Web-Armour: Mitigating Reconnaissance and Vulnerability Scanning with Scan-Impeding Delays in Web Deployments

Yousif Dafalla
EECS and I2S
University of Kansas
Lawrence, KS; USA
yousif.dafalla@ku.edu

Dalton A. Brucker-Hahn
Sandia National Laboratories
Livermore, CA; USA
dabruck@sandia.gov

Drew Davidson
EECS and I2S
University of Kansas
Lawrence, KS; USA
drew.davidson@ku.edu

Alexandru G. Bardas
EECS and I2S
University of Kansas
Lawrence, KS; USA
alexbardas@ku.edu

***Abstract*—Reconnaissance is a critical phase in many cyber attacks. Vulnerability scanning, a key component of reconnaissance, has been shown to be a widespread phenomenon on the internet and commonly targets web application/server deployments. By increasing the costs for vulnerability scanning, many of these attacks may be deterred or even prevented, especially for large-scale, internet-wide campaigns.**

In this paper, we propose *Web-Armour*, a mitigation approach to adversarial reconnaissance. Operating as a delay injection mechanism to infrequently executed code portions of a web deployment, Web-Armour significantly increases the cost for attackers to perform automated reconnaissance and vulnerability scanning, while introducing minimal to negligible impact for benign users. We evaluated Web-Armour in a live environment, operated by real users, and in controlled (offline) scenarios. Using Web-Armour, our results show that automated scanning tools may require up to *396 times longer* in an offline setting, and up to *357 times longer* in a real-world operational deployment to complete compared to unprotected installations. In many instances, scanning tools fail to complete their tasks, due to request timeouts. Furthermore, the performance overhead incurred to benign users is minimal, and can be as low as a *0.6%* increase over the baseline.

***Index Terms*—Vulnerability Scanning, Reconnaissance, Delay Injections, Web Security.**

1. Introduction

Automated scanning of hosts on the internet for vulnerable services is a key step in most of today's cyber attacks. Previous studies have shown the proliferation of both internet wide-scans and targeted scans [1]–[3]. These studies show that every public-facing IPv4 address on the internet is under continuous scans, resulting in a huge amount of scanning traffic. For example, Richter, *et al*. [3] examined the firewall traffic logs of approximately 89,000 servers affiliated with a Content Delivery Network (CDN). During this investigation, they discovered that 87% of the logged traffic was a result of scanning activities. A significant portion of this scanning traffic targets web deployments [3]. Scanning is usually performed by automated tools and attempts to identify vulnerable services in order to exploit them [4]. Given the importance of scanning to cyber attack campaigns, raising the bar and making it more difficult for adversaries to obtain meaningful results from web scanning activities, could prove highly beneficial to the security of individual web deployments and to the state of web security overall.

Previous work, such as [5], [6], shows that vulnerabilities are often found in dead, unused, or infrequently executed chunks of code. These portions of code are rarely executed during the benign use of the application, but are primary targets for attack traffic. Therefore, one approach that has been proposed, known as debloating [5], aims to discover and remove this unused code from the application layer of a web deployment. However, this approach suffers from a major drawback in that it may break application functionality if benign usage (profiling) is inaccurate/incomplete.

To overcome the major debloating challenge of consistently needing perfect usage profiles, seldom-used code should perhaps not be removed but be hard to reach. Specifically, the ability for adversaries to explore seldom-used portions of code should be limited. Thus, we explore the use of *delay injection* - adding artificial delays into source code - as a mechanism to limit access to infrequently used portions of code in a web deployment. This expands upon the growing body of work that demonstrates the benefits of delay injections [6], but our approach is specialized to the particularly-applicable domain of web security. In this domain, the adversary has a very limited interface to the attack target and timeouts can be leveraged as an effective defense. In short, timeouts can cause adversaries' requests to fail or obtain incomplete information when appropriate amounts of delays are enforced.

In this paper, we propose a novel framework, called *Web-Armour*, that hinders attackers from effectively scanning web deployments. Our technique aims to severely slow down, or in some cases, timeout requests during reconnaissance and vulnerability scanning activities. Web-Armour accomplishes these goals by injecting scan-impeding delays to *cold code* of a web deployment (both, at the web server layer and at the web application layer).

Funding agency: National Science Foundation

Intuitively, we view cold code as portions of the source code that benign program execution rarely reaches, but reconnaissance and web vulnerability scanning tools frequently visit. To identify cold code, Web-Armour employs a blend of profiling techniques to gather benign usage profiles for a web deployment. These methods aim to thoroughly exercise the benign functionality of the web deployment by extensively emulating benign user behavior with the targeted deployment while dynamically collecting code-coverage information. Leveraging this information, frequently used server-side code is differentiated from rarely used code portions. This way Web-Armour can introduce scan-impeding delays to cold code in the web deployment, at the web server and/or the web application layer. It is worth noting that benign usage profiles do not need to be perfect and Web-Armour can be effective even with imperfect profiles. To the best of our knowledge, Web-Armour is the first approach that utilizes delay injection techniques in web application/server deployments to mitigate automated reconnaissance and web vulnerability scanning.

While Web-Armour does not completely stop web scanning, it significantly increases the cost for adversaries when using automated reconnaissance and web vulnerability scanning tools en masse, meaning that attackers will need to spend significantly more time and effort to find fewer bugs and vulnerabilities. Our proposed approach supports strategies emphasized by US federal agencies such as the National Security Agency (NSA) that acknowledges that it may not be feasible to completely stop adversaries, but rather the goal is to impede, disrupt, and increase the cost for adversaries to accomplish their malicious activities [7], [8].

We evaluated Web-Armour on the two most popular web servers, Apache and Nginx which are used by more than 64% of all web deployments on the internet [9]. Furthermore, we conducted an assessment of Web-Armour on a live real-world website at an educational institution in the United States of America. Moreover, we examine Web-Armour in conjunction with WordPress, the most popular content management system used on the internet [10].

After applying Web-Armour in online (real-world) and offline (controlled) scenarios, we find that automated scanning tools take significantly longer (up to 396 times longer than the baseline) and often scans fail entirely (i.e., timeouts occur). Furthermore, the performance overhead incurred to benign users of a Web-Armour-active deployment is minimal, and can be as low as a 0.6% increase over the baseline. These results indicate that Web-Armour can be effective in thwarting automated vulnerability scanning against web deployments. The contributions of this work are as follows:

- **Explore delay injection techniques in web deployments:** We explore the efficacy of using delay injections as a defense mechanism to thwart adversarial reconnaissance in a variety of web deployments. Our study includes injecting delays using both online and offline profiling, in a variety of application types, and at different levels of the web stack (i.e. at both the web server and/or the web application layer).
- **A novel delay-injection-based framework called Web-Armour:** We introduce Web-Armour, a framework that leverages profiling and instrumentation for web servers and web applications to delay infrequently used portions of code in a web deployment. We evaluate Web-Armour in online (live) and offline deployments using popular web servers and web applications. Our results show that Web-Armour significantly increases the cost of reconnaissance with minimal impact to benign users.
- **Enable enforcement with imperfect profiling:** We show throughout our experiments that Web-Armour offers a more flexible and feasible approach when compared to debloating. While debloating approaches may lead to breaking applications' functionality if profiling is under-approximated or incomplete, Web-Armour handles profiling inaccuracies through minimal time delays for benign users.

2. Background

This section discusses the general architecture for web application/server deployments and the techniques and tools used in reconnaissance and web vulnerability scanning.

Web Servers and Web Applications. A *web server* is a piece of software that is responsible for storing and delivering content through the internet to the clients that request it [11]. The client and the server have a request-response relationship conducted over the HTTP(S) protocol. Modern web servers, such as Apache HTTP Server and Nginx, provide a broad range of functionalities and options for web application developers. With a wide variety of needs and requirements in web applications, numerous web server modules are installed, enabled, and configured in unique combinations [12], [13].

A *web application* is composed of a back-end, server-side component, running on top of a web server, and a client-side portion that runs locally for end-users within a client's web browser. The web application calls upon functionality at the web server level to perform various tasks, such as reading or writing to a database. This model of interaction is well-established in the delivery of online services over the internet [14], [15].

Web Scanning. Reconnaissance is the first step in most cyber attacks and is mainly focused on gathering information about the targeted system to establish an effective attack strategy [16]. A common technique used in reconnaissance is web content scanning [17]. Web content scanners look for existing directories and files in a web application/server deployment through a dictionary-based attack against a target deployment. Dirb [17] and the "http-enum scan" within Nmap [18] are examples of such web content scanners.

Alternatively, web vulnerability scanners are a class of tools that search for security vulnerabilities and issues in web deployments in an automated fashion [14]. They attempt to uncover common security problems such as cross-site scripting, SQL injection, and cross-site request

forgery [19]. At a high-level, a web vulnerability scanner consists of three modules: a crawler module, an attack module, and an analysis module [20]. The crawling module is seeded with a set of URLs, retrieves the corresponding pages, and follows links attempting to identify all reachable pages in a web application.

Moreover, the crawler identifies all potential input vectors to the application, such as GET parameters and fields of web forms. For each of the inputs identified, the attack module generates values that might trigger vulnerabilities. Next, the analysis module attempts to detect potential vulnerabilities by analyzing the responses returned by the target to the attack module probes [20]. A few examples for off-the-shelf automated web vulnerability scanning tools include Nikto [21], Skipfish [22], Nessus [23] and Wapiti [24].

3. Design Overview

In this section, we discuss the design of Web-Armour, its main components, and the corresponding threat model. Web-Armour injects scan-impeding delays to infrequently executed code of a web deployment and considers two different levels where delays can be injected: (i) in the source code of the web server (ii) in the source code of the web application.

3.1. Web-Armour Components

As shown in Figure 1, Web-Armour can be divided in two main components: (i) a profiling engine, and (ii) a delay enforcement engine.

Profiling Engine. We refer to *profiling* as the process of identifying the code in the web deployment, that was executed as a result of processing the client-side HTTP(S) requests sent to the web (application/server) deployment. This enables us to ultimately differentiate cold code from the rest of the code.

Delay Enforcement Engine. Once a profile is generated, it becomes possible to distinguish cold code snippets from the rest of the code. As previously stated, cold code refers to code sections that are seldom accessed by benign users. Once the cold code has been identified, scan impeding delays can be injected to all code chunks that are considered cold. We refer to these delay-present versions of the web servers and web applications as *scan-resistant*.

