# Scanner++: Enhanced Vulnerability Detection of Web Applications with Attack Intent Synchronization

ZIJING YIN, YIWEN XU, and FUCHEN MA, Tsinghua University, China
HAOHAO GAO, China Central Depository & Clearing Co., Ltd., China
LEI QIAO, Beijing Institute of Control Engineering, China
YU JIANG, Tsinghua University, China

Scanners are commonly applied for detecting vulnerabilities in web applications. Various scanners with different strategies are widely in use, but their performance is challenged by the increasing diversity of target applications that have more complex attack surfaces (i.e., website paths) and covert vulnerabilities that can only be exploited by more sophisticated attack vectors (i.e., payloads). In this paper, we propose Scanner++, a framework that improves web vulnerability detection of existing scanners through combining their capabilities with attack intent synchronization. We design Scanner++ as a proxy-based architecture while using a package-based intent synchronization approach. Scanner++ first uses a purification mechanism to aggregate and refine attack intents, consisting of attack surfaces and attack vectors extracted from the base scanners' request packets. Then, Scanner++ uses a runtime intent synchronization mechanism to select relevant attack intents according to the scanners' detection spots to guide their scanning process. Consequently, base scanners can expand their attack surfaces, generate more diverse attack vectors and achieve better vulnerability detection performance.

For evaluation, we implemented and integrated Scanner++ together with four widely used scanners, BurpSuite, AWVS, Arachni, and ZAP, testing it on ten benchmark web applications and three well-tested real-world web applications of a critical financial platform from our industry partner. Working under the Scanner++ framework helps BurpSuite, AWVS, Arachni, and ZAP cover 15.26%, 37.14%, 59.21%, 68.54% more pages, construct 12.95×, 1.13×, 15.03×, 52.66× more attack packets, and discover 77, 55, 77, 176 more bugs, respectively. Furthermore, Scanner++ detected eight serious previously unknown vulnerabilities on real-world applications, while the base scanners only found three of them.

CCS Concepts: • **Security and privacy** → **Vulnerability scanners**; *Web application security*; • **Software and its engineering** → Software testing and debugging;

Additional Key Words and Phrases: Web security, scanner, attack intent, synchronization

This research is sponsored in part by the NSFC Program (No. 62022046, No.92167101, U1911401, 62021002, 61802223), National Key Research and Development Project (Grant No. 2019YFB1706203), and the Tsinghua-Webank Scholar Project (No. 20212001829).

Authors' addresses: Z. Yin, Y. Xu, F. Ma, and Y. Jiang (corresponding author), Tsinghua University, Shuangqing Rd., Haidian District, Beijing, 100084, China; email: Aurora@europe.com; H. Gao, China Central Depository & Clearing Co., Ltd., Jinrong St., Xicheng District, Beijing, 100033, China; L. Qiao, Beijing Institute of Control Engineering, Zhongguancun South St., Haidian District, Beijing, 100081, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.

1049-331X/2023/02-ART7 $15.00
https://doi.org/10.1145/3517036

**ACM Reference format:**
Zijing Yin, Yiwen Xu, Fuchen Ma, Haohao Gao, Lei Qiao, and Yu Jiang. 2023. Scanner++: Enhanced Vulnerability Detection of Web Applications with Attack Intent Synchronization. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 7 (February 2023), 30 pages.
https://doi.org/10.1145/3517036

## 1 INTRODUCTION

Vulnerabilities in web applications are prevalent nowadays, accounting for the vast majority of security issues in the Common Vulnerabilities and Exposures database [10]. Some of them can lead to severe consequences once exploited by attackers. Today, many different methods have been applied to detect and discover security issues in web applications. Among them, web vulnerability scanners are one of the most commonly used tools.

In general, web vulnerability scanners can be divided into two categories, white-box, and black-box. White-box scanners are **static application software testing tools (SAST)**. They can analyze the source code of web applications and trace their program logic to detect security issues. For example, phpSAFE [25] uses static analysis to identify vulnerabilities in PHP-based applications developed with **OOP (Object-oriented Programming)**. RIPS [33] performs semantic analysis based on source code to build a model for the program and detects vulnerabilities using taint analysis. In practice, however, the source code of web applications is sometimes not available. Besides, many vulnerabilities are caused by the overlapping effects between application security issues and inappropriate server configurations, which are difficult for white-box scanners to deal with. These tools can only detect vulnerabilities from the perspective of the application source code, ignoring server factors. It will inevitably lead to more false positives and miss security issues.

In contrast, black-box scanners are **dynamic application software testing tools (DAST)**, only requiring access to the target website. Because of their simplicity of use and low false alarm rate, they have attracted much attention. Some black-box scanners even become a necessary part of several standards like Payment Card Industry Data Security Standard [29]. Black-box scanners work by simulating the hacking process, constructing attack intents against the target website, and analyzing response packets to detect vulnerabilities. A valid attack intent consists of a vulnerable attack surface (i.e., website path) and appropriate attack vectors (i.e., payloads). The work cycle of a typical black-box scanner can be divided into three stages, discovering attack surfaces, generating attack vectors, and analyzing response packets. The first stage mainly performs a content discovery procedure. Scanners will collect the scanning scope, web pages, and input points to determine possible attack surfaces of the target website. Subsequently, in the second stage, the attack module of the scanner will generate attack vectors for each attack surface to construct attack intents and send them to the target site. Finally, when the corresponding response packet is returned, it will be handled by the analysis module for vulnerability confirmation. This module will estimate whether the intent counts as a valid attack or not.

However, when we apply these black-box scanners in practice, their performance varies accordingly in different applications. As part of our preliminary evaluation, we used four scanners, BurpSuite, AWVS, Arachni, and ZAP, to test against several websites for a side-by-side comparison, as shown in Table 1.

SEACMS (*SEA* for short) is a web application used for content management. It provides commonly used functions like article publishing, user comments, software download, user management, etc. In this application, BurpSuite [30] detected 37 vulnerabilities, which has the best bug detection capability among the four tools we tested. However, it performed poorly in the MyBloggie application, or shortly *MYB* (a personal blog application), detecting seven fewer security issues

Table 1. Preliminary Experiments Conducted on Three Web Applications with Four Different Scanners

| Tools / Projects | BurpSuite | | | AWVS | | | Arachni | | | ZAP | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *Cov.* | *Req#* | *Vul#* | *Cov.* | *Req#* | *Vul#* | *Cov.* | *Req#* | *Vul#* | *Cov.* | *Req#* | *Vul#* |
| **SEA** | 75.86% | 4,634 | 37 | 49.15% | 736 | 10 | 32.76% | 2,837 | 5 | 24.14% | 159 | 1 |
| **SCH** | 28.13% | 1,376 | 4 | 28.24% | 414 | 2 | 31.25% | 1,944 | 2 | 14.06% | 198 | 2 |
| **MYB** | 77.78% | 4,510 | 4 | 49.21% | 2,906 | 11 | 50.79% | 46,752 | 2 | 77.78% | 289 | 1 |

The rows marked with "Cov." indicate the coverage rate, the rows with "Req#" indicate the numbers of attack requests, and the rows with "Vul#" indicate the numbers of detected vulnerabilities.

than other tools like AWVS. Previous studies like [4, 5] have also shown the existence of such problem when comparing the performance of scanners, which can seriously affect the effectiveness of vulnerability detection.

When we further collected and analyzed the coverage and the number of attack requests, we found that such inconsistency mainly arises from a single scanner's insufficient strategy when constructing attack intents. The existing intent construction mechanism of each scanner is usually designed with preferences. It can be efficient in some applications but may be less effective in others. More specifically, the problem lies in two aspects.

(1) In the first stage of the working cycle, many scanners' detection performance is limited due to their inability to discover more comprehensive website content. The current content discovery mechanism relies on crawling and dictionary-based path enumeration. The crawling component of different scanners may be implemented with different configurations (e.g., depth of tracking, support for different content types). The embedded enumeration dictionaries also vary among scanners. For different sites, the optimal content discovery strategies to achieve better performance are not the same. Using only one strategy for all different test targets is not enough to obtain comprehensive attack surfaces. If the scanner does not cover a relatively complete scope of pages initially, the subsequent process and the detection of vulnerabilities will be limited significantly. However, if we can synchronize attack intents among scanners, a single tool can acquire the attack surfaces explored by multiple strategies. Take the scanner Arachni [34] as an example. When we scan the *SEA* website directly, it can only cover 32.76% of the website, with only five vulnerabilities detected. If we can give Arachni assistance by synchronizing website structure information explored by the other three scanners, its coverage can reach up to 75.86%, while 42 security issues can be identified.

(2) In the second stage, many scanners have a limited and narrow strategy for generating attack vectors. For various test targets, applying only one type of attack vector generation mechanism is not sufficient. If the attack vectors produced by the scanner can be more diverse, there is a greater probability of detecting more vulnerabilities successfully. Synchronizing attack intents allows a single tool to fuse attack vector generation strategies used by multiple scanners. For example, when scanning the site *SCH*, ZAP [18] was able to generate only 198 attack requests and detected two vulnerabilities. However, when we assisted ZAP in synchronizing attack vectors with the other three scanners, it can generate 64 times the number of attack requests and detect 11 vulnerabilities in the site.

Based on these observations, we propose Scanner++, an enhanced web vulnerability detection framework with attack intent synchronization. Firstly, by extracting and refining contents from request packets sent by base scanners, we can aggregate attack surfaces and attack vectors effectively, forming a synchronized attack intent library. Secondly, we design a run-time intent synchronization mechanism. Through analyzing the detection spot of the target, we synchronize related attack intents to the base scanner, thus augmenting its detection process. In this way, we can consolidate attack surfaces explored by different scanners, and share attack vectors produced by them. By

synchronizing attack intents constructed with various mechanisms, the overall scanning process can be more robust and better applicable for diverse real-world targets.

For evaluation, we implemented Scanner++ and chose four high-performance base scanners working under it. We employed benchmark applications that have been widely used in previous research, and three real-world applications in CCDC, one of the largest security depository companies in the world as test-beds. The experiment results demonstrate that the base scanners perform differently on various applications, while Scanner++ consistently and effectively improves their vulnerability detection performance. Specifically, on ten open-source web applications, working under the Scanner++ framework makes BurpSuite [30], AWVS [2], Arachni [34], and ZAP [18] cover 15.26%, 37.14%, 59.21%, 68.54% more pages, construct 12.95×, 1.13×, 15.03×, 52.66× more unique request packets, and discover 77, 55, 77, 176 more bugs, respectively. Moreover, using Scanner++ can help base scanners detect 205 more security issues, even comparing with their combined results. Furthermore, when we applied Scanner++ on the well-tested real-world web applications in CCDC, eight serious previously unknown vulnerabilities were detected and fixed.

Overall, this paper makes the following contributions:

- We propose a fully non-intrusive attack intent synchronization framework for enhanced web vulnerability detection. A proxy-based architecture and a package-based intent synchronization approach are designed in it to perform intent synchronization among multiple scanners.
- We implement Scanner++,[1] where an attack intent purification mechanism and a run-time intent synchronization mechanism are designed to efficiently synchronize base scanners with relevant attack surfaces and vectors. Each scanner can fully utilize various strategies during testing and achieve better vulnerability detection performance.
- We apply Scanner++ to open-source benchmarks and real-world applications. The experiments show that Scanner++ improved the effectiveness of existing scanners greatly, and confirmed many serious previously unknown bugs.

The remainder of the paper is organized as follows: Section 2 presents a background of web application vulnerabilities and scanners. In Section 3, we present an example to illustrate the motivation of intent synchronization. Section 4 demonstrates the architecture and methodology of Scanner++. Section 5 presents the evaluation of Scanner++. Section 6 discusses the potential threats to the validity of Scanner++. Section 7 introduces related work, and we conclude in Section 8.

## 2 BACKGROUND

When a user is interacting with a web application, the browser will send a request packet to a specific path of the server. The request packet usually contains several parameters to put the information that the user wishes to transmit. Then the browser will wait for the response packet from the server and parse the web page contained in it. There are two most commonly used request methods, the GET method and the POST method. One of the main differences lies in the position of parameters in the request packets. The request packets using the GET method will have parameters placed at the link position, while the POST method will put parameters in the body section of the request packets.

