Exploiting the Potential of Web Application Vulnerability Scanning

Damiano Esposito, Marc Rennhard

School of Engineering
Zurich University of Applied Sciences
Winterthur, Switzerland
Email: espo,rema@zhaw.ch

Lukas Ruf, Arno Wagner

Consecom AG
Zurich, Switzerland
Email: Lukas.Ruf,Arno.Wagner@consecom.com

***Abstract*—Using automated web application vulnerability scanners so that they truly live up to their potential is difficult. Two of the main reasons for this are limitations with respect to crawling capabilities and problems to perform authenticated scans. In this paper, we present JARVIS, which provides technical solutions that can be applied to a wide range of vulnerability scanners to overcome these limitations. Our evaluation shows that by using JARVIS, the vulnerability detection performance of five freely available scanners can be improved by more than 100% compared to using them in their basic configuration. As the configuration effort to use JARVIS is small and the configurations are scanner-independent, JARVIS also allows to use multiple scanners in parallel in an efficient way. In an additional evaluation, we therefore analyzed the potential and limitations of using multiple scanners in parallel. This revealed that using multiple scanners in a reasonable way is indeed beneficial as it increases the number of detected vulnerabilities without a significant negative impact on the reported false positives.**

***Keywords*–Web Application Security; Vulnerability Scanning; Vulnerability Detection Performance.**

I. Introduction

Security testing is important to achieve security and trustworthiness of software and systems. Security testing can be performed in different ways, ranging from completely manual methods (e.g., manual source code analysis), to semi-automated methods (e.g., analyzing a web application using an interceptor proxy), to completely automated ways (e.g., analyzing a web service using a vulnerability scanner).

Ideally, at least parts of security testing should be automated. One reason for this is that it increases the efficiency of a security test and frees resources for those parts of a security test that cannot be easily automated. This includes, e.g., access control tests, which cannot really be automated as a testing tool doesn't have an understanding of which users or roles are allowed to perform what functions. Another reason is that automating security tests allows to perform continuous and reproducible security tests, which is getting more and more important in light of short software development cycles.

There are different options how to perform automated security testing. The most popular approaches include static and dynamic code analysis and vulnerability scanning. Vulnerability scanners test a running system "from the outside" by sending specifically crafted data to the system and by analyzing the received response. Among vulnerability scanners, web application vulnerability scanners are most popular, as web applications are very prevalent, are often vulnerable and are frequently attacked [1]. Note also that web applications are not only used to provide typical services such as information portals, e-shops or access to social networks, but they are also very prevalent to configure all kinds of devices attached to the Internet, which includes, e.g., switches, routers and IoT devices. This further undermines the importance of web application security testing.

At first glance, using web application vulnerability scanners seems to be easy as they claim to uncover many vulnerabilities with little configuration effort – as a minimum, they only require the base URL of the application to test as an input. However, their effective application in practice is far from trivial. The following list summarizes some of the limitations:

1) The detection capability of a scanner is directly dependent on its crawling performance: If a scanner can't find a specific resource in a web application, it can't test it and won't find vulnerabilities associated with this resource. Previous work shows that the crawling performance of different scanners varies significantly [2], [3].
2) To test areas of a web application that are only reachable after successful user authentication, the scanners must authenticate themselves during crawling and testing. While most scanners can be configured so they can perform logins, they typically do not support all authentication methods used by different web applications. Also, scanners sometimes log out themselves (e.g., by following a logout link) during testing and sometimes have problems to detect whether an authenticated session has been invalidated. Overall, this makes authenticated scans unreliable or even impossible in some cases.
3) To cope with these limitations, scanners usually provide configuration options, which can increase the number of detected vulnerabilities [4]. This includes, e.g., specifying additional URLs that can be used by the crawler as entry points, manually crawling the application while using the scanner as a proxy so it can learn the URLs, and specifying an authenticated session ID that can be used by the scanner to reach access-protected areas of the application if the authentication method used by the web application is not supported. However, using these options complicate the usage of the scanners and still do not always deliver the desired results.
4) With respect to the number and types of the reported findings, different vulnerability scanners perform dif-

ferently depending on the application under test [5]. Therefore, when testing a specific web application, it's reasonable to use multiple scanners in parallel and combine their findings. However, the limitations described above make this cumbersome and difficult, as each scanner has to be configured and optimized differently.

The goal of this paper is to overcome these limitations and to evaluate how much this improves the vulnerability detection performance of web application vulnerability scanners. To achieve this goal, we developed JARVIS, which provides technical solutions to overcome limitations 1 and 2 in the list above. Using JARVIS requires only minimal configuration, which overcomes limitation 3. And finally, JARVIS and its usage are independent of specific vulnerability scanners and can be applied to a wide range of scanners available today, which overcomes limitation 4 and which provides an important basis to use multiple scanners in parallel in an efficient way.

JARVIS was then applied to several vulnerability scanners to evaluate its effectiveness and to learn more about the potential and limitations of combining multiple scanners. In this analysis, the five freely available scanners listed in Table I were used.

TABLE I. ANALYZED WEB APPLICATION VULNERABILITY SCANNERS

Scanner	*Version/Commit*	*URL*
Arachni	1.5-0.5.11	http://www.arachni-scanner.com
OWASP ZAP	2.5.0	https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project
Skipfish	2.10b	https://code.google.com/archive/p/skipfish/
Wapiti	r365	http://wapiti.sourceforge.net
w3af	cb8e91af9	https://github.com/andresriancho/w3af

The choice for using freely available scanners was mainly driven by the goal to evaluate the performance of using multiple scanners in parallel. This is a much more realistic scenario with freely available scanners as commercial ones often have a hefty price tag. Also, previous work concluded that freely available scanners do not perform worse than commercial scanners [2], [3]. Arguments for using the scanners in Table I instead of using others include our previous experience with these scanners, that these scanners are among the most popular used scanners in practice, and that they perform well in general according to [3].