3.2. Web Server Augmentation

To facilitate the profiling process on the web server side, first we instrument the source code of the web server using LLVM [25]. Implemented as an LLVM transformation pass that runs as part of the web server compilation process, the instrumentation injects a small snippet of code to each basic block in the web server source code. A basic block is a sequence of instructions that are guaranteed to be executed together without interruption [25]. The added code provides: i) unique identifiers to basic blocks of the web server source code, and ii) a count of the number of times each basic block has been executed. Once the web server source code is instrumented, it can be profiled by sending client-side HTTP(S) requests to it.

A profile contains the web server basic blocks covered (i.e. the block ID), coupled with the execution counts for each of these blocks when sending HTTP(S) traffic. The delay enforcement mechanism is implemented in the web server as an LLVM transformation pass and is used to inject scan-impeding delays during the compilation process. These delays are applied to the cold blocks identified in the profiling stage based on the generated web server profiles. This process allows the injection of a configurable amount of delay to each of the identified cold blocks. It is worth noting that the web server augmentation process is compatible with all C-based web servers such as Apache and Nginx, irrespective of their version.

3.3. Web Application Augmentation

Web-Armour can also introduce delays at the web application layer. As part of this work, we apply scan-impeding delays to PHP web applications due to PHP's widespread use in more than 76% of all websites [26]. Our approach for injecting delays into web applications takes a one-level-higher granularity by injecting scan-impeding delays into cold functions rather than cold blocks. The highly dynamic nature of PHP poses challenges in obtaining accurate block-level coverage on a consistent basis [27]. All off-the-shelf PHP profilers, such as XDebug [28], Xhprof [29], and Zend Server [30] provide function-level profiling information, making this approach a viable alternative to obtaining block-level profiling information.

To streamline the profiling process in the web application, we leverage nikic's PHP-Parser [31] to generate instrumented versions of the web application. This instrumentation assigns a unique identifier to each function within the web application, enabling the recording of executed functions and their respective counts during dynamic operation. The approach mirrors the methodology used for web server augmentation, involving the initial profiling of the deployment under test and the recording of profiling information. In the context of web application augmentation, this profiling information includes the functions being executed coupled with their counts observed during the operation of the web deployment. Subsequently, based on this profiling data, we introduce scan-impeding delays to all cold functions within the web application. For delay injection, we also leverage nikic's PHP-Parser to create new PHP files of the web application with scan-impeding delays injected to all cold functions of the web application. It is important to mention that our approach can be applied to any PHP web application.

3.4. Threat Model

Our threat model mainly considers remote adversaries, who are leveraging automated reconnaissance and vulnera-

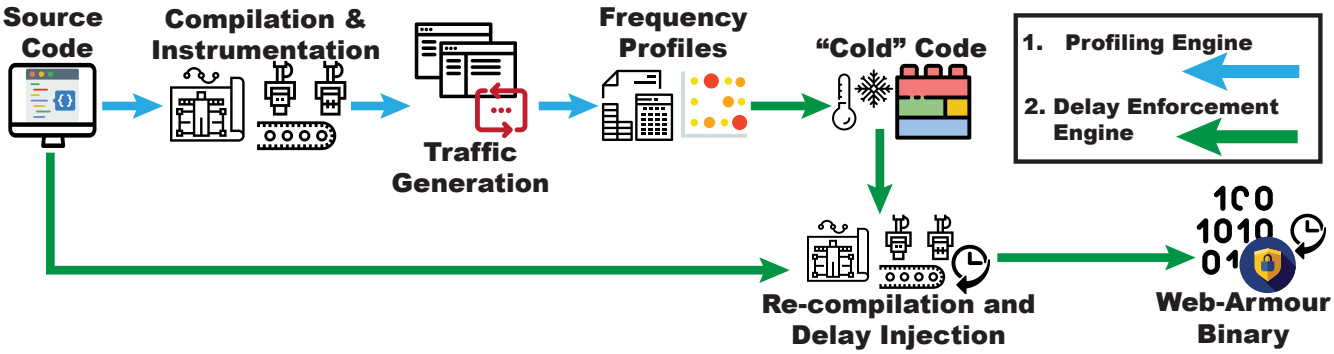

Figure 1. Web-Armour Process For Injecting Scan-impeding Delays – Web-Armour is composed of two main components: 1. Profiling Engine (dynamically identifies code triggered by client-side requests sent to the web deployment) 2. Delay Enforcement Engine (injects configurable, scan-impeding delays into the source code based on the "cold code" detected during profiling).

bility scanning tools against a public-facing web deployment. It is assumed that an attacker can send arbitrary HTTP(S) requests to the targeted deployment. With a range of tools and frameworks easily available to scan for vulnerabilities in web deployments, we believe the likelihood for adversaries conducting such scans to be highly realistic. Internet scans are widespread and the barrier for entry is very low [1]–[3]. For instance, an open-source installation of Kali Linux has more than 600 penetration testing tools preinstalled [32]. Among these are Nikto [21] and Nessus [23], [33], two well-known web vulnerability scanners [19].

4. Offline Scenario Description

In this section we evaluate Web-Armour in an offline scenario where profiling is done offline by emulating benign users interacting with the web deployment.

4.1. Profiling in an Isolated Setup (Offline)

In short, our approach classifies code triggered by benign traffic as hot and all other code portions as cold. To effectively differentiate between hot and cold code, we attempt to fully exercise the benign functionality of the web application/server pro-grammatically. In the first instance, we install an instrumented version of the web deployment. Next, using web crawling techniques, we recursively extract all URLs from the content hosted within the target deployment. A recursive process of following URLs to discover new URLs is conducted until all paths are exhausted. It is important to note that admin pages are included in the profiling.

Once all the URLs have been collected, a Selenium [34] script launches separate web browser tabs for each of the identified URLs. This resembles benign users navigating to all pages of the web deployment and preserves the "User-Agent" interaction with the deployment. For completeness, we also identify all available "form fill" elements within the web content and fill and submit them using Selenium to emulate benign users' interaction with these elements. This profiling process is summarized in Figure 2 and results in a usable and automated basic profiling scheme that can be

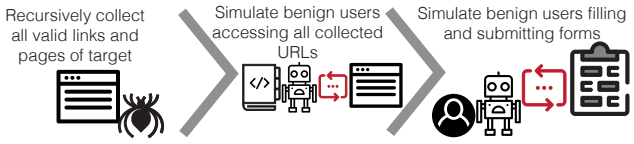

Figure 2. Offline Profiling Framework – The offline profiling framework is used to exercise the benign functionality of a web deployment. The framework emulates benign users following links to all pages in the deployment, and then emulates user interaction with all identified pages.

utilized to emulate benign user interaction with a specific web application/server deployment.

4.2. WordPress & Apache

To demonstrate the effectiveness of Web-Armour in offline scenarios, we construct preliminary experiments using WordPress v6.0.2 [35] and Apache v2.4.46 [36]. WordPress is a popular, feature-rich, and open-source PHP content management system that is used by 43.2% of all websites [10]. In our experiments, we create a sample WordPress blogging deployment and generate sample content using the `fakerpress` [37] WordPress plugin. We study blogging deployments due to their widespread presence in more than six-hundred million websites on the internet, constituting more than a third of all websites [38], [39]. The content generated by `fakerpress` simulates a real-world WordPress blogging deployment, generating images, random user profiles, comment forms, and other types of data commonly found on internet blogs.

Using the offline profiling approach previously described, we interact with the WordPress deployment and record the benign usage profile of the underlying web server and web application. After profiling, we generate scan-resistant versions of the deployment by injecting scan-impeding delays into the cold code within the web server and/or web application. Subsequently, we perform vulnerability scanning on the scan-resistant deployments and compare the scanning times with the default installations without Web-Armour-injected delays. The timing results from these experiments are then compared to the default version of the

deployment. Our small-scale, experimental testbed consists of two hosts. The scanning host is running Kali Linux while the target host is configured with Ubuntu. To prevent any unpredictable network requests or traffic, both hosts are detached on an isolated (offline) subnet.

5. Online Scenario Description

To assess the practicality of Web-Armour in production environments, we tested its use in a real-world web deployment. It is a PHP web application that hosts a course at a US institution for higher education. Students utilize the deployment for accessing course material and for submitting assignments. The materials hosted by this web deployment include video lectures, reading assignments, labs, quizzes and exams. Additionally, the deployment hosts an interactive "evaluation oracle" to which students can submit programming assignments and receive reports on correctness.

5.1. IRB Approval

Our proposed data collection procedure was reviewed and approved by the university Institutional Review Board (IRB). The data collected did not include any private information and the experiments conducted as part of this research work entailed no overhead or specific tasks from the students. Due to this, our IRB Office ruled students' consent was not required to conduct this research.

5.2. Profiling the Online Deployment

For the online scenario we utilize student's traffic to exercise the benign functionality of the deployment. The underlying assumption of this traffic is that students enrolled in the course will interact with the course's web application/server in a benign manner and this will effectively lead to exercising the commonly-used functionality of the deployment. To collect the online web server source code profile, we deployed an instrumented instance of Nginx v1.18 [40]. With the instrumented web server in place, real-world traffic generated by students accessing the course content exercises blocks within the web server and daily reports of triggered blocks are generated to monitor the evolution of the profile over time.

Figure 3 illustrates the two profile collection approaches that were considered as part of this experimentation. The course content is hosted at a publicly-accessible IP address and allows both HTTP and HTTPS traffic. Due to this, initial collection of network traffic and request data contained both legitimate requests from students enrolled in the course, but also external traffic which includes (potentially) malicious behavior and automated scanning traffic from various unidentified sources. To generate accurate profiles, we attempt to isolate requests generated by students from other, suspicious traffic, to create "clean" benign usage profiles.