Some web applications contain severe security vulnerabilities, resulting in sensitive information leakage, user identity theft, or even leaving the entire server under the control of attackers. The process of exploiting a vulnerability is similar to the normal interaction mentioned above. An

[1]Scanner++ will be open-sourced at: https://github.com/ScannerPlusPlus, and artifacts on multiple platforms are released at https://github.com/ScannerPlusPlus/ScannerPlusPlus/tree/main/Artifacts.

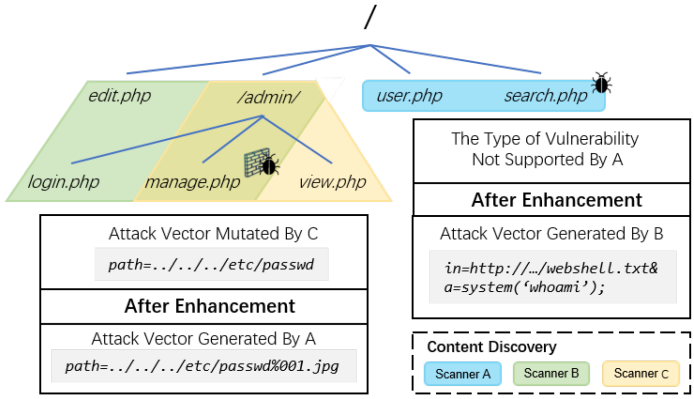


Fig. 1. Motivating example of the enhanced vulnerability scanning of web applications, where the three scanners A, B, and C have different abilities of content discovery and attack vector generation.

attacker will construct request packets with parameters filled with malicious data and send them to a path of the vulnerable host. The attacker then estimates whether they have successfully exploited the vulnerability and gathers the information needed from the response packets. The host of the target server and the corresponding path make up an attack surface, which is the starting point if an invasion happens. The malicious data sent by the hacker is described as attack vector. The attack surface and the corresponding attack vectors constitute of an attack intent. Thus, a complete attack against the target application means to construct a series of valid attack intents. Specifically, it implies that the hacker should find a vulnerable attack surface, generate effective attack vectors, assemble them into request packets, and then send out.

To prevent malicious requests, developers often apply some sanitization measures to user inputs. Sanitization is a set of instruments that estimate whether the input contains attack vectors. Ideally, each user input that may cause an attack will be properly sanitized, but based on previous research [21], reaching such a high standard is challenging. Many attackers will try various methods of mutating attack vectors to bypass such sanitization and achieve successful attacks.

Black-box scanners work essentially by mimicking the process described above. They detect vulnerabilities by automatically constructing attack intents on the target application. First, the scanner gathers as much information about the target's attack surfaces through the content discovery process. The gathered attack surfaces essentially determine the scope of pages that scanners will attempt to attack. The more comprehensive the detection of the attack surfaces, the more effective the scanner will be. Then, for each attack surface, the scanner will generate attack vectors to construct the request packet. The attack vector generation process basically determines the attack capability of the scanners. The more diverse the constructed attack vectors, the more effective the scanner will be. After sending the request, the scanner will analyze the corresponding response to determine whether it constitutes a valid attack. If necessary, it will further mutate the attack vectors to bypass some existing sanitization mechanisms. When the scanner succeeds in composing a valid attack, this indicates that it has found a vulnerability in the website.

## 3 MOTIVATING EXAMPLE

We use a simplified example to illustrate the motivation of enhanced scanning with intent synchronization, as shown in Figure 1. This example is extracted from real-world web applications and the vulnerability detection process of scanners.

Table 2. Vulnerability Detection Capability of the Three Scanners in the Motivating Example

| Tool | The Number of Tested Pages | manage.php | | | search.php | | |
|---|---|---|---|---|---|---|---|
| | | Content Discovery | Attack Vector Generation | Attack Intent Construction | Content Discovery | Attack Vector Generation | Attack Intent Construction |
| *Scanner A* | 2 | □ | ▨ | □ | ■ | □ | □ |
| *Scanner B* | 3 | ■ | □ | □ | □ | ■ | □ |
| *Scanner C* | 2 | ■ | ▨ | □ | □ | □ | □ |
| Work Separately and Combine Results | 6 | ■ | ▨ | □ | ■ | □ | □ |
| Synchonize Intents among Scanners | 6 | ■ | ■ | ■ | ■ | ■ | ■ |

A white box indicates that the scanner is unable to complete the corresponding phase. A hatched box indicates that the attack vector generated by the scanner is invalid. A black box indicates that the scanner is capable of completing the corresponding phase.

The website's root path has three pages (*edit.php*, *user.php*, *search.php*) and an *admin* path, which contains three more subpages (*login.php*, *manage.php*, *view.php*). Among them, *search.php* contains a remote file inclusion vulnerability [36]. *manage.php* has an arbitrary file download vulnerability with a sanitization mechanism to check the extension in input to protect from attack. It will verify whether the requested file is a picture with *jpg* extension. However, the designed mechanism is inadequate, so it is still vulnerable. An attacker can insert NULL character to truncate the path, and harvest the sensitive file content while keeping the extension as *jpg* at the same time. Such insufficient sanitization situation is prevalent in today's web applications.

In each stage of a black-box scanner's work cycle, the strategies often vary in different scanners, leading to inconsistent results. In the first step, the content discovery process, each scanner can detect a portion of the site's structure. *Scanner A* can discover *search.php*, but cannot generate attack vectors against the type of vulnerability hidden on that page. This makes *Scanner A* impossible to discover this vulnerability, as shown in the sixth and seventh columns of the first row in Table 2. Although *Scanner B* supports constructing that kind of attack vectors, it fails to discover the page *search.php* in the first step. The incomplete attack surfaces of *Scanner B* make it unattainable to report the vulnerability as well, as shown in the second row of Table 2. In the second step, attack vector generation, although *Scanner C* generates the attack vector *path=../../../etc/passwd* against the page *manage.php*, but this attack vector is too naive and simple to bypass the website's sanitization mechanism. Thus, *Scanner C* cannot successfully exploit and discover the issue, as shown in the third and fourth columns of the third row in Table 2.

In a word, if we use these three scanners separately, each one can only use its own content discovery approach, thus exploring a part of the attack surfaces in the first stage, as shown in the first row of Figure 2. Subsequently, targeted at that discovered part of attack surfaces, they can only generate attack vectors with their own strategies. Even if we combine the results produced by these three scanners, the overall coverage can be summed up, but in the end, the two security issues still cannot be detected, as shown in the fourth row of Table 2 and the upper part of Figure 2.

Based on this motivation, we design the framework Scanner++, enabling multiple tools to synchronize attack intents during their scanning process. However, unlike many other security testing tools, web vulnerability scanners often operate in a complete closed-loop workflow. Direct interference with its internal state using intrusive methods would severely impair the universality. For open-source scanners, this method will require significant manual labor when integrating new ones. While for proprietary scanners, such a method will be completely unadaptable. How to implement a fully non-intrusive attack intent synchronization framework is the first challenge we need to tackle. Therefore, as shown in Figure 2, we implement a proxy-based architecture and a package-based intent synchronization approach. The former ensures that applying Scanner++

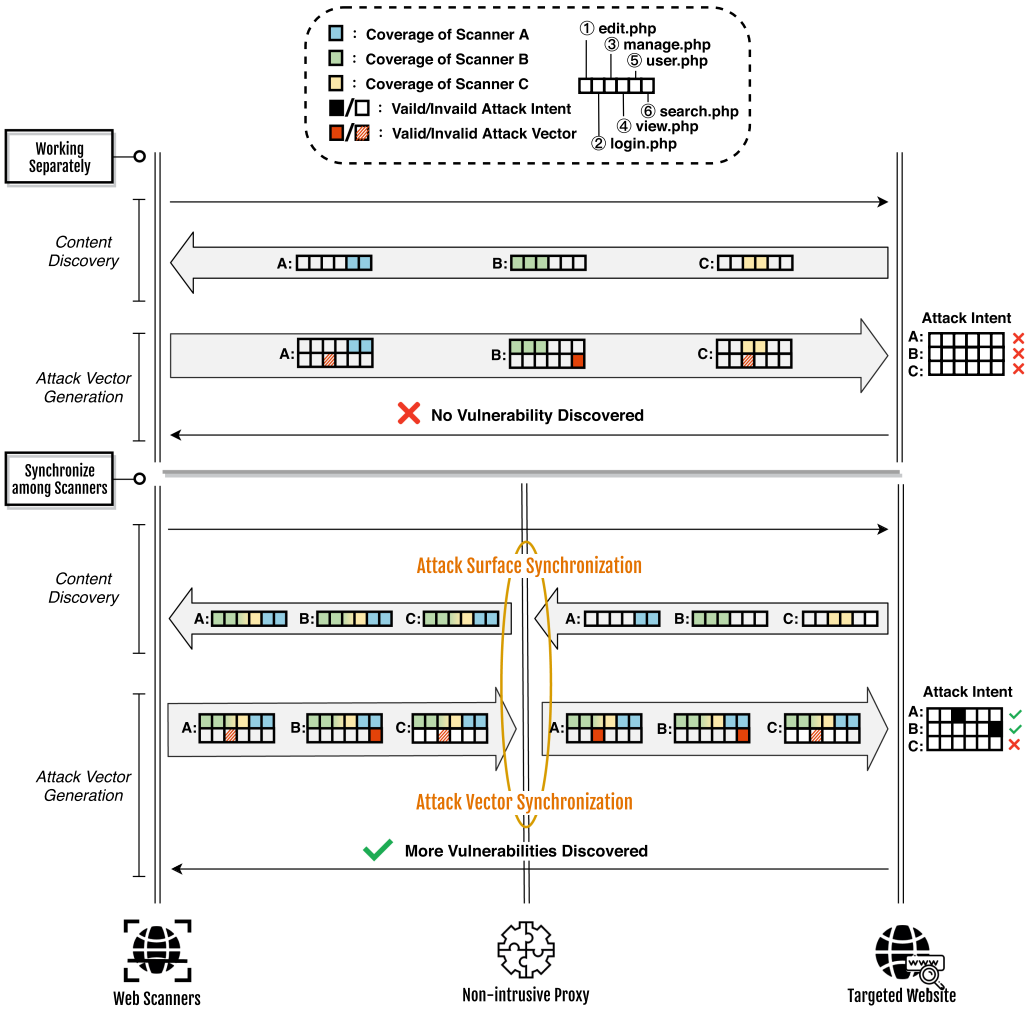


Fig. 2. The illustrative example of Scanner++'s working process. When base scanners work separately, each tool can only discover a part of the website content and generate attack vectors based on their own strategies. The two vulnerabilities cannot be detected. Through attack intent synchronization, each scanner can obtain consolidated attack surfaces discovered by all tools and fusing multiple strategies to generate attack vectors. In the end, the two vulnerabilities can be detected.

does not require any modifications to base scanners, allowing rapid integration of new ones. The latter ensures that Scanner++'s intent-synchronization approach can be applied to all base scanners, as Web applications' interaction is based on the request-response packet exchange process. In this example, Scanner++ will work as a proxy between *Scanner A, B, C*, and the target site. All the attack requests sent by base scanners can be intercepted, and all the responses can be modified by Scanner++.

Originated from this methodology, request and response packets are the appropriate entry points to enhance scanners. We need to make full use of such constrained information to achieve efficient attack intent synchronization. That is the second challenge we need to tackle. Therefore, a purification mechanism and a run-time intent synchronization approach are designed. The

former process aggregates and refines the attack intents extracted from request packets sent by base scanners to construct a synchronized attack intent library. It contains consolidated attack surfaces explored by three scanners during their content discovery process and valuable attack vectors generated based on various strategies. In this way, all the five website pages discovered and the attack vector generated by *Scanner C* on page *manage.php* will be collected. Based on this library, the latter process achieves run-time synchronization by converting and supplementing needed attack intents into corresponding response packets transmitted to base scanners.

In such a manner, *Scanner B* is able to share *Scanner A*'s content discovery results, *B* can reach the page *search.php*. Then *B* can generate an attack vector against it and successfully detect the vulnerability. Meanwhile, *Scanner A* can obtain the attack vector produced by *Scanner C* against *manage.php*, it can further mutate it to produce a more complex attack vector, "*path=../../../etc/passwd%001.jpg*". Such an attack vector has an image extension (i.e., "*.jpg*") that can pass the checks of the sanitization module. However, the file download module reads the file contained in the truncated path (i.e., the path before the NULL character), which is the content of "*/etc/passwd*". In this way, this attack vector can harvest the sensitive file while keeping the extension remains "*jpg*" at the same time. Such a fusing strategy can help them bypass the sanitization mechanism, and detect the vulnerability hidden in *manage.php*.

