmarking of Pentesting Tools. World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering, 11(5), 590-593.

O'Reilly, T. (2007) What is web 2.0: design patterns and business models for the next generation of software, Communication and Strategies 65 (1), 17-37.

Orloff, J (2012) Understanding AJAX vulnerabilities, accessed at www.ibm.com/developerworks/library.

OWASP (2013) Testing for AJAX Vulnerabilities, accessed at https://www.owasp.org.

OWASP (2014) OWASP Testing Guide v4, accessed at https://www.owasp.org/index.php/OWASP_Testing_Guide_v4_Table_of_Contents.

OWASP (2017) OWASP 2017 Top 10, accessed at https://www.owasp.org/index.php/Top_10-2017_Top_10.

OWASP (2018) OWASP Zed Attack Proxy, accessed at https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project.

Portswigger (2018). Burp Suite Professional Features, accessed at www.portswigger.net/burp.

Portswigger (2018). Using Burp As a Point and Click Scanner, accessed at www.portswigger.net/burp/help/scanner_pointandclick.

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.

Salva, S., & Laurencot, P. (2009, May). Automatic Ajax application testing. In Internet and Web Applications and Services, 2009. ICIW'09. Fourth International Conference on(pp. 229-234). IEEE.

Shelly, D., Marchany, R. & Tront, J. (2010) Closing the gap: analyzing the limitations of web application vulnerability scanners, presentation to OWASP AppSec DC 2010.

Suto, L. (2010). Analyzing the accuracy and time costs of web application security scanners. San Francisco, February.

Symantec (2018) Symantec Threat Reports, accessed at www.symantec.com/security_reponse/publications/threatreport.jsp.

Web Application Security Consortium (2012), The WASC Threat Classification v2.0, accessed at www.webappsec.org.

Web Application Security Consortium (2009) web application security scanner Evaluation Criteria, accessed at www.webappsec.org.

Yuliana, M. (2012). Security Evaluation of Web Application Vulnerability Scanners' Strengths and Limitations Using Custom Web Application. California State University.

Zhu, C. (2017). Experimental study of vulnerabilities in a web application, Aalto University.

Annex A: Vulnerability Details

Hackazon was configured with the following vulnerabilities (OWASP 2016-2018):

a) SQL Injection: insertion of an SQL query from the client to the application using user input data;

b) OS Command Injection: the goal is the execution of arbitrary commands on the host OS via the vulnerable web application. These attacks become possible when applications pass unsafe user input data to a system shell;

c) Remote File Inclusion: an attacker causes the web application to include a remote file by when it allows them to insert external scripts or files dynamically. Consequences include information disclosure and Cross-site Scripting (XSS) to Remote Code Execution (Acunetix 2017);

d) Cross Site Scripting (XSS): injection of malicious scripts into otherwise trusted websites. XSS attacks occur when the attacker uses a web application to send malicious code, often as a browser side script, to a different end user;

e) Stored Cross Site Scripting (sometimes referred to as Persistent or Type-I XSS): permanent storage of an injected script on target servers in a database, message forum, visitor log, comment field, or similar. When requesting stored information, the victim unwittingly retrieves the malicious script;

f) Blind SQL: a type of SQL injection attack that queries the database with true or false challenges and establishes the answer based on the responses received. This vulnerability is typically often leveraged when a web application displays generic error messages but has not protected the code that is vulnerable to SQLI;

g) Arbitrary File Upload: an attacker accesses the upload function of the application without authenticating correctly;

h) XML External Entity: occurs when a weakly configured XML parser processes XML input containing a reference to an external entity;

i) Cross-Site Request Forgery (CSRF): forces the end user to execute unwanted actions on a web application in which they are authenticated;

j) Integer Overflow: Buffer overflows corrupt the execution stack of a web application. Arithmetic operations cause a number to grow too large to be represented in the bits allocated to it.