Revised Setup – Separating Network Requests. To facilitate a more controlled approach to collecting benign traffic

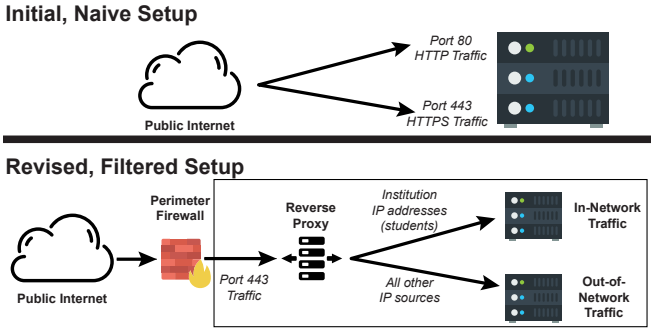

Figure 3. Online Experimentation Setup – The initial setup consisted of a publicly-accessible web server allowing HTTP and HTTPS traffic. The revised setup consists of two identical servers operating behind a reverse proxy, one server handling requests generated from the institutional network by authenticated users (in-network traffic) and the other server instance handling traffic from all other origins (out-of-network traffic).

to the target web server, we adopt architectural changes to server, as shown in the bottom of Figure 3. We make the assumption that interactions from within the authenticated university network to the target deployment are benign, exercising normal functionality, and that all other interactions with the deployment may be potentially malicious.

With the goal of separating benign traffic from all other traffic, we deploy two identical instances of the course web application/server and appropriately filter traffic to each. An Nginx reverse proxy is utilized to separate traffic with IP addresses that originate from within the authenticated network of the higher education institution from all other traffic sources. In this manner, requests that originate from the institution are directed to a dedicated instance of the web deployment and constitute the "in-network" traffic, while all other traffic flows to the second instance of the web deployment and constitutes the "out-of-network" traffic, as shown in Figure 3. We believe that the assumption of requests from in-network IP addresses equating to student/benign requests is sound and reliable due to the fact that in-network users must first authenticate with institutional resources in order to utilize the wireless, the wired, or the institutional VPN network segments. This authentication guarantees that all users that gain internet access through the institutional network have their traffic monitored by the institutional deep-packet inspection tool. Additionally, we validate that the in-network collection of requests contains only institutional-affiliated IP addresses by analyzing the associated web server logs.

Students may also access the web deployment from off-campus resources, however, this traffic will be directed to the out-of-network deployment and will not be included in the in-network profile. Despite this, we find that we have a sufficient amount of traffic to the in-network server in order to generate accurate benign usage profiles. On average, there were around 485 requests per day received by the in-network server during the profiling phase. This represents 12% of the total traffic received by both web deployments.

Scan-Resistant Deployments. To evaluate Web-Armour using real-world data, we employ three different approaches

TABLE 1. THE RESULTS OF SCANNING WEB-ARMOUR-PROTECTED WORDPRESS DEPLOYMENTS UNDER VARYING LENGTHS OF SCAN-IMPEDING DELAYS IN COLD CODE OF THE DEPLOYMENT.

	Profile	0ms	5ms	10ms	20ms	30ms
Web Server	Nikto	83s	1,135s	2,130s	**DNF**	**DNF**
	Nessus	744s	1,532s	2,160s	3,180s	3,960s
	Dirb	190s	27,931s	51,543s	75,322s	**DNF**
	Skipfish	360s	4,270s	**DNF**	**DNF**	**DNF**
	Wapiti	1,285s	7,771s	9,237s	12,298s	15,600s
	Nmap [1]	12.6s	347.9s	665.8s	1310.3s	**DNF**
Web App.	Nikto	83s	87s	104s	142s	165s
	Nessus	744s	3,458s	6,109s	14,100	19,231s
	Dirb	190s	207s	218s	243s	262s
	Skipfish	360s	6,077s	**DNF**	**DNF**	**DNF**
	Wapiti	1,285s	8,202.3s	10,063s	13,970.4s	18,479s
	Nmap [1]	12.6s	13.8s	14.7s	14.7s	17.2s
Combined	Nikto	83s	1,147s	2,153s	**DNF**	**DNF**
	Nessus	744s	4,328	7,579s	16,160s	21,856s
	Dirb	190s	27,948s	51,560s	**DNF**	**DNF**
	Skipfish	360s	**DNF**	**DNF**	**DNF**	**DNF**
	Wapiti	1,285s	**DNF**	**DNF**	**DNF**	**DNF**
	Nmap [1]	12.6s	354.3s	676.9s	1,320s	**DNF**

[1] Using the nmap http-enum scan [18].
DNF indicates a scan that did not finish (terminated because of timeouts).

for creating scan-resistant versions of the web server for the instructional web deployment. These approaches we term (i) *naive*, representing basic blocks triggered by any external, non-institutional (out-of-network) traffic; (ii) *non-naive*, representing basic blocks triggered only by traffic received at the in-network server; and (iii) *frequency analysis*, representing a more fine-grained approach of classifying cold blocks based upon the frequency of execution.

6. Experimentation Results

In this section we present the experimental results of evaluating Web-Armour in the scenarios described in Sections 4 and 5. We show the slowdown experienced by vulnerability scanning tools when targeting the scan-resistant deployments compared to the default installations. Moreover, we examine the performance impact to benign users when browsing the scan-resistant deployments.

6.1. WordPress & Apache

As outlined in Section 4.2, we installed an instrumented WordPress application on top of an instrumented Apache web server. During the initial instrumentation and compilation process, we discovered 117,834 basic blocks within the Apache v2.4.46 web server and identified 11,976 function in the WordPress v6.0.2 web application. Next, we applied the profiling methodology outlined in Section 4.1, to exercise the deployment capturing the web server blocks and WordPress functions triggered during the profiling. Throughout the profiling, 6,307 basic blocks were triggered on Apache while WordPress triggered 1,533 functions. All web server basic blocks and PHP functions executed during the profiling phase were considered hot portions of the code. All the unreached segments of the code were categorized as cold, and constant scan-impeding delays of 5ms, 10ms, 20ms, and 30ms were injected into them for various trials. We conducted multiple experiments involving the injection of scan-impeding delays into either the web server layer, the web application layer, or both.

We present the complete results of our experiments in Table 1. When scan-impeding delays are injected into the web server layer we find that in the best-case scenario, Web-Armour increases the overall time to complete the scan by upwards of 396 times over baseline time requirements. Further, in *every* circumstance of injecting scan-impeding delays to the web server, we see an increase in cost of adversarial actions over baseline timing requirements. Additionally, as the scan-impeding delays values increase, many tools (particularly Skipfish and Nikto) timeout and the scans terminate and fail to complete.

Conversely, when introducing scan-impeding delays into the web application layer, our findings, presented in Table 1, indicate that in the best-case scenario, Web-Armour increases the overall scan completion time by 26 times compared to the baseline requirements specifically in the case of Nessus. In the case of Skipfish, the scans were frequently terminated. Furthermore, in certain scenarios, we observed that scan-impeding delays had minimal impact on scanning tools, and scanning times remained unaffected. For instance, the Nmap scan duration increased only from 12s to 17s when 30ms scan-impeding delays were injected, and similar patterns were observed with Nikto and Dirb.

This happens because Nessus, Skipfish and Wapiti implement web application specific tests like SQL Injection and Cross Site Scripting. In these particular tests, the tools directly engage with the web application by populating forms with malicious payloads and sending a huge amount of POST requests. This causes a significant slowdown experienced by these tools when scan-impeding delays are injected into the web application layer. Conversely, Nmap, Dirb and Nikto scans primarily focus on the web server, involving minimal interaction with the web application layer and limited POST requests. As a result, these tools do not experience a significant slowdown when delays are injected into the web application layer.

Additionally, we conduct experiments involving the simultaneous injection of scan-impeding delays into both the web server and the web application layer. The results of introducing delays into both layers are presented in the bottom section of Table 1. The results suggest that injecting delays into both layers mainly aggregates the delays encountered by automated scanning tools.

Additional Offline Scenarios. To enhance the overall assessment we incorporate two additional offline scenarios. These scenario involves a MediaWiki website on top of Apache and a Social media website on top of Nginx. The

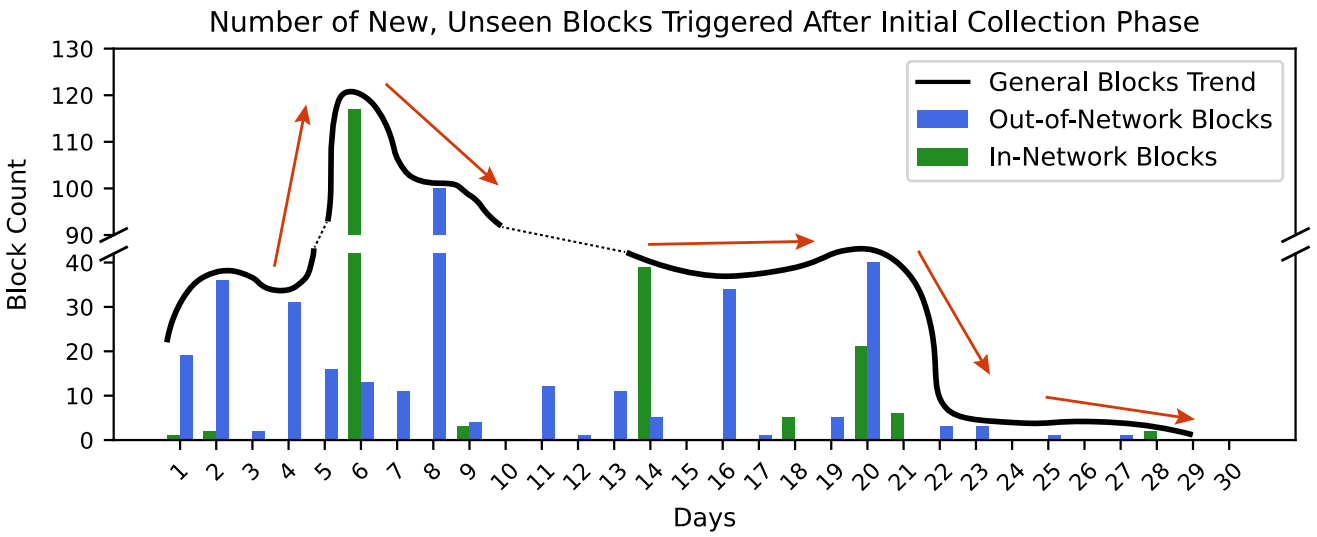

Figure 4. Real-World Profile over Time – This graphical representation captures the evolution of the triggered basic blocks that are new and not previously recorded each day. Overall, our results show a steep increase in recorded basic blocks within the first few days of data collection, but after roughly 20 days of recording, new, unseen basic blocks triggered taper off dramatically.

complete setup and experimental results are presented in Sections A and B in the Appendix. Overall, we observe similar trends as seen in the WordPress scenario.