The main contributions of this paper are the following:

- Technical solutions to improve the crawling coverage and the reliability of authenticated scans of web application vulnerability scanners. In contrast to previous work (see Section IV), our solutions cover both aspects, can easily be applied to a wide range of scanners available today, and require only minimal, scanner-independent configuration.
- An evaluation that demonstrates how much the vulnerability detection performance of five different scanners is improved when using these technical solutions.
- An evaluation that demonstrates the benefits and limitations when using multiple scanners in parallel.

The remainder of this paper is organized as follows: Section II introduces the technical solutions to overcome the limitations of today's scanners and Section III contains the evaluation results. Related work is covered in Section IV and Section V concludes this work.

II. TECHNICAL APPROACH OF JARVIS

One way to improve the vulnerability detection performance of scanners is to directly adapt one or more current scanners. However, the main disadvantage of this approach is that this would only benefit one or a small set of scanners and would be restricted to scanners that are provided as open source software. Therefore, a proxy-based approach was chosen that is independent of any specific scanner, that does not require adaptation of current scanners, and that can be used with many scanners that are available today and most likely also with scanners that will appear in the future. The basic idea is illustrated in Figure 1.

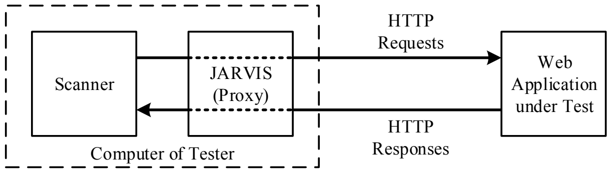


Figure 1. Proxy-based Approach of JARVIS.

A proxy-based approach means that JARVIS, which provides the technical solutions to overcome the limitations of the scanners, acts as a proxy between the scanner and the web application under test. This gives JARVIS access to all HTTP requests and responses exchanged between scanner and web application, which allows to control the entire crawling and scanning process and to adapt requests or responses as needed. This proxy-based approach is possible because most scanners are proxy-aware, i.e., they allow to configure a proxy through which communication with the web application takes place. Note that JARVIS can basically be located on any reachable host, but the typical scenario is using JARVIS on the same computer as the scanner (e.g., on the computer of the tester).

As a basis for JARVIS, the community edition version 1.7.19 of Burp Suite [6] is used. Burp Suite is a tool to support web application security testing that allows to record, intercept, analyze, modify and replay HTTP requests and responses. Therefore, Burp Suite already provides many basic functions that are required to implement JARVIS. In addition, Burp Suite provides an application programming interface (API) so it can be extended and JARVIS makes use of this API.

JARVIS consist of two main components. The first is described in Section II-A and aims at improving the test coverage of scanners. This component should especially help scanners that have a poor crawling performance. The second component, described in Section II-B, aims at improving the reliability of authenticated scans and should assist scanners that have limitations in this area. Finally, Section II-C gives a configuration example when using JARVIS to demonstrate that the configuration effort is small.

A. Improving Test Coverage

Improving test coverage could be done by replacing the existing crawler components of the scanners with a better one (see, e.g., [7]–[9]). While this may be helpful for some scanners, it may actually be harmful for others, in particular if the integrated crawler works well. Therefore, an approach was chosen that does not replace but that assists the crawling components that are integrated in the different scanners. The idea is to supplement the crawlers with additional URLs (beyond the base URL) of the web application under test. These additional URLs are named *seeds* as they are used to seed the crawler components of the scanners. Intuitively, this should significantly improve crawling coverage, in particular if the integrated crawler is not very effective. To get the additional URLs of a web application, two different approaches are used: endpoint extraction from the source code of web applications and using the detected URLs of the best available crawler(s).

Endpoint extraction means searching the source code (including configuration files) of the web application under test for URLs and parameters. The important benefits of this approach are that it can detect URLs that are hard to find by any crawler and that it can uncover hidden parameters of requests (e.g., debugging parameters). To extract the endpoints, ThreadFix endpoint CLI [10] was used, which supports many common web application frameworks (e.g., JSP, Ruby on Rails, Spring MVC, Struts, .NET MVC and ASP.NET WebForms). In addition, further potential endpoints are constructed by appending all directories and files under the root directory of the source code to the base URL that is used by the web application under test. This is particularly effective when scanning web applications based on PHP.

Obviously, endpoint extraction is only possible if the source code of the application under test is available. If that's not the case, the second approach comes into play. The idea here is to use the best available crawler(s) to gather additional URLs. As will be shown later, Arachni provides good crawling performance in general, so Arachni is a good starting point as a tool for this task. Of course, it's also possible to combine both approaches to determine the seeds: extract the endpoints from the source code (if available) *and* get URLs with the best available crawler(s).

Once the seeds have been derived, they must be injected into the crawler component of the scanners. To do this, most scanners provide a configuration option. However, this approach has its limitations as such an option is not always available and usually only supports GET requests but no POST requests. Therefore, the seeds are injected by JARVIS. To do this, four different approaches were implemented based on *robots.txt*, *sitemap.xml*, a landing page, and the index page.

