Warden: Multi-Layered Control Flow Integrity in Web Applications

A Major Qualifying Project (MQP) Report
Submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements
for the Degree of Bachelor of Science in

Computer Science

By:

Kush Shah
Samuel Parks
Keith DeSantis

Project Advisor:

Craig Shue

Date: April 2023

Abstract

This research introduces Warden, a function-level control flow enforcement for web applications. The goal of Warden is to strengthen the security of the Single Use-Server model by detecting attacks such as code injection and remote code execution. Both of these attacks leverage vulnerabilities to allow a user to execute arbitrary code on a server through a public facing website. Due to the nature of this, the code executed can range from covert (such as cryptomining or leaking data) to overt (such as denial of service or ransomware). While SuS architecture could prevent or mitigate damage from leaking and ransoming data, an attacker with knowledge of the underlying system could craft code injection attacks that would not be caught such as cryptomining. In addition to added security, Warden aims to answer the question of whether an asynchronous and layered approach to control flow enforcement is viable. Traditionally, CFI enforcement software uses a graph of function calls and blocks each time a new function is called to ensure that it is valid (found in the underlying graph). Modern web applications often call thousands of functions from each interaction with the user, and blocking on each one adds significant latency. The unique design of Warden is intended to lessen this problem. Warden is able to detect remote code execution in all cases except that which the malicious code was named the same as a valid function, called on the same line, and only used functions that the overwritten function had, on the same lines. However, this added security also increases overhead on the original system, on average increasing utilized CPU by 20%, memory by 8%, and latency by 535%. Cursory optimizations were able to improve this overhead significantly and we believe that further work would be able to increase the efficiency of threading in the application to reduce the overhead much further.

Acknowledgements

1. Craig Shue - For his constant support and knowledge as our advisor.
2. Yunsen Lei - For his incredible help problem solving and introducing us to Single Use Server architectures.

Contents

List of Tables

List of Figures

1 Introduction

Cybersecurity remains one of the most pressing concerns for organizations in today's digital landscape. With modern web applications storing databases of potentially sensitive user information, security of programmatic level execution is critical. Previous work developed the Single Use-Server model as a way to prevent privilege escalation and to containerize users to mitigate the damage of any successful attack on the web application [1, 2]. However, SuS is not without weaknesses. SuS is only able to detect a compromised container once the container attempts to access data without the proper permissions. Specifically, attacks such as code injection and remote code execution would go unnoticed by SuS' security measures.

To fill this gap, we propose Warden, a web-based, function-level control flow enforcement for web applications. Control flow integrity detection and enforcement has been used to increase the security of applications local to client machines, but is a relatively new innovation in terms of web applications [3]. Control flow integrity comes at a cost: it can be computationally intensive and increase latency in the application overall due to intercepting every function transfer. Warden combats this in two ways: asynchronicity and a layered approach. Instead of intercepting each function transfer, Warden operates asynchronously to the main application. Secondly, Warden uses set inclusion for primary validation, which will escalate to a lookup table serving as the ground truth if a function transfer is not recognised.

The proposed approach is expected to provide lower latency, which will allow flexible code analysis. Operating in parallel with the application comes at the cost of instant detection of anomalous behavior; however, we believe that the benefits in responsiveness and latency outweigh this cost. Due to complete control flow graphs for large scale applications being nearly impossible to create, the downside of parallel execution is not really an issue, as stopping execution at any anomaly would likely inhibit normal functionality of the application. The layered approach is also expected to decrease latency due to the use of a bloom filter, as the first layer, instead of the normal approach of a complete graph of the system. Bloom filters operate in constant time for adding and checking the existence of elements in a set. Additionally, bloom filters are space efficient compared to most data structures as they do not need to store the data itself.

While there are other web based CFI implementations in existence, such as Ghostrail [4, 5, 6], a web CFI implementation that forces users to navigate webpages according to expected flows, and ZenIDS, a programatic level control flow enforcement for PHP, our proposed approach is distinct with use of a multi-layered CFI system. In this multi-layered system the majority of benign traffic is approved by a lightweight layer, and only suspicious traffic is elevated to a more computationally intensive second layer. This research

aims to provide a comprehensive analysis of the proposed approach and its benefits, comparing it to existing approaches and identifying areas of improvement.