With Scanner++, three scanners can synchronize their attack intents, and these two vulnerabilities can be detected. By synchronizing the content discovery results, each scanner will be able to obtain the website pages discovered by the others. This can make every scanner form a set of more comprehensive attack surfaces, as shown in the third row of Figure 2. Moreover, by sharing generated attack vectors accordingly, each scanner can reuse, further mutate, and generate more complex and diverse ones. Overall, scanners' test coverage and attacking capability will be significantly enhanced, as shown in the lower part of Figure 2.

## 4 ENHANCED SCANNING ARCHITECTURE

In this section, we will introduce the methodology of Scanner++ to achieve enhanced web vulnerability detection. The framework of purifying and synchronizing attack intents of base scanners is presented in Figure 3.

First, an *Attack Intent Purification* process is conducted. Scanner++ can intercept the original requests from base scanners and extract attack intents. Then it utilizes the intent refinement algorithm to deduplicate and refine the collected intents to construct the *Synchronized Intent Library*. Consisting of a comprehensive *Attack Surface Set* and *Attack Vector Pool*, this library serves as a synchronized summary of the attack intents constructed by each scanner.

To share information effectively between scanners, Scanner++ implements a *Run-time Intent Synchronization* mechanism. In the first step, it selects the relevant attack intents from the synchronized library based on the current scanning position of the target. Then, Scanner++ converts and injects the obtained attack intents into the response packets. Finally, Scanner++ transfers the modified response packets to the base scanners. The supplemented attack surfaces can help the scanner's content discovery process identify a more comprehensive website structure. At the same time, each scanner can use and further mutate the obtained attack vectors to improve the scanner's attack capability.

### 4.1 Attack Intent Purification

The first step in enhanced web vulnerability detection is to extract and refine the attack intents from each scanner's request packets to constitute an attack intent library.

Most scanners work in a closed-loop and do not natively provide an interface to obtain intermediate results (e.g., attack intents). Also, to ensure scalability, we must design a non-intrusive

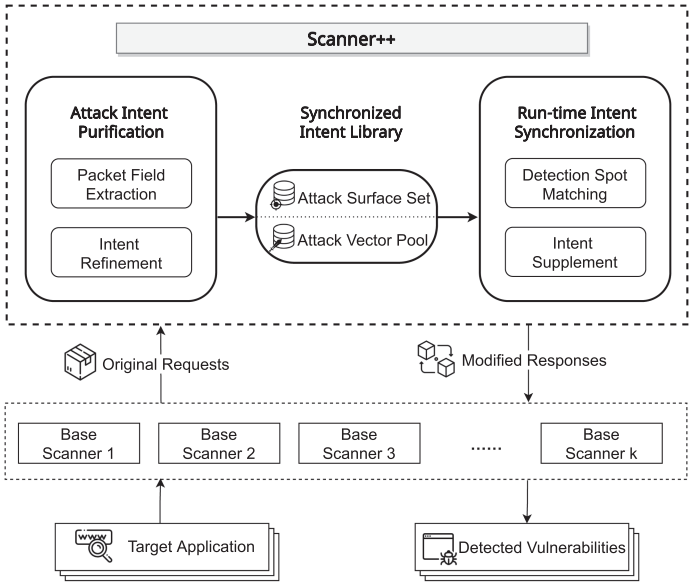


Fig. 3. Overview of Scanner++. The attack intent purification process first extracts and refines attack surfaces and vectors from the request packets sent by base scanners to construct a synchronized attack intent library. The run-time intent synchronization process will then synchronize and supplement related attack intents to the base scanners through response packets.

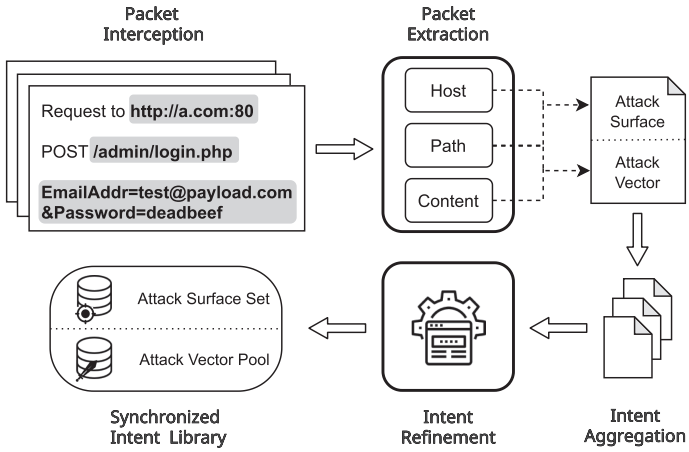


Fig. 4. Request packets sent by base scanners are collected and extracted to obtain attack surfaces and attack vectors, which will be aggregated and refined to construct the synchronized intent library.

method to collect attack intents generated by each scanner. Therefore, we implement the approach of parsing request packets to extract and aggregate attack intents. We intercept all the request packets sent by base scanners, then analyze and extract the host part, path part, and content part out of each request, as shown in Figure 4. A request packet's attack surface is determined by its host and its path, and the attack vector can be described by its path and content. In this way, we can iterate each request packet, and aggregate the attack intents constructed by different scanners.

**ALGORITHM 1:** Attack Vector Refinement Algorithm

**Input**: *AttIntents*[] covers attack intent list extracted from request packets from multiple scanners.
**Output**: *AttIntentLib*[] includes valuable attack intents that will be used in enhanced scanning.

```
1  AttSurfaces[], AttVectors[] ⟵ ExtractFromIntent(AttIntents[]);
2  AttSurfaceSet ⟵ ∅;
3  AttVecPool ⟵ ∅;
4  for Sur in AttSurfaces[] do
5      toSave ⟵ TRUE;
6      for SetAttSur in AttSurfaceSet do
7          if CompareSurface(Sur, SetAttSur) == TRUE then
8              toSave ⟵ FALSE;
9              break;
10     end
11     if toSave == TRUE then
12         EnrichAttackSurfaceSet(AttSurfaceSet, Sur);
13 end
14 for Vec in AttVectors[] do
15     (InPoint, InContent) ⟵ ExtractAttackVector(Vec);
16     AttType ⟵ AnalyseAttackType(InContent);
17     if NewInputPointCovered(AttVecPool, InPoint) then
18         EnrichAttackVecPool(AttVecPool,Vec);
19     else if NewAttTypeDiscovered(AttVecPool, InPoint, AttType) then
20         EnrichAttackVecPool(AttVecPool,Vec);
21     else
22         continue;
23     end
24 end
25 AttIntentLib[] = (AttSurfaceSet, AttVecPool);
```

Exploitations of Web application vulnerabilities have relatively fixed formats, so scanners generally use predefined templates to generate attack vectors. When multiple tools scan the same target, some of the collected attack vectors are not character-wise duplicated, but the semantics are equivalent. Direct synchronization of such attack vectors can lead to poor performance, making vulnerability detection even less effective than the merged results of scanners working in isolation. It is necessary to design a refinement algorithm to reduce such repetitive information and only retain the valuable attack intents. We designed this algorithm based on analyzing input points and attack types, as shown in Algorithm 1. It takes attack intents extracted from multiple scanners as input and output the refined attack intents.

First, it extracts attack surfaces and attack vectors from the input variable, as presented in line 1. Then, it initializes *AttSurfaceSet* and *AttVecPool* as empty sets to store all the unique attack surfaces and attack vectors. To refine the extracted attack surfaces, the algorithm can check the collected information and remove the duplicate data directly based on cross-comparison, as presented in lines 4–13. For refining attack vectors, we need to further estimate their content to guarantee only retaining the valuable ones. As presented in line 15, we first extract the input point and its content from the attack vector. In request packets, input points are determined by the paths and the parameter names, while input contents are the parameter values. Then, we estimate the attack type of the input content. In web applications, exploitations of the same type of vulnerability often have common characteristics. Based on these, we can determine which kind of vulnerability is

targeted by the current attack vector. We summarize the characteristics of 11 common attack types, seven of which are on the OWASP Web Application Security Risk List [26]. The *AnalyseAttackType* function utilizes these exploitation characteristics to parse the content of the requests and evaluate the potential attack type. If the extracted input point has never occurred before, we mark this attack vector as valid since it has discovered a new attack entry. If the input point has been added, we check whether the attack type has already been recorded. If the attack vector is a new type targeted at the input point, *NewAttTypeDiscovered* function will return true, and this attack vector will remain. In this way, we can construct the *AttVecPool*, which contains all the attack entries and exploitations of different types while reducing the duplicate and invalid attack vectors. Using only these valid attack vectors in enhanced scanning can speed up the detection process.

Finally, Scanner++ constructs the attack intent library with a comprehensive attack surface set and attack vector pool. It assembles valuable attack intents constructed by each base scanner, and will be used for the subsequent scanning.

### 4.2 Run-time Intent Synchronization

All the valuable attack intents constructed by different strategies are purified and stored in the attack intent library. We can use it to synchronize the attack intents to base scanners. Considering the closed-loop workflow of scanners, the response packet is the appropriate entry point to supplement the information. Meanwhile, parsing response packets is the critical step for scanners to analyze the target site and decide on the subsequent actions. Appending responses with synchronized attack intents allows the scanner to synthesize the strategies of other tools, improving the potency of generated attacks. To match the scanner's detection progress, Scanner++ tracks the current attack location by analyzing its outgoing request packets and obtains only the relevant attack intents from the synchronized library. Then, Scanner++ converts them into elements that can be recognized by the scanner and injects into the corresponding response packet. Such a run-time synchronization mechanism contains two steps, intent selection and intent injection.

**Intent Selection.** Each scanner tends to set limits on the maximum processing time and the length of response packets. Synchronizing all attack intents into one single page at once would interfere with the normal operation of the scanner. We need to further consider each tool's scanning progress to provide the corresponding scanner with the precise parts of intents it needs. Therefore, according to the scanner's current detection spot, the framework will select only the relevant attack intents from the library at the proper time. To estimate the detection spot of the scanner, we can analyze the sent request packets. Intent selection is based on the "Host" part and the "Path" part in the request packet. According to RFC2616 Section 5 [32], these parts are necessary fields in a request packet. Any valid request packet must contain such information. The host section determines the current target that the tool wishes to scan. The path section determines the specific part of the target it is trying to attack. Therefore, we can obtain the related attack surface information from the synchronized intent library based on the host of the outgoing request packet since it demonstrates the target of scanning. We can also obtain the related attack vectors from the library based on the request's path since it illustrates the specific part of attacking. Through such selection, the relevant intents in the library can be retrieved in real time with the scanning progress of the tool, thus improving the efficiency.

**Intent Injection.** It is difficult for us to interfere with scanners directly in scanning, as the whole process is worked as a closed-loop. To ensure versatility, we need to convert attack intents into a format that all base scanners can process. Therefore, we designed the intent injection process to convert different attack intents to the corresponding HTML elements and supplement them into the response packet.

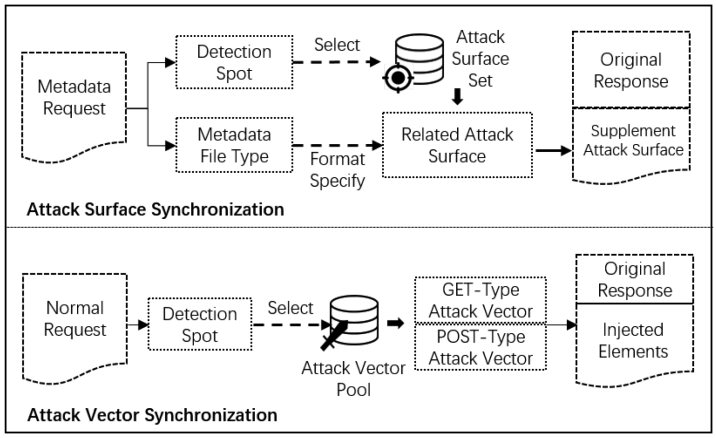


Fig. 5. Related attack surfaces and attack vectors are selected based on the detection spot of the scanner. After converting the intents, it will be injected into the response packets.