Using *robots.txt* and *sitemap.xml* is straightforward. These files are intended to provide search engine crawlers with information about the target web site and are also evaluated by most crawler components of scanners. When the crawler component of a scanner requests such a file, JARVIS supplements the original file received from the web application with the seeds (or generates a new file with the seeds in case the web application does not contain the file at all). Both approaches work well but are limited to GET request.

The other two approaches are more powerful as they also support POST request. The landing page-based approach places all seeds as links or forms into a separate web page (named *landing.page*) and the scanner is configured to use this page as the base URL of the web application under test (e.g., http://www.example.site/landing.page instead of http://www.example.site). When the crawler requests the page, JARVIS delivers the landing page, from which the crawler learns all the seeds and uses them during the remainder of the crawling process. One limitation of this approach is that the altered base URL is sometimes interpreted as a directory by the crawler component of the scanners, which means the crawler does not request the landing page itself but tries to fetch resources below it. This is where the fourth approach comes into play. The index page-based approach injects seeds directly into the first page received from the web application (e.g., just before the </body> tag of the page *index.html*). Overall, these four approaches allowed to successfully seed all scanners in Table I when used to test the web applications in the test set (see Section III-A).

As an example, the effectiveness of the landing page-based approach is demonstrated. To do this, WIVET version 4 [11] is used, which is a benchmarking project to assess crawling coverage. Table II shows the crawling coverage that can be achieved with OWASP ZAP (in headless mode) and Wapiti when they are seeded with the crawling results of Arachni via a landing page.

TABLE II. CRAWLING COVERAGE

Scanner	*Raw coverage*	*Coverage when seeded with the crawling results of Arachni*
Arachni	92.86%	
OWASP ZAP	14.29%	96.43%
Wapiti	48.21%	96.43%

Table II shows that the raw crawling coverage of Arachni is already very good (92.86%), while Wapiti only finds about half of all resources and OWASP ZAP only a small fraction. By seeding OWASP ZAP and Wapiti with the crawling results of Arachni, their coverage can be improved drastically to 96.43%. This demonstrates that seeding via a landing page indeed works very well.

B. Improving Authenticated Scans

Performing authenticated scans in a reliable way is challenging for multiple reasons. This includes coping with various authentication methods, prevention of logouts during the scans, and performing re-authentication when this is needed (e.g., when a web application with integrated protection mechanisms invalidates the authenticated session when being scanned) to name a few. It is therefore not surprising that many scanners have difficulties to perform authenticated scans reliably.

To deal with these challenges, several modules were implemented in JARVIS. The first one serves to handle various authentication methods, including modern methods based on HTTP headers (e.g., OAuth 2.0). The module provides a wizard to configure authentication requests, can submit the corresponding requests, stores the authenticated cookies received from the web applications, and injects them into subsequent requests from the scanner to make sure the requests are interpreted as authenticated requests by the web application. The main advantages of this module are that it enables authenticated scans even if a scanner does not support the authentication method and that it provides a consistent way to configure authentication independent of a particular scanner.

Furthermore, a logout prevention module was implemented to make sure a scanner is not doing a logout by following links or performing actions which most likely invalidate the current session (e.g., change password or logout links). This is configured by specifying a set of corresponding URLs that should be avoided during the scan. When the proxy detects such a request, it blocks the request and generates a response with HTTP status code 200 and an empty message body. In addition, a flexible re-authentication module was developed. Re-authentication is triggered based on matches of configurable literal strings or regular expressions with HTTP response headers (e.g., the *location* header in a redirection response) or with the message body of an HTTP response (e.g., the occurrence of a keyword such as *login*).

C. *Configuration Example*

To give an impression of the configuration effort needed when using JARVIS, Table III lists the parameters that must be configured when scanning the test application BodgeIt (see Section III-A). In this example, the seeds are extracted from the source code.

TABLE III. EXAMPLE CONFIGURATION WHEN SCANNING BODGEIT

Parameter	*Value(s)*
Base URL	http://bodgeit/
Source code	~/bodgeit/
Authentication mode	POST
Authentication URL	http://bodgeit/login.jsp

Authentication parameters	password=password
	username=test@test.test

Out of scope	http://bodgeit/password.jsp http://bodgeit/register.jsp http://bodgeit/logout.jsp
Re-auth. search scope	HTTP response body
Re-auth. keywords	Login, Guest, user
Re-auth. keyword interpretation	Literal string(s)
Re-auth. case-sensitive	True
Re-auth. match indicates	Invalid session
Seeding approach(es)	Landing page, robots.txt, sitemap.xml

The entries in Table III are self-explanatory and show that the configuration effort is rather small. In particular, the configuration is independent of the actual scanner, which implies that when using multiple scanners in parallel (see Section III-D), this configuration must only be done once and not once per scanner.

III. EVALUATION

This section starts with a description of the evaluation setting. Then, the results of the evaluation of the vulnerability detection performance is presented when the scanners are used with and without the improvements described in Section II. In the final step, the benefits and limitations of using multiple scanners in parallel is evaluated.

A. *Evaluation Setting*

Table IV lists the web applications that were used to evaluate the scanners (Cyclone Transfers and WackoPicko do not use explicit versioning).

All these applications are deliberately insecure and well suited for security training and to test vulnerability scanners. The main reason why the applications in Table IV were chosen is because they cover various technologies, including Java, PHP, Node.js and Ruby on Rails.