1.1 Background

In cybersecurity, Control Flow Integrity (CFI) refers to forcing users to follow the intended flows of execution when interacting with an application [3]. Many different types of attacks rely on control flow redirection to alter code behavior. This can vary from injecting malicious code and tricking the application into executing it (such as writing shellcode to an executable section of memory) to redirecting the process' flow to functions that would not normally be callable (such as a common buffer overflow attack on a binary application).

Enforcing CFI in compiled applications and binaries has been a field of significant interest in the computer security community [3]. Multiple CFI defenses have been proposed and widely adopted in compiled applications, from Address Space Layout Randomization (ASLR) [7], stack canaries [8] and PIE [9].

Control flow based attacks are not unique to compiled applications [10]. Many websites are susceptible to and often victim to control flow redirection attacks, from Cross-Site-Scripting (XSS), PHP code injection and web-shell manipulation [11]. Unlike compiled programs, however, the web server model handles computation in a "ad hoc" method, triggering different execution paths based on incoming HTTP requests from many different clients, often handling many at a time. These levels of abstraction make implementing the "traditional" CFI of compiled applications to web services difficult [3]. The ad hoc execution of scripts and complexity in tracking individual user's execution history (due to HTTP being a stateless protocol) make web applications flexible, but also limit our ability to enforce expected control flow.

1.2 Project Description

We propose Warden, a system designed for the Single-Use-Server (SuS) web architecture [1, 2] that aims to apply the tenets of CFI to web execution. The SuS architecture offers a significant advantage in this endeavor, in that it allows us to monitor individual user's execution. SuS operates by creating a unique instance of a web server for each client connection in a Docker container. These containers are managed by the system to defend against many privilege escalation and information leak attacks through back-end proxies and correlating permissions to containers. The separation of client execution into neat containers gives Warden the opportunity to identify and isolate suspicious and/or anomalous users.

Warden is comprised of a two layer CFI system, with a fast and lightweight bloom filter CFI detection system and a slower but more robust hashmap CFI system reserved for function calls labeled suspicious by the lightweight system. In context of this project proposal, "lightweight" CFI systems will refer to CFI systems that are lightweight and fast, but are not necessarily complete, meaning they can flag a function call as suspicious, but not confirm whether the transfer is valid or not. A "robust" CFI system will refer to a more computation intensive CFI system that can validate a function call by investigating WordPress source code. Our implementation allows for experimentation related to the CFI method used, rigidity and precision of the historical data collected, and associated memory, latency, and CPU overhead.

We believe that this work, while it will be engineered for the SuS architecture, could provide real novelty to the field of web-based CFI. A lightweight and robust CFI system could be adapted to a traditional web server architecture with some engineering.

1.3 Research Questions

Warden aimed to iterate on the concept seen in prior works [12] of using historical execution data to detect anomalous execution, and did so using a novel layered approach to CFI detection. In order to direct our efforts we developed two research questions, each with one sub-question:

1. Can historical execution context be effectively used to determine when execution becomes anomalous?
   (a) How often would a system built around such context correctly identify malicious execution? How often would it misclassify benign execution as malicious?
2. Does a resource-conscious "lightweight and robust" CFI approach such as Warden incur lower operating costs than a more traditional, resource-apathetic CFI approach?
   (a) What level of CPU, latency, and memory overhead is introduced by a layered "lightweight and robust" CFI systems compared to both Single Use-Server systems and traditional LAMP architectures?

2 Related Work

Warden was designed to work within the SuS architecture, and builds off of previous efforts in the field of web application CFI. To fully understand the environment Warden was built for and the goals it was built around, an understanding of SuS and the state of web CFI must be attained.

2.1 Single Use Servers

The SuS web-server model was first introduced as a way to prevent the confused deputy problem and lateral attacks (attacks from one client against another)[2]. SuS uses Docker containers to prevent interaction between clients and to confine one client to only the privileges that they should have. A user would not be able to access resources outside their level of permission. This resource denial is handled by middle boxes outside of the untrusted client containers that handle authentication and requests to access data. SuS achieves this by using multiple "middle boxes," or devices and processes that sit between different server components such as the client, container, and database. These middle boxes allow SuS to divide traffic on a user-level, monitor incoming traffic and enforce permissions when interacting with backend resources.

In addition to the security provided by the SuS architecture, one of the SuS middle boxes generates logs of static function calls (as opposed to dynamically generated calls) [1] [13]. To supplement this, PEGASUS provides a PHP function profiling module that logs all function calls as well as some parameters relating to the calls such as call location [13]. We modified this extension to enable Warden to inspect PHP function call logs.