6.2. Online Scenario

We leveraged Web-Armour on the live, instructional website and show the slowdown experienced by scanning tools when targeting the scan-resistant deployments.

Profiling the Real-World Website. Following the same procedure outlined in Section 5.2, we installed the real-world deployment on top of an instrumented instance of the Nginx v1.18 web server for profiling with real-world user's traffic. Using the model illustrated in Figure 3, we use the "Revised, Filtered Setup" to collect, appropriately filter, and create the web server basic block profiles. For both instances of the web deployments we store the daily profiles (basic block IDs & their execution counts) and the requests that generate these profiles.

We profiled the deployment throughout an entire school semester, spanning approximately 3 months. Over this time frame, we continuously monitored the evolution of the web server profiles as students incrementally accessed and executed functionality on the course website. During the initial instrumentation and compilation process we identified a total of 36,531 basic blocks within Nginx 1.18. On the first day of data collection, the in-network server recorded 6,796 basic blocks exercised, while the out-of-network server recorded 7,362 blocks. Subsequently, there has been a continual rise in the number of new blocks encountered on both servers. We observed a significant decline in the number of new blocks triggered after approximately 20 days of data collection for both servers. For example, there were only 35 new blocks exercised on the in-network server between day 21 and day 60, which is less than 0.6% of the blocks exercised in the first 20 days. Based on this observation, we determine that 20 days of profiling was sufficient to cover most of the benign functionality of the deployment.

In-Network Server Profile. The in-network server profile includes the basic blocks (along with their associated counts) that were executed in the in-network web server during the initial 20 days of profiling. On the first day of data collection, there were 6,796 basic blocks exercised on the in-network server. Figure 4 illustrates the evolution of new blocks encountered for the next 30 days following the initial deployment. New content was added on days 6 and 14, corresponding with the observed spikes. However, after 20 days, newly encountered blocks taper off and become very rare. Overall there were 6,963 basic blocks exercised in the in-network server in the first 20 days of profiling.

Out-of-Network Profile. The out-of-network server profile consists of the basic blocks exercised (with their associated counts) as part of the initial 20 day time period from the out-of-network server deployment. On the first day of data collection, there were 7,362 basic blocks exercised on the out-of-network server. Figure 4 illustrates the evolution of new blocks encountered for the next 30 days following initial collection. As previously mentioned, after roughly 20 days new blocks taper off dramatically. Overall there was 7,663 basic blocks executed on the out-of-network server in the first 20 days of profiling. It is important to mention that all the blocks activated on the in-network server were also triggered on the out-of-network server.

Creating Scan-Resistant Online Deployments. With the collected web server profiles, we utilize Web-Armour to create scan-resistant versions of the underlying Nginx web

TABLE 2. ONLINE TIMING ANALYSIS – THE RESULTS OF SCANNING THE INSTRUCTIONAL WEBSITE WITH WEB-ARMOUR UNDER VARYING LENGTHS OF SCAN-IMPEDING DELAYS IN COLD CODE IN THE WEB SERVER. WE COMPARE THREE DIFFERENT SCAN-IMPEDING DELAY INJECTION STRATEGIES.

	Profile	0ms	5ms	10ms	20ms	30ms	50ms
Naive Approach	Nikto	20s	25s	32s	41s	50s	68s
	Nessus	1,075s	1,124s	1,148s	1,158s	1,200s	1,380s
	Dirb	52s	74.8s	101.7s	145.5s	191.23s	284.27s
	Skipfish	366s	1,240s	4,407s	8,582s	17,629s	**DNF**
	Wapiti	119.73s	123.76s	149.89s	154s	176.26s	177s
	Nmap [1]	7.2s	7.7s	8s	8.7s	9.4s	10.5s
Non-Naive Approach	Nikto	20s	1,117s	**DNF**	**DNF**	**DNF**	**DNF**
	Nessus	1,075s	5,400s	**DNF**	**DNF**	**DNF**	**DNF**
	Dirb	52s	2,002.6s	3,847.7s	7,528.8s	11,157s	18,561.6s
	Skipfish	366s	2,965s	5,396.7s	**DNF**	**DNF**	**DNF**
	Wapiti	119.73s	267.9s	311.89s	414.5s	686.8s	1,019s
	Nmap [1]	7.2s	185.8s	225.7s	404.4s	588.8s	956.7s
Frequency Analysis	Nikto	20s	1,357s	**DNF**	**DNF**	**DNF**	**DNF**
	Nessus	1,075s	7,200s	**DNF**	**DNF**	**DNF**	**DNF**
	Dirb	52s	2,323.6s	4,440.8s	8,750.5s	12,358s	19,753s
	skipfish	366s	**DNF**	**DNF**	**DNF**	**DNF**	**DNF**
	Wapiti	119.73s	295.5s	397.1s	613.89s	886.19s	1,218.4s
	Nmap [1]	7.2s	282.5s	302.7s	557.9s	818.3s	1,338.8s

[1] We used the nmap http-enum scan [18].
DNF indicates a scan that did not finish (terminated because of timeouts).

server. Following the three different approaches mentioned in Section 5.2, we create experimental trials for the naive, non-naive, and frequency analysis approaches for selecting cold blocks for the injection of scan-impeding delays.

Naive Approach. To create the scan-resistant web servers following the "naive" approach, we consider all basic blocks executed on the out-of-network server in the first 20 days as hot blocks, while considering all others as cold. Using this strategy, we find that out of the total 36,531 basic blocks present within the Nginx web server 7,663 basic blocks were executed on the first 20 days of profiling. The remaining basic blocks were marked as cold and we injected scan-impeding delays of 5ms, 10ms, 20ms, 30ms and 50ms into the web server source code for different experimental trials (one scan-resistant version of the web server for each delay value). Afterwards, we deployed the real-world web application on top of the scan-resistant Nginx installation while automated scanning tools were leveraged against the scan-resistant deployments.

Our results, shown in Table 2, indicate that under the naive approach, adversarial actions against the target deployment are not dramatically affected. In the best-case scenario, the increase in time cost for the adversary is 48 times greater than the baseline cost and only one experimental trial failed to complete. However, on average, the increase in time cost for the adversary is 4.5 times greater relative to the baseline scanning cost when considering all experimental trials that finished. These results demonstrate a need for a more fine-grained and accurate way to determine cold blocks of code.

Non-Naive Approach. To create scan-resistant web servers following the non-naive approach, we consider all blocks triggered on the in-network server in the first 20 days of profiling as hot, while all other blocks to be cold. Overall there were 6,963 basic blocks exercised in the in-network server in the first 20 days of profiling. The remaining basic blocks were marked as cold and scan-impeding delays of 5ms, 10ms, 20ms, 30ms and 50ms were injected into the web server source for different experimental trials.

Our results, captured in Table 2, show that compared to the naive approach, the non-naive strategy results in significantly higher time cost on adversaries. In the best-case scenario, the increase in time cost is 357 times greater than the baseline. Additionally, a greater number of scans fail to complete, with 11 experimental trials timing out compared to only 1 in the naive approach. Furthermore, on average, of the trials that finished, the average increase in time cost for the adversary was 66.5 times greater relative to the baseline scanning cost. These results demonstrate that significant improvements in cost to adversarial actions is possible when a more fine-grained, accurate profiling method is employed.

Frequency Analysis Approach. In the frequency analysis approach, we utilize the same profile as in the non-naive approach. All blocks not triggered on the in-network server in the first 20 days of profiling are labeled as cold blocks. Additionally, we add blocks that are in the lower quartile of counts to cold blocks. The lower quartile of blocks included blocks that have a count less than 40 in the 20 day profiling period. With this distinction, the frequency analysis profile contains 5,115 basic blocks which are considered hot and all other blocks are considered cold. Following the same process as before, we generated different configurations of scan-impeding delay injections and then target the scan-

resistant deployments with vulnerability scanning tools.

Once the trials were completed, timing results were gathered and are shown in Table 2. Again, relative to previous strategies, the frequency analysis approach provides improvements in the cost required for adversaries to complete scans against the targeted web deployments. With a best-case scenario increase in time cost of 380 times greater relative to the baseline, further refining the profiling strategy results in better outcomes. Additionally, on average, of the trials that finished, the average increase in time cost for the adversary was 86.9 times greater relative to the baseline scanning cost with 13 experimental trials timing out.

Injecting Scan-Impeding delays into the Web Application. We conducted additional tests by introducing scan-impeding delays into the web application layer of the instructional website. We observed that in all the scenarios the scanning tools were not affected by the delays injected. The effectiveness of scan-impeding delays on this layer was limited due to the insufficient number of cold functions identified during the profiling phase. Consequently, the opportunities for injecting effective delays were comparatively low. These results are attributed to the instructional website being a specialized deployment with a small, single-purpose code base compared to WordPress. The instructional website has a total of 298 functions compared to 11,976 functions in WordPress. The full results of these experiments are detailed in Section C in the Appendix.

6.3. Scanning Tools Findings

The automated vulnerability scanning tools that we used as part of the experiments reported on 29 vulnerabilities on the tested deployments, more details are available in Table 10 in the Appendix. These vulnerabilities are well-known and reported in the MITRE CVE database [41]. Aside from the well-know vulnerabilities mentioned, the scanning tools also revealed security misconfigurations, dangerous files, insecure headers and web application sitemaps.

These issues were not discovered, or were reported after a long time, when Web-Armour was deployed because of overall timeouts and delayed responses. In instances where timeouts occur, Web-Armour will lead to preventing attackers from discovering the vulnerabilities in the targeted deployments. For instance CVE-2021-21703 and CVE-2021-21707 were not discovered in the instructional website when 10ms or more scan-impeding delays were injected into the web deployment. In cases of no timeouts, Web-Armour will slow down and disrupt the adversaries. For example, CVE-2022-22719 and CVE-2022-22720 were discovered after 12 minutes in unprotected deployments, whereas it took over 6 hours in scan-resistant deployments. This increased time allows defenders to detect and thwart scanning attempts.

6.4. Impact on Benign Users' Requests

Our proposed framework injects scan-impeding delays to the source code of web deployments, potentially affecting the benign user experience. We measure this performance overhead by making a series of benign testing requests to the target web deployments. These requests are sent to the target deployment and the the round-trip time needed to send all requests and receive responses is recorded. Additionally, we calculate the average time per request. Ultimately, we compare the outcomes of the scan-resistant deployments with the baseline installations that have no injected delays.