First, we need to guide the scanners to use the attack surfaces of the obtained intents during their content discovery, as shown in the upper part of Figure 5. A site's metadata is often stored in files such as *robots.txt*, *sitemap.xml*, etc. Their contents are formatted in a specific way to give information about the website's structure. Scanners often begin their content discovery by requesting site metadata as seed information to assist the whole process. Therefore, we can inject the attack surfaces into these metadata files to guide scanners' content discovery procedure. Whenever a scanner is requesting metadata information, we convert the related attack surfaces into the format specified by the metadata file. Then we supplement them in the corresponding response packet. In this way, the obtained attack surfaces will be a part of the sources that scanner will use to identify website structure. Therefore, a scanner can acquire the attack surfaces in the synchronized library discovered by others, thus improving the test coverage.

Secondly, we also need to guide the scanner to leverage the attack vectors of the obtained intents, as shown in the lower part of Figure 5. We first categorize the related attack vectors into two parts according to the request method, GET-type and POST-type. (1) GET-type requests put parameters' names and values in the link. In HTML specification, a "*href*" element can place a link on the page. Therefore, for each attack vector using the GET request method, we read the input point and the input content to construct a "*href*" element. For example, consider an attack request packet shown in the upper-left corner of Figure 6. This is a typical SQL injection attack that uses the GET method. The attack vector contained in this packet can be converted to the element shown in the upper-right corner of Figure 6. (2) POST-type requests can be sent through a form whose parameters will be the fields of that form. A "*form*" element can store such information on a web page based on HTML standard. For attack vectors that use the POST request method, we construct a "*form*" element. For example, consider an attack request packet shown in the lower-left corner of Figure 6. The attack vector contained in this packet can be extracted and converted to the page element shown in the lower-right corner Figure 6. Then we inject these constructed elements into the response packet and forward it to the scanner.

These converted elements are in accordance with the standard HTML syntactic standard [7]. They are generic and can be correctly parsed by all base scanners. Meanwhile, these elements are converted from attack vectors after refinement. It ensures that few valueless attack vectors will be converted and inserted into pages, preventing scanners from sending duplicate and invalid attacks.

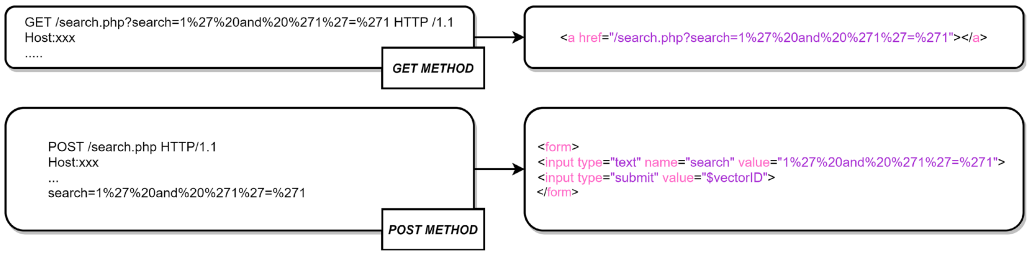


Fig. 6. Two examples about the attack vector injection that uses GET method and POST method.

Therefore, base scanners can share and send attack vectors generated by others through triggering these injected elements, furthermore they can mutate them to construct more complex ones.

Since the scanner's working is motivated by parsing the elements of the response packet, complementing the identifiable elements transformed from the diverse attack intents will optimize the following scanning process. In this way, Scanner++ can synchronize the required attack intents from the library and complement to the scanners at the right time. Each scanner can share other tools' attack intents based on their current needs. Such a run-time synchronization mechanism can expand the attack surfaces and increase the diversity of the generated attack vectors, therefore detecting more vulnerabilities.

### 4.3 Scanner++ Implementation

The proxy part of Scanner++ is implemented based on the *mitmproxy* library, which provides several interfaces for processing HTTP messages. Based on this library, we implemented the packet interception and extraction process to obtain attack intents from request packets. Response packet modification in the intent injection process can also be conducted utilizing this library. The synchronized intent library can be implemented based on various DBMS. The prototype presented in the repository is implemented based on MySQL. This database should have at least four tables that store the GET and POST attack intents before and after refinement. Extracted attack surfaces can be stored either in the database or as a separate file. Scanner++ is directly applicable to most current scanners and has good scalability to integrate more tools, since almost all scanners can be configured with a proxy server in the setup options.

In this paper, we instantiate Scanner++ framework with four base scanners, including two commercial scanners—BurpSuite [30] and Acunetix Web Vulnerability Scanner (AWVS) [2], and two open-source scanners—Arachni [34] and ZAP [18]. They are selected as base scanners for their relatively good vulnerability scanning effect studied in previous performance comparison research like [4, 5, 15], and their wide range of uses. AWVS and BurpSuite are popular commercial scanners, which are among the top ten products in Gartner's Survey [17]. Arachni is a top-performing popular open-source scanner [23]. ZAP is developed by OWASP [27], a famous non-profit organization focusing on web application security. Scanner++ is not limited to these four scanners since other scanners can also be easily integrated by configuring the proxy server option.

Furthermore, we implemented three auxiliary components to make enhanced scanning easier to use.

**Infinite Content Substitution.** Some website contents like calendars or photo galleries, are generated dynamically and infinitely by the same component, but the response packets to such contents are different each time scanners request it. They are likely to be identified as different pages by scanners, tested repeatedly, and analyzed as an infinite sequence of pages. Removing pages containing such content out of the scanning scope can be tedious. Furthermore, removing

the whole page might make scanners miss some security issues hidden in the other parts of that web page. With Scanner++, users can specify such infinite and useless contents with a regular expression prior to scanning. Then, these contents will be automatically removed from all response packets during subsequent process. In this way, with a single configuration, we can avoid meaningless attacks and analysis against these infinite contents while keeping other parts of that web page available.

**Session Sharing Mechanism.** Many sites often require a user login status to use the full functionality. If the scanner performs vulnerability detection without logging in, it will necessarily fail to reach several parts of the target and miss a considerable number of security issues. With the session sharing mechanism, users can set the cookie when visiting the target website in the framework. During the subsequent scanning process, the framework automatically adds the cookie information to the request packets sent by all scanners. In this way, multiple scanners can perform vulnerability detection in the already logged-in context through a single configuration.

**Polluting Requests Blocking.** During the work of the scanner, various request packets are sent out. Some requests can invoke functions like password change or user deletion, which can seriously affect the website's running. To avoid such requests from polluting the target website and causing interruptions in the scanning process, the user can characterize these polluting request packets in the framework. When the scanner sends such requests, the framework will automatically block them. Also, users can use this technique in combination with the session sharing mechanism mentioned above. For example, to avoid the scanner from triggering the logout function that would invalidate a configured session, the user can automatically set the framework to block the logout request packets.

## 5 EVALUATION

To present the effectiveness of enhanced scanning, we conduct thorough evaluations on ten benchmark web applications that have been used as test-beds by previous research like [3, 13, 35]. We also applied Scanner++ to detect vulnerabilities in three real-world web applications of CCDC (Central Depository & Clearing Co., Ltd), an important securities depository company, to verify its performance in industrial practice. We answer the following five questions:

- **RQ1**: Does Scanner++ help base scanners perform better than working separately?
- **RQ2**: Can the refinement algorithm improve the efficiency of enhanced scanning?
- **RQ3**: How is the effectiveness of the synchronized intent library of Scanner++ when enhancing a new scanner?
- **RQ4**: How does Scanner++ perform on real-world vulnerability scanning scenarios?
- **RQ5**: How is the scalability of Scanner++?

**RQ1** focuses on verifying the performance improvement of Scanner++ for the base scanners. We use the coverage rate, number of attack requests, and numbers of detected vulnerabilities as metrics to compare the performance of scanners working with and without Scanner++. **RQ2** is used to evaluate the effectiveness and performance of the intent refinement algorithm. Experiments are conducted to compare attack intents and the performance of scanners with and without intent refinement algorithm. **RQ3** is mainly used to assess the effectiveness of the synchronized intent library of Scanner++. In addition to the four base scanners, we adapt a new scanner, Wapiti, with Scanner++ by leveraging the existing attack intent library. **RQ4** is used to evaluate the performance of Scanner++ in real-world web applications. We also give a case study to illustrate the process of the framework in assisting vulnerability detection. **RQ5** is primarily used to evaluate the scalability of Scanner++. By gradually increasing the scanners involved in the synchronization, we evaluate the performance of Scanner++ under various base tool scenarios.

Table 3. Benchmark Web Applications Used for Evaluation

| Abbr. | Project Name | Description | Version | Lines# |
|---|---|---|---|---|
| SEA | SeaCMS | A software download website. | 1.0 | 99155 |
| SCH | SchoolMate | An alumni contact book. | 1.5.4 | 7445 |
| MYB | myBloggie | A blog website. | 2.1.4 | 7428 |
| OSC | osCommerce | An online shopping website. | 2.3.3 | 57898 |
| GEC | geccbblite | An online message board. | 0.1 | 478 |
| ELE | Elemata | A personal website. | 3.0 | 87410 |
| WAC | Wackopicko | A scanner performance test-bed. | - | 3389 |
| WCH | WebChess | An online chess game. | 0.9 | 5011 |
| SCR | SCARF | Stanford research forum. | - | 1615 |
| FAQ | FAQforge | An FAQ publish website. | 1.3.2 | 1667 |

Column Abbr. means the name abbreviations of projects. Column Line# means the lines of source code in this project.

### 5.1 Data and Environment Setup

Scanner++ and the base scanners (two commercial scanners—BurpSuite [30] and AWVS [2], and two open-source scanners—Arachni [34] and ZAP [18]), were deployed on Ubuntu 20.04 with 16-cores of 2.90GHz each and 16GB RAM.

Firstly, we evaluate Scanner++ on ten benchmark web applications, as shown in Table 3. These applications have been widely used in previous research to compare scanner performance, e.g., [3, 13, 35]. The lines of code range from 478 to 99,155, representing web applications of various sizes. The selected dataset also covers various application categories, such as discussion forums, personal blogs, online shopping sites, online games, etc. We deployed each application on the server, created appropriate user accounts, browsed the application, and submitted necessary forms to ensure that the website is fully functional. Then, we backup the entire website and its database to preserve the original state of the application. After each scanner performing vulnerability detection, the website needs to be restored to ensure a fair comparison. We have uploaded all benchmark applications, related information and a complete end-to-end demonstration to the repository.[2]

Secondly, we use Scanner++ to detect vulnerabilities in real-world web applications to further evaluate its performance in industrial practice. Since a black-box scanner works like a hacker's attacking process, it may affect the target site's normal operation. Therefore, we cannot arbitrarily choose online websites as test targets. After communication and discussion, we decided to cooperate with CCDC, applying this framework to scan their web applications. **China Central Depository & Clearing Co., Ltd.(CCDC)** is one of the largest securities depository companies in the world. At the end of 2020, it had a total of 110 trillion RMB of various assets under its registration and management. There are three target applications for our evaluation, Bond Information Network, Bank Information Registration System, and Bond Information Disclosure System. The main features of test targets are described in Table 4. All of them are well-tested core web applications responsible for trade dealing and financial service. We made a complete mirror deployment of them on the intranet of CCDC. We then installed the framework with base scanners on the intranet for vulnerability detection to avoid affecting real running websites. In this way, we are able to evaluate the effectiveness of the framework in real-world industrial applications.

[2]Applications and configuration demonstrations are uploaded at https://github.com/ScannerPlusPlus/ScannerPlusPlus/tree/main/BenchmarkWebsites.

Table 4. Real-world Web Applications for Evaluation

| Abbr. | Web Application | Description |
|---|---|---|
| BIN | Bond Information Network | Includes bond price updating, dynamic display pages. |
| BIRS | Bank Information Registration System | Includes bank account management, bank information submission functions. |
| BIDS | Bond Information Disclosure System | Contains a series of trade account management, transaction submission functions. |

Column Abbr. means the abbreviation names of the targets.