2.2 Control Flow Integrity

The "control flow" of an application typically refers to what functions and code segments are executed as a result of user interaction and what order those segments of code are executed in relative to one another [3]. CFI has been a popular area of cybersecurity research for over a decade. Many types of attacks on compiled applications utilize Control-Flow Hijacking through memory corruption to manipulate applications to execute in unexpected and possibly dangerous ways. CFI has traditionally been applied to such compiled applications, using a variety of mechanisms and techniques to try and provide assurance of CFI without sacrificing performance. In traditional compiled CFI, a custom compiler is built which performs static analysis on a binary's source code at compile time. It uses this to generate a Control Flow Graph (CFG), which defines all expected behavior of the given application. The compiled binary is then built with a security system that references this CFG to ensure the application does not deviate. The level of granularity and completeness that CFI implementations can achieve significantly impact their effectiveness as stopping Control-Flow Hijacking attacks. A CFI mechanism that achieves very coarse-grain enforcement can leave an attacker more than enough room to hijack a program, while very fine-grained enforcement system can incur significantly more overhead in either latency or memory, and can still remain exploitable [14]. The manner in which a CFG is constructed and enforced can vary wildly and has been the topic of multiple research

endeavors [3].

In January of 2018, Burow et al. published a survey of well known CFI implementations at the time along with their respective benefits and shortcomings [3]. They explain how CFI is comprised of an `Analysis` phase where a system learns from an application then approximates a CFG (or other ground-truth data structure), followed by an `Enforcement` phase where the application is monitored in production and its execution is compared to the approximate ground-truth. They explain how CFI does not solve the inherent bug which allowed for Control-Flow Hijacking, but rather acts as guard rails of sorts, stopping vulnerable programs from being manipulated to any significant degree.

The survey focused on traditional compiled CFI implementations, and distinguished different works both qualitatively and quantitatively, as the researchers found many compiled CFI works incurred similar performance overhead, ranging from 0%-20%. They found that even in compiled applications a theoretically complete CFG could not be built for "nontrivial programs." Many compiled CFI systems ended up overestimating the CFG, allowing for unnecessary edges that attackers could utilize, and increasing the precision of the mechanism (finer-grain enforcement) negatively impacted the system's performance. They also acknowledge the incremental nature of CFI technology advancement. Most mechanisms they researched were built off of and inspired by the previous works.

As research into CFI has expanded, the methods and goals of the practice have been adapted to help address some of the issues mentioned by Burow et al., such as performance, completeness, and complexity. Coarse-Grained Syscall-Flow-Integrity (SFIP) [15] is a promising effort in the CFI field that takes a non-traditional approach to its `Analysis` and `Enforcement` phases. SFIP focused on limiting a malicious actor's ability by enforcing CFI at a system call level. Similar to how Warden attempts to apply CFI to a PHP function level, SFIP aimed to test if the concept of CFI could be applied to the user-kernel boundary. Like those research by Burow et al., SFIP was a CFI enforcement system built for compiled programs. Unlike many mechanisms, SFIP combined a syscall state machine and syscall origin mapping to achieve its own form of CFI without needing to construct an entire CFG.

SFIP used static analysis of source code at compile time to create its CFI data structures, recording where each syscall was located in memory and what syscall had preceded it, allowing the kernel to check each syscall executed at runtime against trusted behavior. The researchers found that SFIP added a 13.1% and 1.8% overhead in syscall execution time in micro- and macro-benchmark tests respectively. They also found that compilation time of binaries was increased by a factor of up to 28.

While SFIP was not an attempt to enforce CFI at a web server level, the strategies and findings of

the study are nonetheless helpful and related to our work with Warden. The overhead, complexity, and lack of guarantees discussed by Burow et al. led the developers of SFIP to approach the problem using a trusted set rather than a full model, and to apply the tenets of CFI to a different level of execution. SFIP offers insight that CFI can be thought of and enforced using non-traditional methods. Rather than constructing a complete CFG and validating full sequences of function transfers, SFIP looks to trade theoretical completeness for performance and ease of use by recording only short sequences of syscalls and their location in memory. Similarly, Warden aims to enforce a type of CFI using PHP function origination, that is verifying a function call comes from a valid PHP script and expected line number. However, the methods used to obtain such integrity and the definition of success must be adapted to the more volatile and dynamic environment of web applications. Related fields such as anomaly detection have used methods like set-theory, virtualization, and log analysis to detect deviations from expected behavior, and these methods may be useful when dealing with web CFI [16, 17].