TABLE IV. WEB APPLICATIONS USED FOR THE EVALUATION

Application	*Version*	*URL*
BodgeIt	1.4.0	https://github.com/psiinon/bodgeit
Cyclone Transfers	–	https://github.com/thedeadrobots/bwa_cyclone_transfers
InsecureWebApp	1.0	https://www.owasp.org/index.php/Category:OWASP_Insecure_Web_App_Project
Juice Shop	2.17.0	https://github.com/bkimminich/juice-shop
NodeGoat	1.1	https://github.com/OWASP/NodeGoat
Peruggia	1.2	https://sourceforge.net/projects/peruggia/
WackoPicko	–	https://github.com/adamdoupe/WackoPicko

The evaluation uses four different configurations that are listed in Table V.

TABLE V. CONFIGURATIONS USED DURING THE EVALUATION

Config.	*The scans are executed...*
-/-	...without seeding and non-authenticated (i.e., using the basic configuration of the scanners by setting only the base URL)
S/-	...with seeding and non-authenticated (i.e., using the technical solution described in Section II-A)
-/A	...without any seeding and authenticated (i.e., using the technical solution described in Section II-B)
S/A	...with seeding and authenticated (i.e., using both technical solutions described in Sections II-A and II-B)

As the source code of all these applications is available, the endpoint extraction approach described in Section II-A is used for seeding in configurations *S/-* and *S/A*.

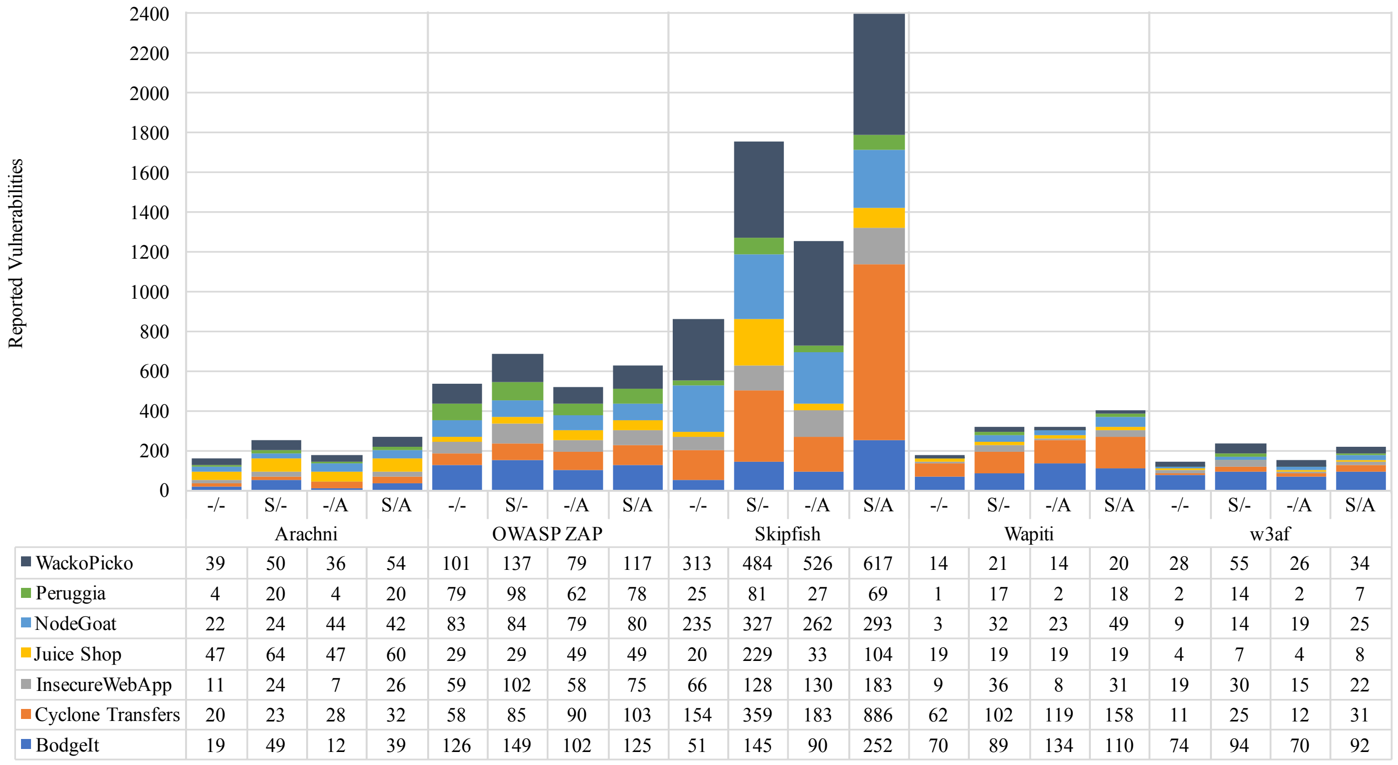
The test applications were run in a virtual environment that was reset to its initial state before each test run to make sure that every run is done under the same conditions and is not influenced by any of the other scans.

B. *Overall Evaluation*

The first evaluation analyzes the overall number of vulnerabilities that are reported by the scanners when using the four different configurations described in Table V. Figure 2 illustrates the evaluation results.

The first observation when looking at Figure 2 is that some scanners identify many more vulnerabilities than others. For example, Skipfish reports about ten times as many findings as Arachni or w3af. However, this doesn't mean that Skipfish is the best scanner, because Figure 2 depicts the "raw number of vulnerabilities" reported by the scanners and does not take into account false positives, duplicate findings, or the criticality of the findings. For instance, about 80% of the vulnerabilities reported by Skipfish are rated as *info* or *low* (meaning they have only little security impact in practice) while the other scanners report a much smaller fraction of such findings.