Table 5. Scan Coverage Reached by Different Base Scanners

| Project | BurpSuite | | AWVS | | Arachni | | ZAP | | Combined | |
|---|---|---|---|---|---|---|---|---|---|---|
| | − | + | − | + | − | + | − | + | - | + |
| SEA | 75.86% | 80.71% | 49.15% | 77.59% | 32.76% | 75.86% | 24.14% | 79.31% | 77.19% | 80.71% |
| SCH | 28.13% | 59.38% | 28.24% | 46.88% | 31.25% | 40.63% | 14.06% | 46.88% | 40.63% | 59.38% |
| MYB | 77.78% | 80.95% | 49.21% | 79.37% | 50.79% | 66.67% | 77.78% | 80.95% | 77.78% | 80.95% |
| OSC | 82.03% | 84.00% | 56.18% | 84.00% | 45.64% | 80.14% | 46.91% | 85.82% | 84.00% | 85.82% |
| GEC | 84.62% | 92.31% | 76.92% | 92.31% | 69.23% | 92.31% | 69.23% | 92.31% | 84.62% | 92.31% |
| ELE | 61.67% | 65.00% | 28.33% | 61.67% | 10.00% | 50.00% | 23.33% | 60.00% | 61.67% | 65.00% |
| WAC | 66.04% | 75.51% | 58.49% | 69.39% | 59.18% | 71.43% | 49.06% | 73.50% | 74.00% | 77.55% |
| WCH | 55.56% | 68.19% | 51.85% | 72.72% | 59.26% | 73.37% | 59.26% | 73.37% | 66.67% | 73.37% |
| SCR | 75.00% | 94.74% | 89.47% | 94.74% | 52.63% | 94.74% | 52.63% | 94.74% | 94.74% | 94.74% |
| FAQ | 78.95% | 89.47% | 68.42% | 84.21% | 47.37% | 84.21% | 47.37% | 94.74% | 84.21% | 94.74% |
| Average Improvement | | +15.26% | | +37.14% | | +59.21% | | +68.54% | | +7.92% |

The columns with "-" present the coverage of the corresponding base scanner working separately without the Scanner++ framework. The columns marked with "+" present the coverage reached by the corresponding scanner working under the Scanner++ framework. The column "Combined" indicates the combination coverage reached by the four scanners.

### 5.2 Evaluation on Benchmark Applications

According to the design of Scanner++, base scanners are enhanced mainly in two aspects, expanding the attack surface and increasing the diversity of attack vectors. In this section, we illustrate the effectiveness in these two aspects and finally compare base scanners' performance working with and without the framework.

The evaluation process is as follows. We first use each of the four base scanners working separately to detect vulnerabilities against the target website. Subsequently, we then have scanners work under Scanner++ to test against target sites. We choose page coverage, unique attack request packets, and vulnerability detection results as metrics. Reported vulnerabilities are verified based on code auditing and exploitation attempt. For each vulnerability detected by the scanner, we conduct a code audit on the reported web page and try to construct a payload based on the reported vulnerability type. A security issue is confirmed if an attack path does exist based on the auditing process and if the vulnerability can be exploited through the attack attempt. The data is presented in Tables 5 and 6.

To evaluate the effect of Scanner++ on expanding each base scanner's attack surfaces, we measured the coverage rate on target websites, as shown in Table 5. The coverage of all tested targets was significantly improved by the enhanced scanning. Compared to the results of each tool

Table 6. The Number of Attack Request Packets Sent by Each Base Scanner

| Project | BurpSuite | | AWVS | | Arachni | | ZAP | | Combined | |
|---|---|---|---|---|---|---|---|---|---|---|
| | − | + | − | + | − | + | − | + | - | + |
| SEA | 4,634 | 19,040 | 736 | 5,124 | 2,837 | 102,711 | 159 | 25,279 | 8,366 | 152,154 |
| SCH | 1,376 | 27,315 | 414 | 2,537 | 1,944 | 162,584 | 198 | 12,749 | 3,932 | 205,185 |
| MYB | 4,510 | 96,839 | 2,906 | 31,351 | 46,752 | 121,169 | 289 | 51,879 | 54,457 | 301,238 |
| OSC | 12,021 | 133,397 | 13,144 | 20,891 | 26,368 | 167,838 | 6,058 | 480,388 | 57,591 | 802,514 |
| GEC | 3,007 | 36,733 | 832 | 7,156 | 2,286 | 344,652 | 625 | 49,794 | 6,750 | 438,335 |
| ELE | 5,268 | 179,873 | 960 | 1,138 | 1,069 | 381,109 | 143 | 101,228 | 7,440 | 663,348 |
| WAC | 7,837 | 91,667 | 4,836 | 8,488 | 32,504 | 484,539 | 1,468 | 2,630 | 46,645 | 587,324 |
| WCH | 13,719 | 74,002 | 3,371 | 6,619 | 38,317 | 251,020 | 4,936 | 50,714 | 60,343 | 382,355 |
| SCR | 4,226 | 143,039 | 29,082 | 35,903 | 32,253 | 379,445 | 4,253 | 70,146 | 69,814 | 628,533 |
| FAQ | 5,719 | 67,494 | 683 | 1,884 | 5,880 | 654,084 | 656 | 163,105 | 12,938 | 886,567 |
| Average Improvement | | +12.95× | | +1.13× | | +15.03× | | +52.66× | | +14.38× |

The columns with "-" present the amount of the corresponding base scanner working separately without the Scanner++ framework. The columns with "+ present the packet amount sent by the corresponding scanner working under the Scanner++ framework. The column named "Combine" shows the total combined amount of request packets of four scanners, working together with or without Scanner++, denoted with "+" and "-", respectively.

working individually, the average coverage of BurpSuite, AWVS, Arachni, and ZAP using Scanner++ increased by 15.26%, 37.14%, 59.21%, and 68.54%, respectively. During the enhanced scanning process, Scanner++ provides the corresponding attack surfaces from the synchronized intent library to base scanners according to their detection spot so as to guide their content discovery process. In this way, each scanner can further perform content discovery based on a set of expansive attack surfaces synchronized from other scanners, culminating in a more complete structure of the target site.

The variety of generated attack vectors determines a scanner's ability to bypass the sanitization mechanism and exploit security issues. The attack vectors will finally be constructed into request packets and sent out. Therefore, the diversity of request packets can reflect the attack capability of the scanners. We compared the number of unique request packets sent by each scanner with and without the Scanner++ framework, as shown in Table 6. In the framework, base scanners can synchronize valuable attack vectors generated by others and further mutate on them to increase the diversity of the final request packets. With Scanner++, the number of unique attack request packets sent by BurpSuite, AWVS, Arachni, and ZAP increased by 12.95×, 1.13×, 15.03× and 52.66×, respectively. Different scanners achieving different increases of attack vectors is mainly due to their various scanning preferences. For example, AWVS is a commercial tool that is often used to scan sites in production directly, so the strategy is more conservative and tends to send fewer request packets. In contrast, ZAP's scanning appears to be more aggressive and will attempt to attack in greater numbers. It is precisely these strategic differences that allow us to make synchronized scanning more effective. Regardless of the scanning preferences, Scanner++ can integrate different strategies of the base scanners and significantly increase the diversity of generated attack vectors, improving their attacking capabilities.

We further verified the vulnerabilities reported by each base scanner with and without Scanner++. For each scanner, the numbers of detected vulnerabilities are shown in Table 7. We can find that Scanner++ can help BurpSuite, AWVS, Arachni, and ZAP identify 77, 55, 77, and 176 additional security issues, respectively. After deduplication and comparison of the detection results of all scanners, the outcome shows that 205 additional unique vulnerabilities can be found using enhanced scanning than simply having multiple scanners working individually and then merging their results. At the same time, as shown in Table 7, the false alarm rates of BurpSuite, AWVS, Arachni, and ZAP has changed by +3.77%, −0.52%, −2.44%, and +9.43% after using Scanner++,

Table 7. The Number of Vulnerabilities Detected by Different Base Scanners

| Project | BurpSuite | | AWVS | | Arachni | | ZAP | | Combined | |
|---|---|---|---|---|---|---|---|---|---|---|
| | − | + | − | + | − | + | − | + | − | + |
| SEA | 37/0 | 44/0 | 10/0 | 15/0 | 5/0 | 42/3 | 1/0 | 20/0 | 45/0 | 73/3 |
| SCH | 4/0 | 18/1 | 2/0 | 20/0 | 2/1 | 11/1 | 2/1 | 11/4 | 6/2 | 32/6 |
| MYB | 4/0 | 15/1 | 11/1 | 17/1 | 2/0 | 6/0 | 1/0 | 25/4 | 15/1 | 40/6 |
| OSC | 4/0 | 8/0 | 0/0 | 3/1 | 0/2 | 2/0 | 0/0 | 40/22 | 4/2 | 47/23 |
| GEC | 4/0 | 9/0 | 1/0 | 10/0 | 4/0 | 10/0 | 1/0 | 6/1 | 6/0 | 16/1 |
| ELE | 1/0 | 5/0 | 8/0 | 8/0 | 1/0 | 8/1 | 1/0 | 29/7 | 8/0 | 32/8 |
| WAC | 3/0 | 8/0 | 5/0 | 5/0 | 4/0 | 5/2 | 2/1 | 2/2 | 6/1 | 10/4 |
| WCH | 4/0 | 10/0 | 10/0 | 13/0 | 11/0 | 12/0 | 6/1 | 8/1 | 13/1 | 16/1 |
| SCR | 1/0 | 17/0 | 17/1 | 20/1 | 6/1 | 8/1 | 6/0 | 35/10 | 20/2 | 50/12 |
| FAQ | 14/0 | 19/4 | 2/0 | 10/0 | 2/0 | 10/1 | 2/0 | 22/3 | 17/0 | 29/5 |
| Total Amount | 76/0 | 153/6 | 66/2 | 121/3 | 37/4 | 114/9 | 22/3 | 198/54 | 140/9 | 345/69 |
| Delta of FP Rate | | +3.77% | | −0.52% | | −2.44% | | +9.43% | | +10.63% |
| Average Improvement | | +77 | | +55 | | +77 | | +176 | | +205 |

Numbers in front of the "/" indicate true positive amounts and numbers behind indicate false positive amounts. The columns with "-" present the data when the scanner works separately without Scanner++. The columns marked with "+" present the data when the scanner works with the Scanner++ framework. The column "Combined" indicates the deduplicated number of unique vulnerabilities reported by the four scanners.

respectively. Such small variation in the false alarm rate is mainly due to the changes in the overall number of vulnerabilities detected and the scope of pages covered. The seriousness of false positives depends on the vulnerability oracle in base scanners' analysis module, which the attack intent synchronization does not interact with. Therefore, from the standpoint of Scanner++'s working principle, it will not be a cause of false alarms for base scanners.

The statistics demonstrate that Scanner++ can expand the attack surfaces, increase the base scanners' ability to generate attack vectors through synchronized intents, and thus improve the base scanners' vulnerability detection performance. It is worth noticing that simply integrating the results of each scanner directly can only achieve limited performance. As shown in the "Combined" columns of Tables 5, 6, and 7, Scanner++ still shows significant improvements compared to direct result combination across scanners. This fully demonstrates that higher coverage, more attack vectors, and better vulnerability detection can be achieved if scanners' attack intents can be efficiently synchronized.

**Answer to RQ1:** Compared to scanners working separately, working under Scanner++ can help scanners reach a higher coverage rate, generate more diverse attack vectors, and detect more web application vulnerabilities.

### 5.3 Efficiency of Attack Intent Refinement

The attack intent library is one of the most critical components of Scanner++ and is used to accomplish synchronization of attack intents among base scanners. The refinement algorithm is the key to ensure that the synchronized information in the library is valid and effective. In this section, we illustrate its effectiveness through experiments.

Since the attack surface refinement is simply just a cross-comparison process, we mainly focus on evaluating the refinement of attack vectors. We counted the number of attack vectors before and after applying the refinement algorithm, as shown in Table 8. In each site tested, an average

Table 8. The Number of Attack Vectors Before and After Using the Refinement Algorithm

| Project | Without Refinement | | With Refinement | | Improve |
|---|---|---|---|---|---|
| | GET | POST | GET | POST | |
| SEA | 833 | 47 | 62 | 5 | 92.39% |
| SCH | 13 | 732 | 2 | 56 | 92.21% |
| MYB | 382 | 514 | 22 | 150 | 80.80% |
| OSC | 44,102 | 6,168 | 2,812 | 599 | 93.21% |
| GEC | 1,108 | 285 | 121 | 8 | 90.74% |
| ELE | 5,339 | 23 | 1,447 | 21 | 72.62% |
| WAC | 12,432 | 567 | 360 | 42 | 96.91% |
| WCH | 1,655 | 470 | 137 | 68 | 90.35% |
| SCR | 4,017 | 1,885 | 227 | 224 | 92.36% |
| FAQ | 4,032 | 371 | 177 | 29 | 95.32% |
| Average | 8497 | | 657 | | 92.26% |

The column "GET" indicates the number of GET-type attack vectors, and the column "POST" indicates the number of POST-type attack vectors.

of 8,497 attack vectors are extracted from the request packets of the base scanner. Among these attack vectors, some are valid, and some are repetitive or ineffective. After applying the refinement algorithm, the number of attack vectors drops to 657, reduced by 92.26% on average. It can be seen that the algorithm can significantly reduce the number of shared attack vectors.