TABLE 3. THE PERFORMANCE IMPACTS ON THE WORDPRESS DEPLOYMENTS WHEN INJECTING SCAN-IMPEDING DELAYS AT THE DIFFERENT LAYERS OF THE WEB DEPLOYMENT.

	Metric	0ms	5ms	10ms	20ms	30ms
Web Server	RTT [1]	62.3s	63.3s	63.4s	64.2s	64.8s
	TPR [2]	228	231	232	234	237
	% Increase	-	1.4	1.8	2.9	4
Web App.	RTT [1]	62.3s	63.3s	63.9s	64.9s	66.1s
	TPR [2]	228	231	233	237	241
	% Increase	-	1.5	2.5	4.1	6.1
Combined	RTT [1]	62.3s	63.7s	64.4s	65.3s	66.6s
	TPR [2]	228	232	235	238	243
	% Increase	-	2.2	3.3	4.7	6.9

[1] RTT refers to the round trip time for all the requests.
[2] TPR refers to the average time per request measured in ms/request.

6.4.1. WordPress. To measure the performance impact for offline scenarios we adopt a methodology similar to the one suggested by Meng et al. [42]. We leveraged the OWASP ZAP [43] as a network proxy to capture the interactions between the client and the web deployment. We crawled the WordPress deployment with ZAP's spider program and stored the generated requests for testing purposes. Overall, there were 274 requests captured for testing. These testing requests were then replayed to the various scan-resistant versions of the WordPress deployments and the round-trip time for all requests and the associated responses were measured. We additionally calculate the average round-trip time per each request.

Table 3 captures the full results of these experiments. For the WordPress Blogging deployment, when compared relative to the baseline time costs, we observe that the performance impact to benign users introduced by Web-Armour is 1.4% over baseline cost in the best-case scenario of 5ms scan-impeding delay injection and 6.9% increase over baseline cost in the worst-case scenario of 30ms scan-impeding delay injection to both the web server and web application. We observe only a minor increase in the average time per request. For example, even in the worst-case scenario, the average server response time per request is 243 ms/request, which remains within acceptable limits. According to [44], [45], any server response time up to 826 ms/request is deemed acceptable for general-purpose web environments. It is also worth mentioning that there

TABLE 4. THE PERFORMANCE IMPACTS ON THE ONLINE SCENARIO WHEN INJECTING SCAN-IMPEDING DELAYS INTO COLD CODE AT THE WEB SERVER LAYER.

	Metric	0ms	5ms	10ms	20ms	30ms
Non-Naive	RTT [1]	931.8s	937.8s	944.7s	944.9s	946.9s
	TPR [2]	600	604	608	608	610
	% Increase	-	0.64	1.38	1.41	1.62
Frequency Analysis	RTT [1]	931.8s	966.2s	973s	1005s	1031s
	TPR [2]	600	622	627	647	664
	% Increase	-	3.7	4.4	7.9	10.6

[1] RTT refers to the round trip time for all the requests.
[2] TPR refers to the average time per request measured in ms/request.

were no instances of request timeouts, and responses were successfully received for all requests sent.

6.4.2. Online Deployment. In the online scenario, we utilize data that was gathered but not used in the development of the web server profiles. Specifically, we randomly select some web requests from outside the initial profiling period of 20 days. Particularly, we gathered 1,553 requests from the web server logs of the in-network server. It's noteworthy that all 1,553 requests originated from institutional IP addresses, and therefore known to be benign. These testing requests were then launched in an automated fashion against the various scan-resistant versions of the real-world deployments created using the Non-Naive and frequency analysis approaches and the round-trip time for all requests was measured. We also calculate the round-time per request.

Table 4 captures the full results of these experiments. We find that for the non-naive approach in the best-case scenario, Web-Armour introduces a benign performance impact of 0.6% (5ms scan-impeding delay configuration) over baseline time costs and in the worst-case scenario, Web-Armour introduces a benign performance impact of 1.6% (30ms scan-impeding delay configuration) over the baseline time values. For the frequency analysis approach, the performance impact is higher reaching up to 10% when 30ms scan impeding delays are injected. It is important to note that there were no instances where a request timed out, and responses were received for all sent requests.

7. Discussion and Limitations

Web-Armour has the potential to turn the tables on adversaries by increasing the costs of reconnaissance and vulnerability scanning. However, there are a number of trade-offs that need to be considered when deciding on how to use Web-Armour.

Injecting delays into Web Servers vs Web Applications. Different vulnerability scanning tools exercise web deployments in different ways. While certain tools will perform web application tests, other tools will focus more on the web server layer. For example, Nessus, Skipfish, and Wapiti perform web-application-specific tests, such as SQL injections and cross site scripting, and will therefore generate large amounts of POST requests trying to inject malicious payloads to all form elements of a website. For such tools injecting scan-impeding delays into the web application layer is more effective in interfering with their actions.

In contrast, tools such as Nikto, Dirb, and Nmap focus on the web server. Dirb and Nmap perform web content scanning and have little interaction with the web application [17], [46]. Nikto is a web server scanner that scans against known vulnerabilities and performs version checks on web servers [21]. For these tools Web-Armour is more effective when scan-impeding delays are injected in the web server source code. Our experiments further suggest that scan-impeding delays are not effective in the web application layer for specialized web deployments with a small code base. All is all, the decision on which layer(s) the scan-impeding delays should target (web server, web application, or both) can be specific to that particular deployment. Our work shows that it is feasible to inject scan-impeding delays and a delay-injection-based framework such as Web-Armour can provide real security benefits in thwarting web vulnerability scanning with a minimal impact on benign users.

Under-approximation in Profiling – Using an Automated Approach. Our profiling depends on resembling benign usage interactions with the web deployment. Given the diverse range of technologies and environments utilized in web deployments, this can present a significant challenge. For example, certain web applications rely on the WebSocket protocol instead of the traditional HTTP(S) protocol, necessitating their integration into the benign usage profiling process. While we recognize that the profiling approaches (refer to Sections 4.1 and 5.2) we have developed for our specific scenarios might not universally apply to all web deployments, the core approach of Web-Armour would remain the same across different deployments.

Previous research shows that Selenium can be used to build comprehensive profiling frameworks to exercise different web deployments [5], [47]. By automating actions such as clicking buttons, filling out forms, and navigating through web pages, Selenium can accurately emulate user interactions that occur during benign usage. Moreover, Selenium's support for multiple browsers and platforms ensures that the replication resembles the diverse environments in which the web application will be used. The widespread adoption of Selenium in industry [48] and in academic settings [5] further validates its effectiveness for accurately replicating benign usage interaction with a web deployment. Considering that, we believe that Selenium allows users to develop comprehensive profiling frameworks to mimic benign user interactions with web deployments under realistic conditions. While using real user traffic for profiling is considered optimal, it might not be feasible in certain situations. However, since Web-Armour can tolerate profiling inaccuracies, automated profiling approaches can still facilitate robust profiling as shown in our experiments.

Over-approximation in Profiling – Accounting for Pol-

luted Profiling Data. While Web-Armour enables some flexibility, profiling remains crucial in mitigating adversarial efforts against a targeted deployment. Including attack traffic in the profiling phase can affect the efficiency of Web-Armour as shown in the scanning results of the Naive approach in Table 2. We envision profiling to be done in a controlled environment in conjunction with internal web application testing, ensuring that the deployment is profiled with no (or a limited amount of) malicious traffic. Such testing is an essential element in web development. Thus, attack traffic will (likely) only interact with the scan-resistant web deployment. While accurate profiles enable greater success in this direction, our primary goal is not to develop optimal profiling methods, but rather to explore a technique that slows down adversarial reconnaissance and automated web vulnerability scanning.

Web Server Usage. We noticed throughout our experiments that the web applications utilize only a small portion of the web server source code. For instance, the number of code blocks in the Apache version (2.4.46) we used for experimentation was 117,834 blocks while the Nginx version (1.18) had 36,531 blocks. The WordPress blogging website used 6,307 blocks which is less than 6% of the total blocks of the Apache web server. On the other hand, the instructional website deployment used 6,963 blocks which is less than 20% of the total blocks in Nginx.

These results indicate that while Apache and Nginx offer a wide range of functionality and features, only a small portion of that functionality is actually leveraged by a given web application. Different applications will trigger different portions of the web server source code based upon the applications' needs and the content being served. This showcases the unique level of variety in web server configuration; not only on the explicit modules being used, but also the implicit differences that stem from the content being served. Web-Armour attempts to fend off the unused code/functionality in the web server from being exploited by adversaries.

Sources of the Slowdowns. The primary cause of scanner slowdowns is that, unlike benign traffic, vulnerability scanners largely explore error handling portions of the code. For instance, during a Nikto scan on the WordPress deployment, three blocks within the apache http_filters.c file were executed over 26,400 times and four blocks in the same file were executed approximately 295 times. This file handles various HTTP errors (e.g. Content-Length errors, overflow/underflow errors, Content-Encoding errors, etc..) [49]. These blocks were not triggered by benign user requests. On another case, during a Dirb scan on the WordPress deployment, two blocks within the apache server/log.c were executed over 5,000 times, while two additional blocks in the same file were executed more than 4,000 times. This file handles error logging [50] and the mentioned blocks were not triggered by benign requests. We observed similar patterns with other web vulnerability scanning tools. This reinforces our primary argument that web vulnerability scanners focus on dead, unused or infrequently executed chunks of the code that are not commonly targeted by benign traffic.

8. Related Work

In this section, we present related work most relevant to this study. Previous works have looked into different approaches to increase the security of web deployments.

Web Security. Fraunholz, *et al*. [51] illustrated the potential of using delay strategies against automated scanners. Their approach relies upon delaying requests from certain IP addresses if they exceed a specific number of requests within a time window. Instead, Web-Armour injects scan-impeding delays to infrequently-used blocks of web server source code, resulting in a more robust solution capable of withstanding adversaries that can spoof their IP address.