More importantly, Figure 2 shows that the technical solution to improve test coverage works well with all scanners and all test applications included in the evaluation. The number of vulnerabilities reported when seeding is used is nearly always greater than without seeding. For instance, Arachni reports 64 vulnerabilities in Juice Shop in configuration *S/-* compared to 47 in configuration *-/-*. Similarly, when using authenticated scans, Arachni reports 39 findings in BodgeIt in configuration *S/A* compared to 12 in configuration *-/A*. The same is true when adding up the vulnerabilities of a specific scanner over all test applications: Configuration *-/-* always reports fewer findings than configuration *S/-* (e.g., 162 vs. 254 with Arachni) and configuration *-/A* always reports



	-/-	S/-	-/A	S/A	-/-	S/-	-/A	S/A	-/-	S/-	-/A	S/A	-/-	S/-	-/A	S/A	-/-	S/-	-/A	S/A
	Arachni				OWASP ZAP				Skipfish				Wapiti				w3af			
⬛ WackoPicko	39	50	36	54	101	137	79	117	313	484	526	617	14	21	14	20	28	55	26	34
🟩 Peruggia	4	20	4	20	79	98	62	78	25	81	27	69	1	17	2	18	2	14	2	7
🟦 NodeGoat	22	24	44	42	83	84	79	80	235	327	262	293	3	32	23	49	9	14	19	25
🟨 Juice Shop	47	64	47	60	29	29	49	49	20	229	33	104	19	19	19	19	4	7	4	8
⬜ InsecureWebApp	11	24	7	26	59	102	58	75	66	128	130	183	9	36	8	31	19	30	15	22
🟧 Cyclone Transfers	20	23	28	32	58	85	90	103	154	359	183	886	62	102	119	158	11	25	12	31
🟦 BodgeIt	19	49	12	39	126	149	102	125	51	145	90	252	70	89	134	110	74	94	70	92

Figure 2. Reported Vulnerabilities per Scanner and Test Application.

fewer findings than configuration *S/A* (e.g., 178 vs. 273 with Arachni).

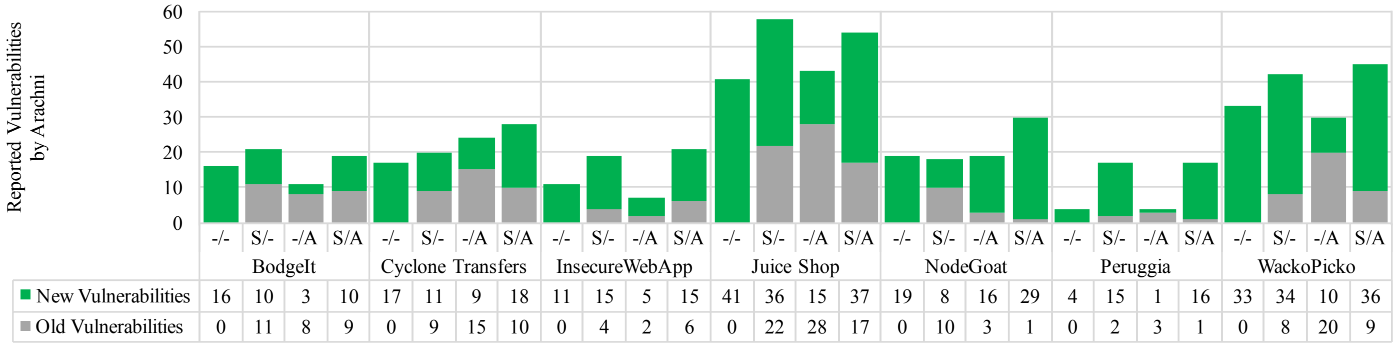
Likewise, Figure 2 demonstrates that the technical solution to improve authenticated scans also works well. For instance, when scanning Cyclone Transfer, Wapiti reports 62 findings in configuration *-/-* and 119 findings in configuration *-/A*. Also, scanning in configuration *S/-* delivers 102 vulnerabilities, which can be increased to 158 in configuration *S/A*. And finally, this also holds true over all applications, as for most scanners, the bars in Figure 2 are higher in configuration *-/A* compared to *-/-* and in configuration *S/A* compared to *S/-*. Note that to make sure that authenticated scans were carried out reliably, the involved requests and responses were analyzed after each scan. This showed that it was indeed possible to maintain authentication during the entire scan, which further undermines that the technical approach is sound.

Intuitively, additionally seeding a scanner or performing authenticated scans should always also report all vulnerabilities that are detected when scanning without additional seeding or without authentication. However, this is not the case. To demonstrate this, it was analyzed how many of the vulnerabilities reported in the basic configuration are also found when scanning in other configurations. To do this, the reports of the scanners were first processed with ThreadFix [12]. ThreadFix allows to normalize reports of different scanners, to eliminate duplicates, and to compare the results of different scanners or different runs by the same scanner. Figure 3 shows the results of the analysis for the findings reported by Arachni. Note that because of the processing with ThreadFix and in contrast to Figure 2, the bars now represent the number of unique findings that were reported.