With the significantly reduced attack vectors by refinement, Scanner++ can help detect more vulnerabilities in less time. To evaluate the specific efficiency improvements achieved by the refinement algorithm, we further conducted another comparison of Scanner++ with and without the refinement algorithm. Since some scanners cannot stop running automatically on a few websites when the refinement algorithm is not used, we set each scanner's maximum working time at 24 hours. We compared the number of vulnerabilities detected and the time consumption, as shown in Table 9. With the refinement algorithm, the average time consumption decreases from 32,489 to 14,379, a 55.74% reduction. However, the numbers of detected vulnerabilities increase from 63 to 345.

Compared with the statistics shown in Table 7, without the refinement algorithm, synchronized scanning will not even be as effective as the direct combination of individual results. With the algorithm, however, the time consumption is significantly reduced, but the vulnerability detection performance becomes better. Based on our observation of the scanning process through the experiment, there are three main reasons. First, since no refinement algorithm is used, many attack vectors may be inserted into one page simultaneously, and the response packets become so large that some tools (e.g., Arachni) cannot handle the excessively long responses. This might cause the scanner to immediately stop analyzing the page and miss some security issues. Second, many scanners set limits for the processing and attack time of one page. Without using the algorithm, invalid or duplicate attack intents are injected into the page, misleading the scanner. If the tool wastes time on ineffective attack vectors, the truly valid ones will be discarded and ignored once the threshold is exceeded. Besides, detection of various web vulnerabilities, such as time-based blind SQL injection and time-based command injection, is very time-consuming. Extra useless attack intents can lead to an exponential increase in scan time and eventually lead to timeouts. Third, too many invalid attack intents might cost significant memory resource to store and analyze. Some scanners (e.g., BurpSuite) might stop working once the allocated memory is exhausted.

Table 9. Average Time Consumption and Vulnerabilities Detected by Scanners With and Without the Refinement Algorithm

| Project | Without Refinement | | With Refinement | |
|---|---|---|---|---|
| | Time(s) | Vul | Time(s) | Vul |
| | Avg: 32,489 | Total: 63 | Avg: 14,379 | Total: 345 |
| SEA | 8,574 | 20 | 3,661 | 73 |
| SCH | 12,051 | 5 | 7,572 | 32 |
| MYB | 32,703 | 9 | 9,249 | 40 |
| OSC | 45,280 | 2 | 17,323 | 47 |
| GEC | 22,955 | 2 | 5,516 | 16 |
| ELE | 43,484 | 9 | 30,646 | 32 |
| WAC | 36,868 | 3 | 15,529 | 10 |
| WCH | 12,217 | 7 | 9,796 | 16 |
| SCR | 85,020 | 1 | 23,971 | 50 |
| FAQ | 25,740 | 5 | 20,525 | 29 |

Column "Vul" represents the number of true positive vulnerabilities detected.
Column "Time" column indicates the average seconds the scan took.

Overall, with the attack intent refinement, each tool's time consumption can be greatly reduced, and 282 more vulnerabilities are discovered.

> **Answer to RQ2:** The attack intent refinement algorithm is able to reduce the invalid attack vectors, save scan time significantly, and increase the overall performance of enhanced scanning.

### 5.4 Effectiveness of Synchronized Intent Library

Scanner++'s synchronized intent library is a key component for sharing content discovery and attack vector generation process among multiple scanners. It is sufficiently universal and contains valuable attack intents. When limited time is available, Scanner++ can leverage an existing library to enhance new scanners' vulnerability detection, even if the new tool has not engaged in the construction process of it. With the already-built library, Scanner++ can directly work on the *Run-time Intent Synchronization* stage. It can estimate the current detection spot of the new scanner at run-time, then obtain the needed attack intents from the existing library constructed by other scanners, convert them to page elements that the scanner can recognize, and finally, insert it into the response packet to assist in its detection process. Such flexible use allows Scanner++ to adapt new tools in a much shorter time, letting a new scanner's detection process to be quickly enhanced using the constructed attack intent library.

To further illustrate this, we conducted an experiment to enhance a new scanner, Wapiti [39], with the attack intent library built from the four base scanners, BurpSuite, AWVS, Arachni, and ZAP mentioned above. The experiment results are shown in Table 10. With Scanner++ and the constructed attack intent library, the coverage rate of Wapiti increased by 68.65%, with 28.98× more attack vectors generated on average and 62 more vulnerabilities discovered. This fully demonstrates the effectiveness of the synchronized intent library. Even if its attack intents are obtained from a different set of tools, it can still be directly used to enhance a new scanner with the help of Scanner++. Naturally, if the attack intents generated by Wapiti can be collected and refined to the attack intent library, the four base scanners can be further enhanced by the capabilities of Wapiti.

Table 10. Coverage, the Number of Attack Requests, and the Detected Vulnerabilities of the New Scanner **Wapiti**

| Project | Coverage | | The Number of Requests | | The Number of Vulnerabilities | |
|---|---|---|---|---|---|---|
| | − | + | − | + | − | + |
| SEA | 24.14% | 68.42% | 47 | 26,341 | 0 | 13 |
| SCH | 14.06% | 46.88% | 214 | 14,135 | 2 | 12 |
| MYB | 77.78% | 80.95% | 1,141 | 9,812 | 1 | 5 |
| OSC | 46.91% | 84.00% | 4,937 | 364,975 | 0 | 3 |
| GEC | 53.84% | 76.92% | 498 | 52,315 | 3 | 5 |
| ELE | 10.00% | 30.00% | 221 | 3,504 | 0 | 4 |
| WAC | 49.06% | 75.51% | 5,434 | 14,276 | 1 | 3 |
| WCH | 55.56% | 73.37% | 4,739 | 13,204 | 4 | 8 |
| SCR | 52.63% | 94.74% | 5,299 | 37,844 | 3 | 9 |
| FAQ | 36.84% | 78.95% | 558 | 155,710 | 2 | 16 |
| Average Improvement | | +68.65% | | +28.98× | | +62 |

The columns denoted with "+" show the performance of **Wapiti** with the Scanner++ framework using the already-built intent library. The columns denoted with "-" show the performance of **Wapiti** without the Scanner++ framework.

Table 11. Vulnerabilities Detected by Scanners in CCDC With and Without using the Framework Scanner++

| Target | Without Scanner++ | With Scanner++ |
|---|---|---|
| BIN | Sensitive Information Disclosure(1) | Sensitive Information Disclosure(1), Boolean Type SQL Injection(1) |
| BIRS | XSS(1) | XSS(1), Unrestricted File Upload(1), Arbitrary File Download(2) |
| BIDS | XSS(1) | XSS(1), Path Traversal(1) |
| Total | 3 | 8 |

The number in parentheses indicates the number of vulnerabilities detected in the corresponding category. The row "Total" indicates the number of vulnerabilities detected.

**Answer to RQ3:** The synchronized intent library of Scanner++ is effective and very flexible to use. Even with an existing attack intent library, Scanner++ can still effectively improve the performance of a new scanner.

### 5.5 Performance on Real-world Applications

To evaluate the framework's performance in real industrial practice scenarios, we worked with CCDC, deploying Scanner++ to detect the vulnerabilities in their well-tested web applications. We created a full image copy of three target websites on its intranet and deployed the framework with base scanners. We then detected vulnerabilities with and without the framework, respectively. After that, we deduplicated and verified the detected security issues. Since we do not have access to the source code of their website, we cannot tally specific coverage information. Instead, we use the number of vulnerabilities detected as the metric, as shown in Table 11.

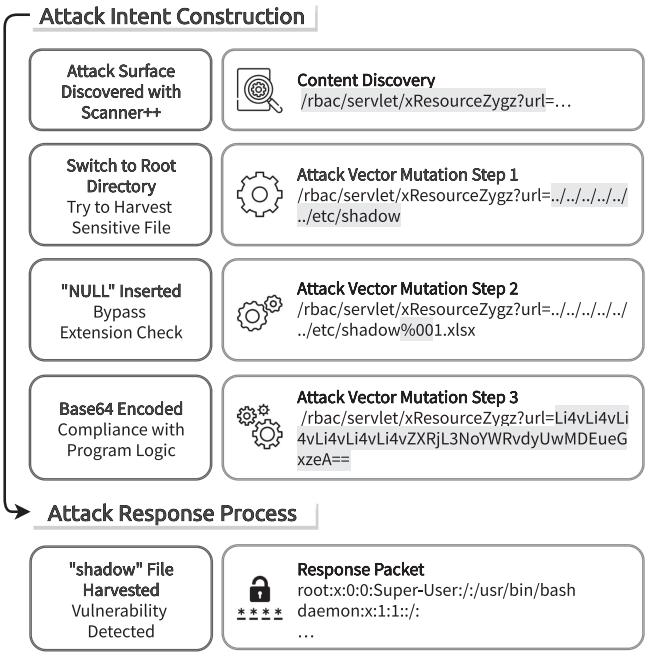


Fig. 7. Detection process of the arbitrary file download vulnerability in the site BIRS.

Without the enhanced scanning framework, the base scanners can only detect three vulnerabilities in the target applications, and the types are relatively simple, only with XSS and sensitive information disclosure. However, with Scanner++, all base scanners can synchronize their attack intents. They can discover more attack surfaces and share valuable attack vectors. This allows the scanners to cover deeper site paths and pages, and generate augmented attack intents to detect a wider variety of covert vulnerabilities. As a result, it detects eight vulnerabilities in the application with six types, including path traversal, XSS, unrestricted file upload, arbitrary file download, sensitive information disclosure, and boolean type SQL injection.

For instance, let us see how Scanner++ helps improve tools' ability to detect one of the arbitrary file download vulnerabilities hidden in the BIRS site, as shown in Figure 7. First, the vulnerable page is quite deep and circuitous in the site structure, making it difficult for a single scanner to find a simple content discovery strategy. With Scanner++'s attack surface sharing, scanners can expand their discovery results and successfully reach the page. Second, to detect the vulnerability, the attack vector must have a NULL character placed at the end of the target file path to bypass the extension check which is used as the sanitization mechanism of the server. Then, the attack vector needs to be converted to base64-encoded format, since the back-end program in BIRS will first decode the parameter content as Base64 text. Such relatively complex attack vector processing prerequisites can make a scanner with one sole mutation strategy fail to detect the vulnerability. However, with Scanner++'s attack vector sharing, the scanner can generate elaborate attack vectors and successfully detect this vulnerability. Ultimately, base scanners with Scanner++'s runtime intent synchronization approach have detected this high-risk security issue that allows an attacker to access arbitrary files on the server.

All these eight vulnerabilities are previously unknown security issues. They have been labelled as high risk by the developers of CCDC and have been fixed accordingly.

**Answer to RQ4:** The enhanced vulnerability detection framework is also effective when scanning real-world web applications, assisting base tools to detect more complex security issues in industrial practice.

### 5.6 Scalability of Scanner++

As an enhancement framework that requires the integration of multiple base scanners, it is an important issue whether Scanner++ is sufficiently scalable. To better illustrate this, we conducted an additional experiment of Scanner++ with four different numbers of base scanners.

Compared with other benchmark applications, four scanners show relatively consistent performance on the *GEC* and *FAQ* application. On such a consistent performance basis, if Scanner++ has significant enhancements when synchronizing different numbers of base tools, it can demonstrate that the framework is applicable with various base scanner scenarios and has great scalability. Execution time, memory consumption, coverage, number of attack request packets and detected vulnerabilities are used as metrics. The results are shown in Table 12.