Multiple studies have proposed debloating as a technique to improve the security of a web deployment [5], [52]–[54]. While debloating removes dead/uneccessary code and has the potential to eliminate vulnerabilities, it may also break application functionality if benign usage (profiling) is under approximated or incomplete [5]. In contrast, Web-Armour adopts a more flexible strategy of penalizing traffic that explores infrequently/rarely utilized portions of code with scan-impeding delays, rather than removing functionality entirely. Thus, with Web-Armour the worst penalty encountered by benign users are slight time delays, while adversarial actions are severely affected.

On the other hand Li, *et al*. [55] reinforces our primary argument that a web vulnerability scanner interacts differently with a web deployment compared to benign traffic/users. The authors use web vulnerability scanners' and benign traffic to train a machine learning model and then to differentiate between the two. While their approach enforces controls at the proxy level, it can complement our method which involves injecting delays into the web deployment's source code.

General Software Security. Within traditional software security, "fuzzification" has been proposed as a recent mitigation strategy to hinder fuzzing attempts against software binaries [6]. This approach also focuses on injecting delays to cold paths in the software binary execution. While fuzzification assumes that the attacker possesses both the binary for the software application and the associated application settings, adversaries targeting web deployments may only access the web deployment through a web interface. Thus, our approach focuses on thwarting (automated) reconnaissance efforts since attackers must leverage reconnaissance and scanning techniques to obtain information about the web deployment before configuring and launching their exploits. Although fuzzification can be very effective, the lack of a timeout procedure in general-purpose applications may severely limit its benefits. We believe that injecting delays is a natural fit for web deployments since it accounts for and leverages request timeouts. Normally, a web client will wait for a specific amount of time for the server to respond, before the request times out [11], [56]. Thus, Web-Armour can thwart attacks by forcing timeouts in the reconnaissance/probing phase of a cyberattack.

Automatic software diversity is another technique that has been proposed to prevent exploiting software vulnerabilities [57]. This approach attempts to introduce uncertainty for the adversary when performing reconnaissance against the target but there are challenges such as performance impact and compatibility issues that hinder its adoption in real-world software applications [57].

Anti Reconnaissance Techniques. Previous studies [58]–[60] suggested the utilization of address mutation techniques to disrupt reconnaissance attacks. Their methodology involves mutating IP addresses of designated hosts with a high degree of unpredictability. However, these techniques offer no defense when the adversary already knows the domain name of the target host [58]. Additional studies conducted by Achleitner et al. [61], [62] explored the use of deception techniques to thwart network reconnaissance. Their methodology involves the generation of counterfeit virtual networks by manipulating network traffic, aiming to conceal critical resources and vulnerable endpoints. Web-Armour does not explicitly manipulate network traffic but rather delays automated malicious scanning requests.

9. Conclusions

In this work we explore delay injection as a defense mechanism in web deployments and present Web-Armour, our proposed framework for mitigating adversarial reconnaissance and automated web vulnerability scanning efforts against web application/server deployments. Through experimentation in isolated environments and on a real-world web deployment, we demonstrate that Web-Armour can significantly increase security, making it harder for adversaries to carry out reconnaissance and vulnerability scanning activities on web deployments, all while adding minimal performance overhead for benign users.

10. Acknowledgements

We are grateful to our anonymous reviewers for their thoughtful and constructive comments. This work was supported by the National Science Foundation (NSF) Award 2143393. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

References

[1] M. Allman, V. Paxson, and J. Terrell, "A brief history of scanning," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 77–82. [Online]. Available: https://doi.org/10.1145/1298306.1298316

[2] Z. Durumeric, M. Bailey, and J. A. Halderman, "An Internet-Wide view of Internet-Wide scanning," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 65–78. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/durumeric

[3] P. Richter and A. Berger, "Scanning the scanners: Sensing the internet from a massively distributed network telescope," in *Proceedings of the Internet Measurement Conference*, ser. IMC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 144–157. [Online]. Available: https://doi.org/10.1145/3355369.3355595

[4] Center For Strategic & International Studies, "Significant Cyber Incidents," https://www.csis.org/programs/strategic-technologies-program/significant-cyber-incidents, Accessed 01/2024.

[5] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: Quantifying the security benefits of debloating web applications," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1697–1714. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/azad

[6] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim, "Fuzzification: Anti-fuzzing techniques," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1913–1930. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/jung

[7] National Security Agency, "Look around: Women in cybersecurity episode 6," https://www.youtube.com/watch?v=5LpJIORWaYE, Accessed 01/2024.

[8] J. Dykstra, K. Shortridge, J. Met, and D. Hough, "Sludge for good: Slowing and imposing costs on cyber attackers," 2022.

[9] W3techs, "Comparison of the usage statistics of Nginx vs. Apache for websites," https://w3techs.com/technologies/comparison/ws-apache,ws-nginx, Accessed 01/2024.

[10] ——, "Usage statistics and market share of WordPress," https://w3techs.com/technologies/details/cm-wordpress, Accessed 01/2024.

[11] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach (6th Edition)*, 6th ed. Pearson, 2012.

[12] "Apache HTTP Server Project," https://httpd.apache.org/docs/2.4/mod/core.html, Accessed 01/2024.

[13] "Module ngx_http_core_module," http://nginx.org/en/docs/http/ngx_http_core_module.html, Accessed 01/2024.

[14] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A State-Aware Black-Box web vulnerability scanner," in *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 523–538. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe

[15] A. Hoffman, *Web application security: Exploitation and countermeasures for modern web applications*. O'Reilly Media, 2020.

[16] W. Mazurczyk and L. Caviglione, "Cyber reconnaissance techniques," *Commun. ACM*, vol. 64, no. 3, p. 86–95, feb 2021. [Online]. Available: https://doi.org/10.1145/3418293

[17] "Dirb," https://www.kali.org/tools/dirb/, Accessed 01/2024.

[18] "Script http-enum," https://nmap.org/nsedoc/scripts/http-enum.html, Accessed 01/2024.

[19] OWASP, "Vulnerability Scanning Tools," https://owasp.org/www-community/Vulnerability_Scanning_Tools, Accessed 01/2024.

[20] A. Doupé, M. Cova, and G. Vigna, "Why johnny can't pentest: An analysis of black-box web vulnerability scanners," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Kreibich and M. Jahnke, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 111–131.

[21] C. Sullo, "Nikto," https://www.cirt.net/Nikto2, Accessed 01/2024.

[22] Google, "Skipfish," https://code.google.com/archive/p/skipfish/, Accessed 01/2024.

[23] Tenable, "Tenable Nessus," https://www.tenable.com/products/nessus, Accessed 05/2024.

[24] N. Surribas, "Wapiti the web-application vulnerability scanner," https://wapiti-scanner.github.io/, Accessed 01/2024.

[25] A. Sampson, "LLVM for Grad Students," https://www.cs.cornell.edu/~asampson/blog/llvm.html, Accessed 01/2024.

[26] W3techs, "Usage statistics of PHP for websites," https://w3techs.com/technologies/details/pl-php, Accessed 01/2024.

[27] M. Madsen, "Static analysis of dynamic languages," PhD thesis, Aarhus University, March 2015.

[28] D. Rethans, "Xdebug," https://xdebug.org/docs/profiler, Accessed 01/2024.

[29] Phacility, "xhprof function-level hierarchical profiler for PHP." https://github.com/phacility/xhprof, Accessed 01/2024.

[30] zend.com, "Zend Profiler." https://help.zend.com/zendstudio/current/content/profiling.htm, Accessed 01/2024.

[31] N. Popov, "PHP-Parser," https://github.com/nikic/PHP-Parser, Accessed 01/2024.

[32] Offensive Security, "What is Kali Linux?" https://www.kali.org/docs/introduction/what-is-kali-linux/, Accessed 01/2024.

[33] Tenable, "Tenable Nessus," https://www.tenable.com/blog/tips-for-using-nessus-in-web-application-testing, Accessed 05/2024.

[34] J. Huggins, "Selenium," https://www.selenium.dev/, Accessed 01/2024.

[35] "Wordpress," https://wordpress.com/, Accessed 05/2024.

[36] Robert McCool, "Apache HTTP Server Project," https://httpd.apache.org/, Accessed 01/2024.

[37] G. Bordoni, "FakerPress," https://wordpress.org/plugins/fakerpress/, Accessed 01/2024.

[38] wpbeginner, "2024 Blogging Statistics, Trends & Data – Ultimate List (UPDATED)," https://www.wpbeginner.com/research/2022-blogging-statistics-trends-data-ultimate-list-updated/, Accessed 01/2024.

[39] Web Tribunal, "How Many Blogs Are There? We Counted Them All!" https://webtribunal.net/blog/how-many-blogs/, Accessed 01/2024.

[40] F5, "What is Nginx," https://www.nginx.com/resources/glossary/nginx/, Accessed 01/2024.

[41] The MITRE Corporation, "CVE," https://cve.mitre.org/, Accessed 01/2024.

[42] W. Meng, C. Qian, S. Hao, K. Borgolte, G. Vigna, C. Kruegel, and W. Lee, "Rampart: Protecting web applications from cpu-exhaustion denial-of-service attacks," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 393–410. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/meng

[43] OWASP, "Zed Attack Proxy," https://www.zaproxy.org/, Accessed 01/2024.

[44] littledata.io, "What is the average server response time?" https://lp.littledata.io/average/server-response-time, Accessed 01/2024.

[45] datadome.co, "8 ways to effectively reduce server response time," https://datadome.co/learning-center/how-to-reduce-server-response-time/, Accessed 01/2024.

[46] G. F. Lyon, *Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning*. Sunnyvale, CA, USA: Insecure, 2009.

[47] E. Vila, G. Novakova, and D. Todorova, "Automation testing framework for web applications with selenium webdriver: Opportunities and threats," in *Proceedings of the International Conference on Advances in Image Processing*, ser. ICAIP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 144–150. [Online]. Available: https://doi.org/10.1145/3133264.3133300

[48] J. Whittaker, J. Arbon, and J. Carollo, *How Google Tests Software*. Addison-Wesley Professional, 2012.

[49] Robert McCool, "Apache HTTP Server Project," https://github.com/omnigroup/Apache/blob/master/httpd/modules/http/http_filters.c, Accessed 08/2024.

[50] ——, "Apache HTTP Server Project," https://github.com/apache/httpd/blob/trunk/server/log.c, Accessed 08/2024.

[51] D. Fraunholz and H. D. Schotten, "Defending web servers with feints, distraction and obfuscation," in *2018 International Conference on Computing, Networking and Communications (ICNC)*, 2018, pp. 21–25.