First of all, Figure 3 undermines what was observed above: Additional seeding and authenticated scans result in a greater number of reported findings, as can be seen by comparing the heights of the bars. In addition, as the bars represent unique vulnerabilities, the additional findings are not just duplicates of already detected findings, but they are truly new findings. Beyond this, Figure 3 confirms that when using additional seeding and/or authenticated scans, not all vulnerabilities that are reported in the basic configuration *-/-* are detected again. For example, considering BodgeIt, Arachni reports 16 findings in configuration *-/-*. When using configuration *S/-*, then 21 findings are reported in total, of which 10 are “new” findings compared to *-/-* (indicated by the green part of the bar). However, only 11 of the 16 vulnerabilities reported in configuration *-/-* are detected again (“old” findings, indicated by the gray part of the bar) and 5 are missing. The same can be observed with the other configurations and with all test applications, which means that in general, additional seeding and/or authenticated scans deliver a significant number of new findings, but also misses several of the findings that are reported in the basic configuration. Note that the same behavior can be observed with all scanners, but only the results of Arachni are included due to space restrictions.

Determining the exact reasons for this behavior would require a detailed analysis of the crawling components of the scanners and the web applications in the test set, which is beyond the scope of this work. Therefore, only a few arguments are given that show the observed behavior is reasonable:

- Providing the crawler component of a scanner with additional seeds has a direct impact on the order in which the pages are requested. A different order



	BodgeIt				Cyclone Transfers				InsecureWebApp				Juice Shop				NodeGoat				Peruggia				WackoPicko			
	-/-	S/-	-/A	S/A	-/-	S/-	-/A	S/A	-/-	S/-	-/A	S/A	-/-	S/-	-/A	S/A	-/-	S/-	-/A	S/A	-/-	S/-	-/A	S/A	-/-	S/-	-/A	S/A
🟩 New Vulnerabilities	16	10	3	10	17	11	9	18	11	15	5	15	41	36	15	37	19	8	16	29	4	15	1	16	33	34	10	36
▨ Old Vulnerabilities	0	11	8	9	0	9	15	10	0	4	2	6	0	22	28	17	0	10	3	1	0	2	3	1	0	8	20	9

Figure 3. Reported Unique Vulnerabilities by Arachni and per Test Application.

implies different internal state changes within the web application under test [7], which typically leads to a different behavior of the web application and therefore to different findings.

- When doing authenticated scans, some of the resources that do not require authentication are often no longer reachable, e.g., registration, login and forgotten password pages. As deliberately insecure web applications often use these resources to place common vulnerabilities, this has a major impact in this test setting.

The consequence of this observation is that when scanning a web application, the scanners should be used in all four configurations to maximize the number of reported findings. And obviously, although this was not analyzed in detail, when an application provides different protected areas for different roles, scanning should be done with users of all roles.

C. Detailed Evaluation

The evaluation in Section III-B demonstrates that when considering just the number of reported vulnerabilities, JARVIS works well. However, its still unclear whether there's a true benefit in practice because it may be that the additionally found vulnerabilities are mainly false positives or non-critical issues.

To get a better understanding, a more detailed analysis focusing on SQL injection (SQLi) and cross-site scripting (XSS) vulnerabilities was done. To do this, all reported vulnerabilities of these types were manually verified to identify them as either true or false positives. This required a lot of effort, which is the main reason why the focus was set on these two types. Nevertheless, this serves well to evaluate the true potential of JARVIS as both vulnerabilities are highly relevant in practice and highly security-critical. In addition, the test applications contain several of them, which means SQLi and XSS vulnerabilities represent a meaningful sample size. Figure 4 shows the results of this analysis. Just like in Figure 3, the bars represent the number of unique findings that were reported.

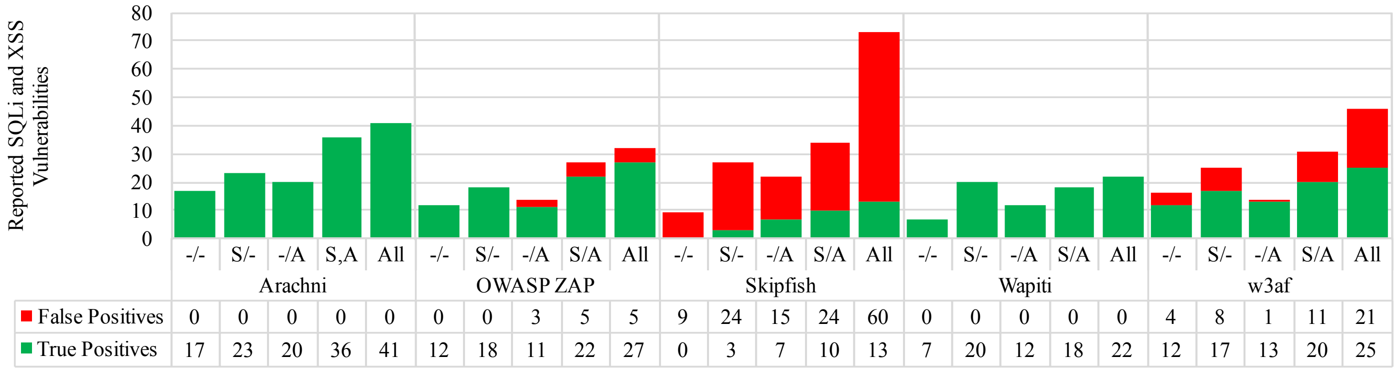
Looking only at the true positives (green bars), Figure 4 confirms that JARVIS indeed works well in the sense that using additional seeding and authenticated scans allows the scanners to detect highly relevant and security-critical vulnerabilities that are not reported in the basic configuration, which is true for all scanners. The results also undermine that it's important to perform scans in all four configurations (named configuration *All*), as the sums of the detected vulnerabilities (bars labeled with *All*) are always greater than the vulnerabilities detected in any of the other configurations. Furthermore, the results demonstrate that for each of the five scanners, combining the results of all configurations yields more than twice as many vulnerabilities (true positives) as when performing scans only in the basic configuration -/-, so JARVIS results in an improvement of over 100%.