As presented in the Table, Scanner++ can work well with different numbers of base scanners. As more scanners are involved in synchronization, the coverage rate, the number of attack requests, and the number of detected vulnerabilities gradually increase. In the site *GEC*, the coverage rate of base scanners increased from 69.23% to 92.31%. There is also a significant increase in the number of request packets when more scanners are involved in synchronization. The number of packets generated by ZAP, for example, gradually increased from 691 to 49,794. At the same time, the numbers of detected vulnerabilities by each scanner also improved gradually. For example, when two scanners are involved in synchronization, AWVS is only able to detect six vulnerabilities on the *GEC* site. Then, one more bug can be detected by AWVS when BurpSuite is included into the framework. When all four scanners are used, AWVS is able to report 10 vulnerabilities. The same trend can also be seen in the *FAQ* site. This fully demonstrates that in spite of the numbers of base scanners, Scanner++ can still enhance their performance.

**Answer to RQ5:** Scanner++ has great scalability working with different numbers of base scanners. With more tools involved in the synchronization, the overall performance gradually increased at the same time.

## 6 DISCUSSION

Based on the evaluation of benchmark applications and real-world applications, we demonstrate that Scanner++ helps base scanners perform better. However, some limitations still threaten the usability and performance of enhanced scanning. The main limitations are discussed below.

The first potential threat is the selection of base scanners. Diversified base scanners can facilitate the effectiveness of Scanner++. Not only is this reflected in the types of vulnerabilities supported, but the variety of content discovery and attack vector generation strategies can also make the synchronized scanning better. Subsection 5.1 describes our base scanner selection. We selected these scanners based on previous performance comparative research. Also, there is diversity in the types of vulnerabilities supported by the four base scanners. BurpSuite supports 153 different types of vulnerabilities, Arachni supports 27 types of vulnerabilities, AWVS supports 42 types of vulnerabilities, and ZAP supports 158 types of vulnerabilities. Specific type lists can be found in [1, 20, 28, 31]. However, just because scanners work well on their own does not necessarily mean that they are sufficiently diverse in their strategies. To the best of our knowledge, there has been no previous comparative research of the strategy differences between scanners at each stage. Many

Table 12. Scanner++ Working with Different Numbers of Base Scanners

| Tools | | GEC | | | | | FAQ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **Cov.** | **Req#** | **Vul#** | **Time** | **Mem.** | **Cov.** | **Req#** | **Vul#** | **Time** | **Mem.** |
| One Scanner | ZAP | 69.23% | 691 | 1 | 31 | 113.81 | 47.37% | 657 | 2 | 79 | 116.54 |
| Two Scanners | ZAP | 69.23% | 5815 | 2 | 353 | 127.06 | 68.42% | 12984 | 9 | 1671 | 115.23 |
| | AWVS | 76.92% | 4038 | 6 | 581 | 109.8 | 68.42% | 1673 | 7 | 735 | 104.78 |
| Three Scanners | ZAP | 76.92% | 21849 | 5 | 1332 | 160.75 | 78.95% | 95520 | 18 | 12043 | 181.1 |
| | AWVS | 84.62% | 6104 | 7 | 1493 | 141.02 | 84.21% | 1806 | 10 | 799 | 158.34 |
| | BurpSuite | 84.62% | 28436 | 7 | 1365 | 162.31 | 84.21% | 39469 | 17 | 2649 | 168.11 |
| Four Scanners | ZAP | 92.31% | 49794 | 6 | 1851 | 159.91 | 94.74% | 163105 | 22 | 18352 | 165.36 |
| | AWVS | 92.31% | 7156 | 10 | 1818 | 136.15 | 84.21% | 1884 | 10 | 1034 | 125.69 |
| | BurpSuite | 92.31% | 36733 | 9 | 1777 | 169.8 | 89.47% | 67494 | 19 | 4731 | 184.06 |
| | Arachni | 92.31% | 344652 | 10 | 16620 | 160.47 | 84.21% | 654084 | 10 | 57983 | 169.12 |

Column with "Cov." indicates the coverage achieved by the scanner. Column with "Req#" indicates the number of attack requests sent by the scanner. Column with "Vul#" indicates the number of detected vulnerabilities. Column with "Time" indicates the time consumption(seconds) of the tool scanning the target. The column marked with "Mem." indicates the memory consumption(MB) of Scanner++.

scanner developers are also reluctant to disclose the specific mechanisms they used. Therefore, for the time being, we can only select the scanners based on their performance. A possible solution to this threat is to try more scanners testing against the same site and use the framework to collect and observe the attack surfaces and attack vectors explored and constructed by each one. We can have a grasp on their content discovery strategies and attack vector generation mechanisms by analyzing their intermediate process. We can then further evaluate their diversity based on the result and decide which scanners have more strategy differences. These scanners are suitable to participate in the framework Scanner++.

The second potential threat is that the enhanced scanning process may still not easy enough to use. In Section 4.4, we described three facilities to improve the usability of scanning and reduce the repetitive configuration process of scanners. These facilities can handle common scenarios that require unique settings for individual scanners and improve preliminary preparation efficiency. However, the usability can still be improved for better scalability on various applications and different scan tasks. For example, we can integrate some APIs that call scanner functions into the framework. The user can then operate the framework, and the framework translates the corresponding operation into the API of each scanner to call the corresponding function. This allows the framework to provide a unified access portal to the user.

The third potential threat to validity is the overhead of enhanced scanning. Based on our experiment, Scanner++ itself does not have much impact on the base scanners' efficiency, as the number of attack requests per unit time does not decrease significantly. The number of attack requests sent per second only changes from 10.81 to 9.87, which is only a 8.70% decrease, demonstrating Scanner++'s low overhead despite the added mechanisms. Meanwhile, it takes only 1.31 seconds to refine 8,479 attack vectors on average, demonstrating the high efficiency of the refinement algorithm. The process of intent injection is essentially converting attack intents into website metadata or page elements and appending them to the end of the response packet. The conversion process follows a predefined template and is therefore very fast. Based on our experiment, the conversion time of 5,367 GET-type attack vectors and 1,202 POST-type attack vectors (all the attack vectors collected from ten benchmark websites) is only 0.0061 seconds and 0.0512 seconds, respectively. Each intent injection only needs a part of these vectors, so it will not slow down the working process. These statistics demonstrate that intent injection will not affect the efficiency of base scanners. It

is worth noting that using the total execution time of the tool to measure the additional overhead of efficiency is not an appropriate metric. Unlike some other dynamic testing tools (e.g., fuzzers) that run continuously, web application scanners automatically stop after iterating through all explored paths and predefined attack vector generation strategies. Lower coverage and fewer attack vectors of scanners working alone will result in shorter runtime, while its effectiveness will be limited. But with Scanner++, along with a wider range of attack surfaces and more vectors, we can accomplish those increasing number of valid attacks or requests with more scanning time, thus detecting more vulnerabilities. In the future, we can make better use of the distributed scanning capabilities of some base scanners, deploying Scanner++ in a distributed manner to further improve the efficiency of enhanced scanning through parallelization.

The fourth potential threat is the manual efforts needed for adapting Scanner++ to the new scanners. As a non-intrusive proxy running between base scanners and target applications, Scanner++ can be easily applied to most black-box scanners. Adapting new scanners only requires configuring their proxy server. The vast majority of scanners provide a direct interface for this. For the few scanners that do not provide an interface, adaptation can be made using a simple external proxy tool such as ProxyChains. Once configured the proxy settings, Scanner++ can monitor every exchanged packet, fetch and complement information of base scanners without any modification.

## 7 RELATED WORK

**Vulnerability Detection on Web Applications.** With increasing attacks on web applications, many researchers have developed web vulnerability detection tools.

White-box vulnerability detection tools use techniques like static analysis and symbolic execution to analyze the source code of web applications. Among them, RIPS [33] is one of the most widely used scanners in industrial practice. Dahse and Holz [11] used intra- and inter-procedural analysis to model application in PHP language and uses taint analysis to detect vulnerabilities. phpSAFE [25] constructed a program model with AST and supported analyzing web applications developed with OOP. Livshits et al. [22] used specifications which were provided by users to find security issues in Java web applications.

Black-box vulnerability detection tools can work without the source code of applications and check security issues by simulating the attacking process. Commercial scanners like BurpSuite [30], AWVS [2], and Nessus [38] are the most common testing tools used by companies. A few open-source scanners, like Arachni [34], ZAP [18], and Wapiti [39] also perform quite well in vulnerability detection. In addition to these tools, many researchers are trying to optimize black-box scanners to detect more kinds of vulnerabilities. In [12], authors proposed a state-aware scanner, by inferring the application's internal state to achieve higher coverage and assist the detection process. In [14], authors used genetic algorithm to generate attack requests and detect cross site scripting vulnerabilities in web applications. In [8], authors focused on detecting inconsistent input validation between the front-end and back-end, using black-box detection methodology to examine parameter tampering opportunities in web applications. In [37], authors used taint tracking to scan persistent client-side XSS in the wild. Some research also studied the hybrid black-box and white-box approaches to detect vulnerabilities. For example, Navex [3] used static analysis to construct a property graph and combines dynamic analysis to scan PHP web application vulnerabilities. Saner [6] used static analysis technique to model the sanitization process, and composed it with dynamic analysis to execute sanitization code on malicious inputs to detect improper sanitization procedures.

Unlike previous works, we are not proposing a new concrete scanner. Instead, we systematically study the framework of enhanced scanning with attack intent oriented synchronization. Black-box scanners can be integrated into our framework and performs better than working separately.

**Enhance Vulnerability Detection with Multiple Tools.** There are some research working on fusing multiple tools for vulnerability detection. Some works tries to combine the results of single tools directly to reduce the false negative. For example, SmartBugs [16] combines the results of multiple vulnerability detection tools to perform security checks on smart contracts. Nunes et al. [24] combine the union of results from several static analysis tools (i.e., white-box scanners) to detect security issues in PHP-based web applications. Different from those work, Scanner++ supports interaction of different scanners during the whole scanning process and can enhance multiple black-box scanners without access to the source code, thus supporting web applications developed in any languages.

In fuzzing of libraries, EnFuzz [9] integrated diverse fuzzers with global seed pool to achieve higher coverage and bug discovery. Based on EnFuzz, Cupid [19] designed a complementary selecting strategy to automatically combine fuzzers and improve the final coverage. Different from fuzzers, black-box web vulnerability scanners work in distinct ways and solve different problems. Fuzzers randomly mutate and generate invalid data as test cases to detect bugs in programs and libraries. In comparison, scanners need to interact with web applications based on the request-response model and generate highly structural test cases based on vulnerability exploitation templates to detect security issues like SQL injection. Therefore, fuzzers like AFL cannot be used to detect web application vulnerabilities.

Due to the differences of scanners and fuzzers above, the research challenges and target domain of Scanner++ and cooperating fuzzers like EnFuzz are significantly different. Scanner++ is a non-intrusive black-box intent synchronization framework targeted at web application vulnerabilities. Since black-box scanners work in a closed-loop manner, we adopted a proxy-based working architecture and a package-based intent synchronization approach to ensure scalability. Ensemble fuzzers like Enfuzz and Cupid are intrusive grey-box fuzzing integration frameworks targeted at bugs in programs or libraries. They enhance base fuzzers' coverage through seed synchronization.

Therefore, Scanner++ is entirely incomparable with cooperating fuzzers because of the following reasons. First, integration strategies adopted by cooperating fuzzers cannot apply to scanners. Unlike fuzzers, most scanners work in a complete closed-loop manner. Any intrusive methods designed in cooperating fuzzers will severely change base scanners' original workflow. Second, web application vulnerabilities cannot be detected by cooperating fuzzers like EnFuzz and Cupid. They can only be detected through highly structural test cases generated based on exploitation characteristics and mutated based on semantic equivalent substitution. Meanwhile, since most vulnerabilities in web applications will not trigger crashes, they can only be identified by parsing response packets. Therefore, fuzzers, based on random mutation strategy and crash-based oracle, cannot identify web application vulnerabilities.