[52] Q. Xin, M. Kim, Q. Zhang, and A. Orso, "Subdomain-based generality-aware debloating," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 224–236.

[53] B. Amin Azad and N. Nikiforakis, "Role models: Role-based debloating for web applications," in *Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '23. New York, NY, USA: Association for Computing Machinery, 2024, p. 251–262. [Online]. Available: https://doi.org/10.1145/3577923.3583647

[54] I. Koishybayev and A. Kapravelos, "Mininode: Reducing the attack surface of node.js applications," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 121–134. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/koishybayev

[55] X. Li, B. Amin Azad, A. Rahmati, and N. Nikiforakis, "Scan me if you can: Understanding and detecting unwanted vulnerability scanning," in *Proceedings of the ACM Web Conference 2023*, ser. WWW '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2284–2294. [Online]. Available: https://doi.org/10.1145/3543507.3583394

[56] K. Reitz, "Requests: HTTP for Humans," https://requests.readthedocs.io/en/stable/user/advanced/\#timeouts, Accessed 01/2024.

[57] P. Larsen, S. Brunthaler, and M. Franz, "Automatic software diversity," *IEEE Security Privacy*, vol. 13, no. 2, pp. 30–37, 2015.

[58] J. H. Jafarian, E. Al-Shaer, and Q. Duan, "An effective address mutation approach for disrupting reconnaissance attacks," *IEEE Transactions on Information Forensics and Security*, vol. 10, no. 12, pp. 2562–2577, 2015.

[59] ——, "Openflow random host mutation: transparent moving target defense using software defined networking," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 127–132. [Online]. Available: https://doi.org/10.1145/2342441.2342467

[60] E. Al-Shaer, Q. Duan, and J. H. Jafarian, "Random host mutation for moving target defense," in *Security and Privacy in Communication Networks*, A. D. Keromytis and R. Di Pietro, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 310–327.

[61] S. Achleitner, T. La Porta, P. McDaniel, S. Sugrim, S. V. Krishnamurthy, and R. Chadha, "Cyber deception: Virtual networks to defend insider reconnaissance," in *Proceedings of the 8th ACM CCS International Workshop on Managing Insider Security Threats*, ser. MIST '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 57–68. [Online]. Available: https://doi.org/10.1145/2995959.2995962

[62] S. Achleitner, T. F. La Porta, P. McDaniel, S. Sugrim, S. V. Krishnamurthy, and R. Chadha, "Deceiving network reconnaissance using sdn-based virtual topologies," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 1098–1112, 2017.

[63] W3techs, "Usage statistics and market share of MediaWiki," https://w3techs.com/technologies/details/cm-mediawiki, Accessed 01/2024.

[64] "Pixelfed," https://github.com/pixelfed/pixelfed, Accessed 01/2024.

[65] The ZAP Dev Team, "Options AJAX Spider screen," https://www.zaproxy.org/docs/desktop/addons/ajax-spider/options/, Accessed 01/2024.

Appendix

1. MediaWiki & Apache Scenario

In this section we present an additional offline testing scenario where we evaluate Web-Armour on a MediaWiki web deployment. MediaWiki is a popular open-source PHP content management system that is used by some of the highest traffic sites on the internet (e.g. WikiPedia) [63].

1.1. Scan-Resistant Deployments. In our experiments we create an instrumented sample MediaWiki v1.29.0 deployment and fill it with generated random content and run it on top of an instrumented Apache v2.4.46. This random content simulates a real MediaWiki deployment and contains MediaWiki pages, discussions and user profiles. Leveraging the offline profiling methodology mentioned in Section 4.1, we exercise the deployment and record the benign usage profile of the underlying Apache web server and the MediaWiki web application. After profiling, a scan-resistant version of the web deployment is created by injecting scan-impeding delays to the cold blocks and/or the cold functions of the web deployment. Once a scan-resistant deployment is created, automated reconnaissance and web vulnerability scanning tools are launched against the deployment. The timing results from these experiments are then compared to the default version of the deployment. Our small-scale, experimental testbed consists of two hosts. The scanning (malicious) host is running Kali Linux while the target host is configured with Ubuntu Linux.

During the initial instrumentation process, we identified a total of 117,834 basic blocks in the Apache web server and 27,781 function in the MediaWiki PHP application. During the profiling process, 6,214 basic blocks were triggered on the Apache web server and 3,712 functions were triggered on the MediaWiki web application. We create scan-resistant deployments by injecting scan-impeding delays into cold blocks of the web server and/or cold functions of the web application across various experimental trails. These deployments were then targeted with vulnerability scanning tools and the timing results were recorded.

TABLE 5. THE RESULTS OF SCANNING WEB-ARMOUR-PROTECTED MEDIAWIKI DEPLOYMENTS UNDER VARYING LENGTHS OF SCAN-IMPEDING DELAYS IN COLD CODE OF THE DEPLOYMENT.

	Profile	0ms	5ms	10ms	20ms	30ms
Web Server	Nikto	70s	672s	1,254s	**DNF**	**DNF**
	Nessus	720s	2,931s	3,391s	4,116s	4,850s
	Dirb	62s	21,737s	**DNF**	**DNF**	**DNF**
	Skipfish	617s	**DNF**	**DNF**	**DNF**	**DNF**
	Wapiti	627s	1,137s	2,373s	3,614s	4,555s
	Nmap [1]	9.6s	152s	**DNF**	**DNF**	**DNF**
Web App.	Nikto	70s	75s	84s	96s	114s
	Nessus	720s	3,402s	4,878s	6,513s	7,586s
	Dirb	62s	120s	131s	144s	163s
	Skipfish	617s	**DNF**	**DNF**	**DNF**	**DNF**
	Wapiti	627s	6,629s	12,688s	20,055s	**DNF**
	Nmap [1]	9.6s	9.8s	11s	13s	14s
Combined	Nikto	70s	682s	1,270s	**DNF**	**DNF**
	Nessus	720s	3,680s	5,156s	6,719s	7,886s
	Dirb	62s	21,750s	**DNF**	**DNF**	**DNF**
	Skipfish	617s	**DNF**	**DNF**	**DNF**	**DNF**
	Wapiti	627s	6,731s	12,888s	22,055s	**DNF**
	Nmap [1]	9.6s	170s	**DNF**	**DNF**	**DNF**

[1] We used the nmap http-enum scan [18].
DNF means the scan did not finish (terminated because of timeouts).

The full results of these experiments is presented in Table 5. When introducing scan-impeding delays into the web server layer, there is a consistent rise in the time required for adversarial actions compared to the baseline timing requirements in every scenario. In addition, as the scan-impeding delay values increase, a number of tools (particularly Nikto, Skipfish, Nmap and Dirb) timeout and the scans fail to complete. In total, there was 12 scans that were terminated because of timeouts when scan-impeding delays are injected into the web server layer.

Upon injecting scan-impeding delays into the web application layer, we observe that in the best-case scenario, Web-Armour increases the total scan duration, by upwards of 32 times over baseline time requirements in the case of Wapiti. For Skipfish, it was noted that the scans were often terminated. Notably, the introduction of scan-impeding delays into the web application layer does not significantly affect the scanning times for Nikto, Dirb, and Nmap. This is due to the fact that Nmap, Dirb and Nikto scans are more focused on the web server side and do not have a lot of interaction with the web application layer.

Furthermore, we perform experiments where scan-impeding delays are simultaneously injected into both the web server and the web application layer. The outcomes of introducing delays into both layers are outlined in Table 5. These results indicate that injecting delays into both layers accumulates the delays experienced by automated reconnaissance and web vulnerability scanning tools.

1.2. Benign Performance Impact. To measure the performance impact we adopt a methodology similar to the one

TABLE 6. THE PERFORMANCE IMPACTS ON THE MEDIAWIKI DEPLOYMENTS WHEN INJECTING SCAN-IMPEDING DELAYS AT THE DIFFERENT LAYERS OF THE WEB DEPLOYMENT.

	Metric	0ms	5ms	10ms	20ms	30ms
Web Server	RTT [1]	64.4s	65.7s	66.6s	67.7s	69.2s
	TPR [2]	210	215	218	221	226
	% Increase	-	2.1	3.5	5.2	7.4
Web App.	RTT [1]	64.4s	65s	65.6s	66.3s	67.1s
	TPR [2]	210	212	214	217	219
	% Increase	-	0.9	1.8	2.9	4.2
Combined	RTT [1]	64.4s	66.3s	67.3s	68.5s	69.7s
	TPR [2]	210	217	220	224	228
	% Increase	-	3	4.5	6.3	8.3

[1] RTT refers to the round trip time for all the requests.
[2] TPR refers to the average time per request measured in ms/request.

suggested by Meng et al. [42]. We leveraged the OWASP ZAP [43] as a network proxy to capture the interactions between the client and the web deployment. We crawled the MediaWiki deployment with ZAP's spider program and stored the generated requests for testing purposes. Overall, there was 306 requests captured for testing purposes. The testing requests were replayed to the various versions of the MediaWiki deployments and the round-trip time for all requests and the associated responses was measured. We also compute the average round-trip per request.

Table 6 captures the full results of these experiments. In general, there is a marginal decrease in the performance of the scan-resistant web deployments compared to unprotected deployments. For instance, even in the most unfavorable scenario, the average server response time per request is 228 ms/request, staying within acceptable thresholds. As per [44], [45], any server response time up to 826 ms/request is considered acceptable for general-purpose web environments. It is important to mention that there were no instances of requests timing out, and responses were successfully received for all the requests that were sent.

2. Social Media Scenario

In this section we present an additional scenario where we evaluate Web-Armour on a social media platform. Our experimentation focuses on Pixelfed v0.11.2, an open-source PHP image sharing social network platform [64].

2.1. Scan-resistant deployments. We construct experiments with Pixelfed v0.11.2 on top of Nginx v1.18. To emulate a real social media deployment, we established a sample Pixelfed deployment and populate it with randomly generated content, encompassing user profiles, images, and posts. Utilizing the profiling methodology detailed in Section 4.1 and augmented by the ZAP Ajax Spider, we exercise the deployment. The ZAP Ajax Spider, utilizing Selenium,

TABLE 7. THE RESULTS OF SCANNING WEB-ARMOUR-PROTECTED PIXELFED DEPLOYMENTS UNDER VARYING LENGTHS OF SCAN-IMPEDING DELAYS IN COLD CODE OF THE DEPLOYMENT.