In addition, Figure 4 shows that scanners that tend towards reporting false positives (red bars) do so also in the advanced configurations, but overall, the fraction of false positives remains more or less constant independent of the configuration. That's an important results as it demonstrates that the technical improvements result in more true findings without an increased percentage of false positives. And finally, Figure 4 allows to compare the scanners. In particular, based on the test applications and focusing on SQLi and XSS vulnerabilities, it shows that Arachni performs best (without producing a single false positive) and Skipfish performs quite poorly, especially with respect to false positives. This also puts into perspective the results of the first evaluation (see Figure 2), where Skipfish reported many more vulnerabilities than the other scanners.

D. Combining Multiple Scanners

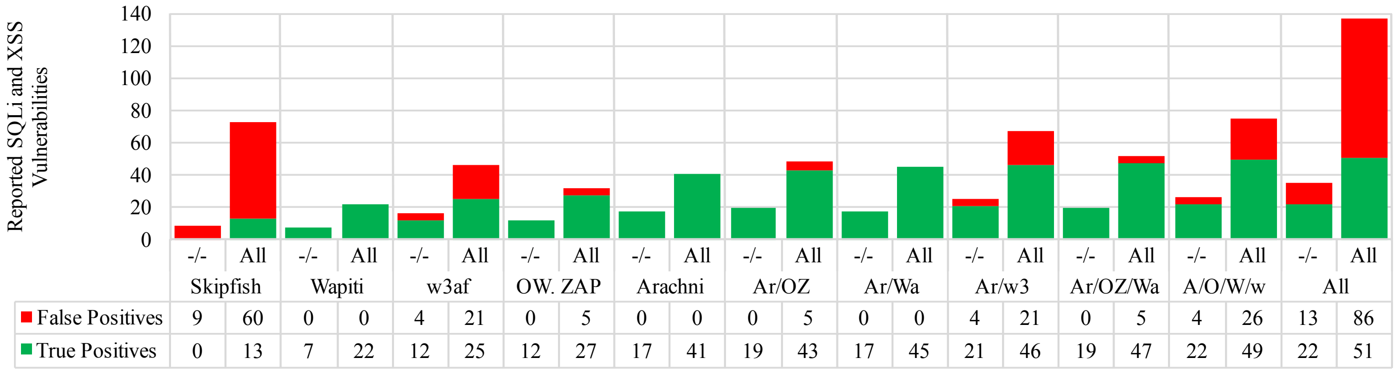
In the final evaluation, the benefits of using multiple scanners in parallel are analyzed. Figure 5 shows the combined unique true and false positives when using individual scanners and different combinations thereof and when using the scanners in the basic configuration -/- or in configuration *All*. The results are ranked from left to right according to the number of true positives that are identified in configuration *All*.

Looking at the results in configuration *All*, the rightmost bar combines the results of all five scanners, which obviously delivers most true positives (51), but which also delivers most false positives (86). The results also show that in this test setting, Arachni performs very well on its own, as it finds 41 true positives (without a single false positive), which means that the other four scanners combined can only detect 10 true positives that are not found by Arachni. Looking at combinations of scanners, then Arachni & Wapiti (Ar/Wa) perform well and identify 45 of the 51 true positives without any false positive. Combining Arachni, OWASP ZAP & Wapiti (Ar/OZ/Wa) is also a good choice as it finds 47 true positives with only a few false positives. This demonstrates



	Arachni					OWASP ZAP					Skipfish					Wapiti					w3af				
	-/-	S/-	-/A	S,A	All	-/-	S/-	-/A	S/A	All	-/-	S/-	-/A	S/A	All	-/-	S/-	-/A	S/A	All	-/-	S/-	-/A	S/A	All
🟥 False Positives	0	0	0	0	0	0	0	3	5	5	9	24	15	24	60	0	0	0	0	0	4	8	1	11	21
🟩 True Positives	17	23	20	36	41	12	18	11	22	27	0	3	7	10	13	7	20	12	18	22	12	17	13	20	25

Figure 4. Reported Unique SQLi and XSS Vulnerabilities per Scanner, over all Test Applications.



	Skipfish		Wapiti		w3af		OW. ZAP		Arachni		Ar/OZ		Ar/Wa		Ar/w3		Ar/OZ/Wa		A/O/W/w		All	
	-/-	All	-/-	All	-/-	All	-/-	All	-/-	All	-/-	All	-/-	All	-/-	All	-/-	All	-/-	All	-/-	All
🟥 False Positives	9	60	0	0	4	21	0	5	0	0	0	5	0	0	4	21	0	5	4	26	13	86
🟩 True Positives	0	13	7	22	12	25	12	27	17	41	19	43	17	45	21	46	19	47	22	49	22	51

Figure 5. Reported Unique SQLi and XSS Vulnerabilities using different Scanner Combinations, over all Test Applications.

that combining multiple scanners is beneficial to increase the number of detected true positives without a significant negative impact on the number of reported false positives. However, blindly combining as many scanners as possible (e.g., all five scanners used here) is not a good idea in general as although this results in most true positives, it also combines all false positives. Finally, comparing the results in configuration *All* with the ones in configuration *-/-* demonstrates that even when combining multiple scanners, configuration *All* increases the number of detected true positives always by more than 100%, which again undermines the benefits of JARVIS.