## 8 CONCLUSION

In this paper, we systematically investigate the idea of attack intent synchronization to enhance the vulnerability detection of web applications. Scanner++ improves the capability of existing scanners by synchronizing valuable attack intents and supplementing related ones among different base scanners according to their detection spots at run-time. First, this framework assists the scanners' content discovery process by providing related attack surfaces identified by each other. In this way, scanners can obtain a more comprehensive site structure and achieve higher coverage. Second, it enhances scanners' attack vector generation process by sharing related attack vectors generated by each other. Scanners can be guided to mutate and generate more complex attack vectors, thus increasing their attack capabilities. Based on our evaluation, Scanner++ helps popular base scanners perform better in terms of coverage, unique request amount and detected vulnerabilities on benchmark applications. On real-world web applications used in CCDC, we have found

eight serious previously unknown vulnerabilities. Moreover, Scanner++ can be easily utilized to integrate more base scanners for industrial practice.

Our future work mainly focuses on three aspects: the first is to evaluate the diversity of different base scanners more systematically and try to choose more diverse ones for the enhanced scanning framework; the second is to design more advanced attack intent purification mechanism and run-time intent synchronization mechanism to further optimize the performance; the third is to upgrade the usability of the framework and design more auxiliary facilities to reduce time consumption of the pre-configuration process.

## APPENDICES

## A EXPLOITATION OF THE VULNERABILITY DETECTED IN CCDC

Vulnerabilites in the tested real-world applications can cause a great deal of damage. As an example, we illustrate in more detail how the arbitrary file download vulnerability found by Scanner++ in the application BIRS can be exploited. As shown in Figure 8, we further attempted to exploit this vulnerability in three ways. (A) First, we harvested all the source code of the web application with this vulnerability. For attackers, trying to compromise the application based on analyzing the source code will make vulnerability discovery and exploitation much easier. (B) Subsequently, we tried to obtain configuration files of the web application. Checking through the harvested files, we found that the needed credential to connect to the database is exposed. With this information, we successfully connected to the Oracle database attached to the application. A malicious attacker may tamper with sensitive data stored in the database directly or with an SSRF vulnerability. With accessing the database, we used system command execution functions (e.g., *DBMS_EXPORT_EXTENSION* in *Oracle DBMS*) to get a reverse shell and gain control of the server. (C) In addition, many web applications are hosted on servers that have multiple network interfaces. The server used by BIRS has two network interfaces, one connected to the public network to receive requests from users and the other connected to the intranet to obtain necessary information from other internal hosts. We harvested network-related files such as */proc/net/arp*, */etc/network/interfaces* to probe internal network structure and gather information of other internal hosts. Since the internal systems are normally protected by the network topology, they often have a weaker security posture. Considering that the authorization scope of our study is limited to this server, we did not go further trying to compromise intranet hosts. For an attacker, this server can be used as a perfect entry point to carry out attacks on other hosts in the intranet.

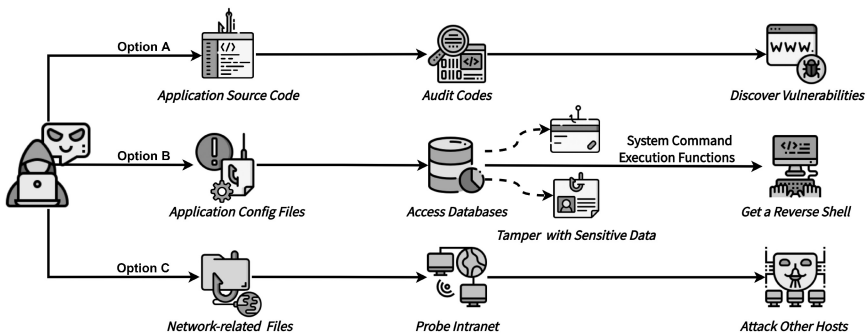


Fig. 8. Exploitation attempt of the arbitrary file download vulnerability detected in the site BIRS. Attackers can harvest application source code, configuration files or network-related files to discover vulnerabilities, get a reverse shell, and compromise other hosts in the intranet.

## B ENGINEERING EFFORTS REQUIRED TO ADD NEW SCANNERS

Since Scanner++ uses a proxy architecture, adapting new scanners only requires configuring their proxy server. Therefore, little manual efforts are needed to add new scanners to Scanner++. In the repository, we provide a detailed procedure for configuring proxies for several popular scanners. Also, we list the configuration process of the tool ProxyChains as a general proxy configuration scheme.

To further demonstrate this, we invited five software engineers from CCDC to configure five base scanners, BurpSuite, AWVS, Arachni, ZAP, and Wapiti, working under Scanner++. We recorded the time consumption of the configuring process, as shown in Table 13.

Table 13. Time Consumption of Five Software Engineers Adapting Base Scanners Working Under Scanner++

| Engineer ID | Time Consumption (Seconds) | | | | |
|---|---|---|---|---|---|
| | BurpSuite | AWVS | Arachni | ZAP | Wapiti |
| 1 | 214 | 62 | 234 | 204 | 37 |
| 2 | 267 | 84 | 241 | 247 | 46 |
| 3 | 231 | 71 | 187 | 215 | 35 |
| 4 | 195 | 72 | 201 | 186 | 28 |
| 5 | 209 | 63 | 192 | 196 | 34 |
| Average | 223.2 | 70.4 | 211 | 209.6 | 36 |

The row "Average" indicates the average time consumption.

Considering the different configuration process of base scanners, the average preparation time of adapting the tool varies from 36 seconds to 223.2 seconds. Meanwhile, all software engineers can prepare base scanners for working under Scanner++ within five minutes. This fully illustrates that Scanner++ can be quickly applied to new scanners.

## REFERENCES


[1] Acunetix. 2021. Vulnerabilities - Acunetix. https://www.acunetix.com/vulnerabilities/web/.

[2] Acunetix. 2021. Web Application Security with Acunetix Web Vulnerability Scanner. https://www.acunetix.com/vulnerability-scanner/.

[3] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V. N. Venkatakrishnan. 2018. NAVEX: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15–17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 377–392. https://www.usenix.org/conference/usenixsecurity18/presentation/alhuzali.

[4] Mansour Alsaleh, Noura Alomar, Monirah Alshreef, Abdulrahman Alarifi, and AbdulMalik Al-Salman. 2017. Performance-based comparative assessment of open source web vulnerability scanners. *Security and Communication Networks* 2017 (5 2017), 1–14. https://doi.org/10.1155/2017/6158107

[5] Richard Amankwah, Jinfu Chen, Patrick Kwaku Kudjo, and Dave Towey. 2020. An empirical comparison of commercial and open-source web vulnerability scanners. *Software: Practice and Experience* 50, 9 (2020), 1842–1857. https://doi.org/10.1002/spe.2870 arXiv: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2870..

[6] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. 2008. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 387–401. https://doi.org/10.1109/SP.2008.22

[7] T. Berners-Lee and D. Connolly. 1995. RFC1866: Hypertext Markup Language - 2.0.

[8] Prithvi Bisht, Timothy Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and V. N. Venkatakrishnan. 2010. NoTamper: Automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS'10)*. Association for Computing Machinery, New York, NY, USA, 607–618. https://doi.org/10.1145/1866307.1866375

[9] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX*

*Security 19)*. USENIX Association, Santa Clara, CA, 1967–1983. https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang.

[10] The MITRE Corporation. 2020. CVE - Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.

[11] Johannes Dahse and Thorsten Holz. 2014. Simulation of built-in PHP features for precise static code analysis. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23–26, 2014*. The Internet Society. https://www.ndss-symposium.org/ndss2014/simulation-built-php-features-precise-static-code-analysis.

[12] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the state: A state-aware black-box web vulnerability scanner. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 523–538. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe.

[13] Adam Doupé, Marco Cova, and Giovanni Vigna. 2010. Why Johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Christian Kreibich and Marko Jahnke (Eds.). Springer Berlin, 111–131.

[14] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. 2014. KameleonFuzz: Evolutionary fuzzing for black-box XSS detection. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy* (San Antonio, Texas, USA) *(CODASPY'14)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/2557547.2557550

[15] Damiano Esposito, Marc Rennhard, Lukas Ruf, and Arno Wagner. 2018. Exploiting the potential of web application vulnerability scanning. In *ICIMP 2018 the Thirteenth International Conference on Internet Monitoring and Protection, Barcelona, Spain, 22–26 July 2018*. IARIA, 22–29.

[16] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu. 2020. SmartBugs: A framework to analyze solidity smart contracts. (Sep. 2020), 1349–1352.

[17] Gartner. 2021. Application Security Testing Tools Reviews 2021 | Gartner Peer Insights. https://www.gartner.com/reviews/market/application-security-testing.

[18] OWASP ZAP Attack Proxy. 2021. OWASP. https://www.zapproxy.org/.

[19] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. 2020. Cupid: Automatic fuzzer selection for collaborative fuzzing. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC'20)*. Association for Computing Machinery, New York, NY, USA, 360–372. https://doi.org/10.1145/3427228.3427266

[20] Tasos Laskos. 2014. Supported Features - Arachni. https://www.arachni-scanner.com/features/framework/{#}Checkshttp://www.arachni-scanner.com/.

[21] Benjamin Livshits and Stephen Chong. 2013. Towards fully automatic placement of security sanitizers and declassifiers. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL'13)*. Association for Computing Machinery, New York, NY, USA, 385–398. https://doi.org/10.1145/2429069.2429115

[22] V. Benjamin Livshits and Monica S. Lam. 2005. Finding security errors in Java programs with static analysis. *Proc. Usenix Security Symposium* (2005), 271–286. https://www.usenix.org/legacy/publications/library/proceedings/sec05/tech/full{_}papers/livshits/livshits.pdf.

[23] Sectool Market. 2016. The Prices vs. Features of Web Application Vulnerability Scanners. http://www.sectoolmarket.com/price-and-feature-comparison-of-web-application-scanners-opensource-list.html.

[24] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, and M. Vieira. 2017. On combining diverse static analysis tools for web security: An empirical study. In *2017 13th European Dependable Computing Conference (EDCC)*. 121–128. https://doi.org/10.1109/EDCC.2017.16

[25] P. J. C. Nunes, J. Fonseca, and M. Vieira. 2015. phpSAFE: A security analysis tool for OOP web application plugins. (June 2015), 299–306. https://doi.org/10.1109/DSN.2015.16

[26] OWASP. 2021. OWASP Top Ten Web Application Security Risks | OWASP. https://owasp.org/www-project-top-ten/.

[27] OWASP Foundation. 2021. Open Web Application Security Project. https://owasp.org/.

[28] OWASP. 2021. Supported Vulnerabilities - OWASP ZAP. https://www.zaproxy.org/docs/alerts/.

[29] PCI SECURITY STANDARDS. 2017. Approved Scanning Vendors. https://www.pcisecuritystandards.org/assessors_and_solutions/approved_scanning_vendorshttps://www.pcisecuritystandards.org/documents/ASV_Program_Guide_v3.0.pdf.

[30] PortSwigger. 2020. Burp Suite - Application Security Testing Software. https://portswigger.net/burp.

[31] PortSwigger. 2021. Issue Definitions - PortSwigger. https://portswigger.net/kb/issues.

[32] RFC2616. 1999. HTTP/1.1: Request. https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html.

[33] RIPSTECH. 2020. RIPS - The Technology Leader in Static Application Security Testing. https://www.ripstech.com/.

[34] Sarosys LLC. 2017. Arachni - Web Application Security Scanner Framework. https://www.arachni-scanner.com/.

[35] Lim Kah Seng, Norafida Ithnin, and Syed Zainudeen Mohd Said. 2018. The approaches to quantify web application security scanners quality: A review. *International Journal of Advanced Computer Research* 8, 38 (2018), 285–312. https://doi.org/10.19101/IJACR.2018.838012

[36] H. Shahriar, M. A. I. Talukder, M. Rahman, H. Chi, S. Ahamed, and F. Wu. 2019. Hands-on file inclusion vulnerablity and proactive control for secure software development. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 2. 604–609. https://doi.org/10.1109/COMPSAC.2019.10274

[37] Marius Steffens, Christian Rossow, Martin Johns, and Ben Stock. 2019. Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24–27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/dont-trust-the-locals-investigating-the-prevalence-of-persistent-client-side-cross-site-scripting-in-the-wild/.

[38] Tenable. 2020. Nessus Vulnerability Assessment. https://www.tenable.com/products/nessus.

[39] Wapiti. 2021. Wapiti : A Free and Open-Source web-application vulnerability scanner in Python for Windows, Linux, BSD, OSX. https://wapiti.sourceforge.io/.

Received 24 June 2021; revised 29 January 2022; accepted 3 January 2022