	Profile	0ms	5ms	10ms	20ms	30ms
Web Server	Nikto	13s	1,064s	2,297s	**DNF**	**DNF**
	Nessus	1,150s	3,746s	**DNF**	**DNF**	**DNF**
	Dirb	8,016s	15,535s	31,231s	47,156s	**DNF**
	Skipfish	637s	**DNF**	**DNF**	**DNF**	**DNF**
	Wapiti	291.2s	299.5s	324.2s	351.1s	373s
	Nmap [1]	6.9s	182s	349s	684s	**DNF**
Web App.	Nikto	13s	13s	14s	16s	18s
	Nessus	1,150s	1,555s	1,601s	1,799s	1,830s
	Dirb	8,016s	8,020s	8,023s	8,027s	8,029s
	Skipfish	637s	**DNF**	**DNF**	**DNF**	**DNF**
	Wapiti	291.2s	691.3s	1,038s	1,744s	**DNF**
	Nmap [1]	6.9s	8s	8s	9s	11s
Combined	Nikto	13s	1,077s	**DNF**	**DNF**	**DNF**
	Nessus	1,115s	5,074s	**DNF**	**DNF**	**7,586s**
	Dirb	8,016s	15,555s	31,243s	47,171s	DNF
	Skipfish	637s	**DNF**	**DNF**	**DNF**	**DNF**
	Wapiti	291.2s	711.3s	1,077s	1,807s	**DNF**
	Nmap [1]	6.9s	184s	350.9s	685s	**DNF**

[1] We used the nmap http-enum scan [18].
DNF means the scan did not finish (terminated because of timeouts).

emulates user interactions within the web deployment [65]. This spider complements our profiling approach by enabling the simulation of clicks on various web elements presented in the Pixelfed deployment (like,share, etc.), which our initial profiling approach does not capture.

After profiling, we create scan-resistant deployments by injecting scan-impeding delays into cold blocks of the web server and/or cold functions of the web application across various experimental trails. These scan-resistant deployments were then targeted with automated reconnaissance and vulnerability scanning tools. Our small-scale, experimental testbed consists of two hosts. The scanning (malicious) host is running Kali Linux while the target host is configured with Ubuntu Linux.

We present the full results of the experiments in Table 7. We find that in *every* circumstance of injecting scan-impeding delays to the web server source code, we see an increase in the time cost of scanning over baseline timing requirements. Additionally, as the scan-impeding delay values increase, many tools (particularly Skipfish and Nessus) experience timeouts and the scans fail to complete. In total, 11 scans were terminated due to timeouts caused by the injection of scan-impeding delays into the web server layer.

For injecting scan-impeding delays into the web application layer, we find that for Wapiti and Nessus, Web-Armour increases the overall time to complete the scan times over the baseline time. We notice in the case of Skipfish, the scans were terminated in most instances. We also notice in some other scenarios (Nikto, Dirb and Nmap) that the scan-impeding delays have minimal impact on the scanning tools and that the scanning times were not affected by the delays

TABLE 8. THE PERFORMANCE IMPACTS ON THE PIXELFED DEPLOYMENTS WHEN INJECTING SCAN-IMPEDING DELAYS AT THE DIFFERENT LAYERS OF THE WEB DEPLOYMENT.

	Metric	0ms	5ms	10ms	20ms	30ms
Web Server	RTT [1]	55.7s	55.9s	56.8s	58s	60s
	TPR [2]	328	329	334	341	353
	% Increase	-	0.3	1.9	4.2	7.7
Web App.	RTT [1]	55.7s	56.5s	56.7s	57s	57.8s
	TPR [2]	328	332	333	335	340
	% Increase	-	1.5	1.8	2.4	3.8
Combined	RTT [1]	55.7s	56.6s	56.9s	58.4s	60.4s
	TPR [2]	328	333	335	344	355
	% Increase	-	1.7	2.1	4.9	8.4

[1] RTT refers to the round trip time for all the requests.
[2] TPR refers to the average time per request measured in ms/request.

injected. This happens because the aforementioned scans are more focused on the web server side and do not have a lot of interaction with the web application layer.

Additionally, we conduct experiments involving the injection of delays into both the web server and the web application layer. The results for injecting scan-impeding delays into both layers are shown in the lower part of Table 7. The findings indicate that injecting delays into both layers aggregates the delays experienced by scanning tools.

2.2. Benign Performance Impact. We measure the performance impact arising from the introduction of scan-impeding delays in Pixelfed deployments. We leveraged the OWASP ZAP [43] as a network proxy to capture the interactions between the client and the social media deployment. We crawled the Pixelfed deployment with ZAP's spider program and stored the generated requests (approximately 170 requests) for testing purposes. These testing requests were replayed across different versions of the Pixelfed deployments, and we measured the round-trip time for all requests along with their corresponding responses. Additionally, we calculated the average round-trip time per request.

The complete findings of these experiments are detailed in Table 8. Overall, there is a slight decrease in the performance of the scan-resistant web deployments when compared to unprotected deployments. Even in the least favorable scenario, the average server response time per request is 355 ms/request, remaining within acceptable thresholds. According to [44], [45], any server response time up to 826 ms/request is considered acceptable for general-purpose web environments. It is important to note that there were no instances of requests timeout, and responses were successfully received for all the requests sent.

TABLE 9. THE RESULTS OF SCANNING WEB-ARMOUR-PROTECTED INSTRUCTIONAL WEBSITE DEPLOYMENTS UNDER VARYING LENGTHS OF SCAN-IMPEDING DELAYS IN THE WEB APPLICATION LAYER.

	Profile	0ms	5ms	10ms	20ms	30ms
Web App.	Nikto	20s	20s	21s	21s	22s
	Nessus	1,075s	1,075s	1,075s	1,075s	1,076s
	Dirb	52	58.15s	58.8s	60s	63s
	Skipfish	366s	412s	420.8s	434.4s	444.4s
	Wapiti	119.73s	122s	148s	156.8s	165s
	Nmap	7.2s	7.28s	7.38s	7.45s	7.5s

We used the nmap http-enum scan [18].
DNF means the scan did not finish (terminated because of timeouts).

3. Instructional Website - Injecting scan-impeding delays into the web application

As mentioned in Section 6.2 we experimented with injecting scan-impeding delays into the web application layer of the instructional website. During the initial instrumentation we identified 298 functions within the web application. We exercised the deployment using the profiling approach detailed in Section 4.1. During the profiling phase 210 functions were executed, leaving 88 functions categorized as cold. To asses the impact of scan-impeding delays, constant delays of 5ms,10ms,20ms and 30ms were injected into all cold functions of the web application. Subsequently, vulnerability scanners were executed against these scan-resistant deployments, and the results were compared with default installations where no delays were introduced.

The full findings of the experiments are outlined in Table 9, showing that in all scenarios, the scanning tools were not affected by the delays injected. These results are attributed to the instructional website being a smaller deployment with minimal reliance on external dependencies. In contrast to the instructional website's 298 functions, our other test applications WordPress and MediaWiki have significantly larger code bases, with 11,976 and 27,781 functions, respectively. These results illustrate that, for web applications with small code bases, Web-Armour may not be effective when applied at the web application layer. Instead, its effectiveness is better realized when implemented at the web server layer, emphasizing the importance of considering the specific characteristics of the web deployment when employing Web-Armour protection.

4. Vulnerabilities discovered by the scanning

In this section, we provide a description for each of the vulnerabilities that were discovered by the automated scanning tools on the targeted deployments. Overall, the scanning tools reported on 29 vulnerabilities on the tested deployments. These vulnerabilities are well known and reported on the MITRE CVE database. Table 10 has the CVE number and the corresponding description for each one of the identified vulnerabilities.

TABLE 10. THIS TABLE DESCRIBES THE VULNERABILITIES REPORTED BY THE SCANNING TOOLS ON THE TESTED DEPLOYMENTS.

Vulnerability	Description
CVE-2021-21703	Invalid memory reads and Writes.
CVE-2021-21707	Unauthorized file access using NULL character injection.
CVE-2021-21708	Memory Overwrite (possible software crash).
CVE-2021-30641	Unexpected matching behavior with 'MergeSlashes OFF'.
CVE-2021-39275	Buffer Overflow in ap_escape_quotes() if given malicious input.
CVE-2021-34798	Malformed Requests may lead the server to dereference a NULL pointer.
CVE-2021-44224	A crafted URI sent to apache configured as a forward proxy can cause a crash.
CVE-2021-44790	Buffer overflow vulnerability in the mod_lua multipart parser.
CVE-2020-35452	mod_auth_digest: possible stack overflow by one null byte while validating the Digest nonce.
CVE-2021-26691	mod_session: Fix possible crash due to NULL pointer dereference, (Denial of Service).
CVE-2021-26690	mod_session: Fix possible crash due to NULL pointer dereference, (Denial of Service).
CVE-2020-13950	mod_proxy_http: Fix possible crash due to NULL pointer dereference, (Denial of Service).
CVE-2020-13938	Local users can stop the httpd process on Windows.
CVE-2019-17567	mod_proxy_wstunnel, mod_proxy_http: Handle Upgradable protocols end-to-end negotiation.
CVE-2022-22719	A carefully crafted request body can cause the process to crash.
CVE-2022-22720	HTTP request smuggling.
CVE-2022-22721	Possible buffer overflow with very large or unlimited LimitXMLRequestBody in core.
CVE-2022-23943	Out-of-bounds Write vulnerability (heap overwrite).
CVE-2022-26377	Possible request smuggling in mod_proxy_ajp.
CVE-2022-28330	Read beyond bounds in mod_isapi.
CVE-2022-28614	Read beyond bounds via ap_rwrite().
CVE-2022-28615	Read beyond bounds in ap_strcmp_match().
CVE-2022-29404	Denial of service in mod_lua.
CVE-2022-30522	Denial of Service mod_sed.
CVE-2022-30556	Information Disclosure in mod_lua with websockets.
CVE-2022-31813	X-Forwarded-For dropped by hop-by-hop mechanism in mod_proxy.
CVE-2021-40438	A crafted request uri-path can cause mod_proxy to forward.
CVE-2021-33193	Cache Poisoning.
CVE-2021-36160	A carefully crafted request uri-path can cause crash (DoS).