Note that since seven test web applications that cover several technologies are used, the results are at least an indication that the combinations of scanners proposed above should perform well in many scenarios. However, this is certainly no proof and it may be that other combinations of scanners are better suited depending on the web application under test. This means that in practice, one has to experiment with different combinations to determine the one that is best suited in a specific scenario.

IV. Related Work

Several work has been published on the crawling coverage and detection performance of web application vulnerability scanners. In [2], more than ten scanners were compared, with the main results that good crawling coverage is paramount to detect many vulnerabilities and that freely available scanners perform as well as commercial ones. The same is confirmed by [3], which covers more than 50 free and commercial scanners and which is updated regularly. In [4], Suto concludes that when carefully training or configuring a scanner, detection performance is improved, but this also significantly increases the complexity and time effort needed to use a scanner. Furthermore, Bau et al. demonstrate that the eight scanners they used in their analysis have different strengths, i.e. they find different vulnerabilities [5].

Other work specifically aimed at improving the coverage of vulnerability scanning. In [7], it is demonstrated that by taking into account the state changes of a web application when it processes requests, crawling and therefore scanning performance can be improved. In [8], van Deursen et al. present a Selenium WebDriver-based crawler called Crawljax, which improves crawling of Ajax-based web applications. The same is achieved by Pellegrino et al. by dynamically analyzing JavaScript code in web pages [9].

Our work presented in this paper builds upon this previous work as it delivers practical and effective technical solutions to overcome the limitations and exploit the potential identified by others. What sets our approach apart from other work is that it addresses not only crawling coverage but also the reliability of authenticated scans, that it is scanner-independent, and that it can easily be applied to most vulnerability scanners available today. In addition, we provide a detailed evaluation using several scanners and several test applications that truly demonstrates the benefits and practicability of our technical solutions.

V. CONCLUSION

In this paper, we presented JARVIS, which provides technical solutions to overcome some of the limitations – notably crawling coverage and reliability of authenticated scans – of web application vulnerability scanners. As JARVIS is independent of specific scanners and implemented as a proxy, it can be applied to a wide range of existing vulnerability scanners. The evaluation based on five freely available scanners and seven test web applications covering various technologies demonstrates that JARVIS works well in practice and that the vulnerability detection rate (true positives) of the scanners can by improved by more than 100% compared to using the scanners in their basic configuration.

The configuration effort to use JARVIS is small and the configurations are scanner-independent. Therefore, JARVIS also provides an important basis to use multiple scanners in parallel in an efficient way. The provided analysis shows that combining multiple scanners is indeed beneficial as it increases the number of true positives, which is not surprising as different scanners detect different vulnerabilities. However, it was also demonstrated that blindly combining as many scanners as possible is not a good idea in general because although this results in most true positives, it also delivers the sum of all false positives reported by the scanners. In the evaluation, the combination of Arachni & Wapiti or Arachni, OWASP ZAP & Wapiti yielded the best compromise between a high rate of true positives and a low rate of false positives. As a representative set of web application technologies was used in the evaluation, it can be expected that these combinations work well in many scenarios, but this is no proof and in practice, one has to experiment with different combinations to determine the one that is best suited in a specific scenario.

ACKNOWLEDGEMENT

This work was partly funded by the Swiss Confederation's innovation promotion agency CTI (project 18876.1 PFES-ES).

REFERENCES

[1] WhiteHat Security, "2017 application security statistics report," Tech. Rep., 2017, URL: https://www.whitehatsec.com/resources-category/premium-content/web-application-stats-report-2017/ [accessed: 2018-05-15].

[2] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: An analysis of black-box web vulnerability scanners," in Proceedings of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, ser. DIMVA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 111–131.

[3] S. Chen, "SECTOOL market," 2016, URL: http://www.sectoolmarket.com/price-and-feature-comparison-of-web-application-scanners-unified-list.html [accessed: 2018-05-15].

[4] L. Suto, "Analyzing the accuracy and time costs of web application security scanners," Tech. Rep., 2010, URL: https://www.beyondtrust.com/wp-content/uploads/Analyzing-the-Accuracy-and-Time-Costs-of-Web-Application-Security-Scanners.pdf [accessed: 2018-05-15].

[5] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in Proceedings of the 2010 IEEE Symposium on Security and Privacy, 2010, pp. 332–345.

[6] PortSwigger, "Burp Suite," URL: https://portswigger.net/burp [accessed: 2018-05-15].

[7] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in Proceedings of the 21st USENIX Security Symposium (USENIX Security 12). Bellevue, WA: USENIX, 2012, pp. 523–538.

[8] A. v. Deursen, A. Mesbah, and A. Nederlof, "Crawl-based analysis of web applications: Prospects and challenges," Science of Computer Programming, vol. 97, 2015, pp. 173 – 180.

[9] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow, "jäk: Using dynamic analysis to crawl and test modern web applications," in Research in Attacks, Intrusions, and Defenses, H. Bos, F. Monrose, and G. Blanc, Eds. Cham: Springer International Publishing, 2015, pp. 295–316.

[10] ThreadFix, "ThreadFix endpoint CLI," URL: https://github.com/denimgroup/threadfix/tree/master/threadfix-cli-endpoints [accessed: 2018-05-15].

[11] B. Urgun, "WIVET: Web input vector extractor teaser," URL: https://github.com/bedirhan/wivet [accessed: 2018-05-15].

[12] ThreadFix, "Application vulnerability correlation with ThreadFix," URL: https://threadfix.it [accessed: 2018-05-15].