Burow et al. address the complications that arise when trying to apply CFI to code bases like web applications. Just-in-time compiled and interpreted code lack many of the advantages than traditional compiled code inherently has. They acknowledge that such programs are "inherently dynamic" and present significant problems when trying to construct a complete CFG [3]. Features like dynamically executed strings through functions like `eval` in Python or `call_user_func` in PHP restrict both static analysis' effectiveness and a CFG's complexity drastically. It is for these reasons that works similar to Warden which try to apply CFI principles to web applications have all utilized non-traditional methods in their `Analysis` and `Enforcement` phases.

2.3 CFI in Web Applications

Web frameworks, specifically those like the common Linux, Apache, MySQL, PHP (LAMP) systems utilizing PHP scripts for back-end computation, are not conducive to traditional CFI methods. In a compiled binary, valid function transfers can be programmatically enumerated using a variety of methods, creating partial Control Flow Graphs (CFG). Rules and security measures can be compiled into the binary to enforce said CFG, and mark any function transfer that violates its paths. Web servers on the other hand often respond to requests in a stateless manner, handling HTTP requests from multiple clients as they come in and invoking functions from PHP scripts on the fly. This makes it difficult to create a definitive CFG and track each client's path along it.

For these reasons, implementing "traditional" CFI in a web framework has not been the subject

of as many research projects as its compiled counterpart. However, there have been efforts to incorporate the tenets of CFI to web development. Each project has brought its own methods of determining trust and levels of monitoring granularity to the field. A simplified summary of these differences can be seen in Table 1.

Related Work	Warden	Saphire	ZenIDS	Ghostrail
Protection	Detection	Enforcement	Detection	Enforcement
Analysis Level	PHP Function Calls	Syscalls	PHP Opcode	Web Requests
Connection Context	Per User	None	None	Web-Session
False Positives	None	None	Some	17%
False Negatives	Adjustable Rate	**Unknown**	**Unknown**	**Unknown**
Overhead	535%	$\leq$2%	5%-10%	250-360 ms
CFI Method	Historical Data	Historical Data	Learned CFG	Sandboxing
Execution Env.	PHP Runtime	Kernel-User Space	PHP Interpreter	HTTP Requests

Table 1: Comparison to Related Web CFI Works

2.3.1 ZenIDS

ZenIDS is an intrusion detection system that aims to be effective against remote code execution (RCE) attacks. Implemented as a PHP extension, ZenIDS builds a trusted profile of HTTP requests through a pre-training period. The pre-training period consists of both static and dynamic analysis, much like Warden. It is common in PHP for functions and scripts to be dynamically created via anonymous functions. The execution is being monitored by 5 main PHP hooks [18]. Hook 1 is used to compile PHP code into an opcode sequence, Hook 2 is used to evaluate if the compiled opcode exists within the trusted profile using canonical-name and Hook 3 executes the opcode. Hook 4 and Hook 5 interact the with application state by storing and loading the application state respectively.

When a PHP script is interpreted, the PHP code is converted into an opcode with H1 sequence that is added to the trusted profile, with H2, as a control flow graph, detailing any jumps to other sections of PHP code. Using these hooks, ZenIDS is able to detect any changes to the application state and will initiate different phases based on how the privileged user changed the state, either a data expansion event or a code expansion event. The data expansion event will add new control flow to the trusted profile. The code expansion event is more complicated, involving checks to see if the code is trusted. H4 updates the database during a code expansion event including information such as if the call stack was seen during the training period and if the code generation was influence by user input.

ZenIDS has some similarities with WARDEN's robust CFI implementation. ZenIDS builds a trusted profile existing of PHP opcode control flow graphs while WARDEN stores trusted PHP function calls with

line number granularity. WARDEN's main feature is the preliminary lightweight CFI followed by the robust CFI. It is entirely possible to re-engineer WARDEN's robust CFI to utilize ZenIDS. While WARDEN utilizes a PHP extension to ex-filtrate logs from within the SuS containers, ZenIDS lives exclusively in the PHP environment.

2.3.2 Saphire

Saphire proposes another solution to increased web-application security by leveraging the principle of least privilege (PoLP) specifically for interpreted languages [12]. Saphire profiles scripts and compiles a set of system calls that each script needs to be able to execute when functioning correctly, and creates an allow-list for the script. Then, using `seccomp`, a given script will be constrained to only have access to the function calls in its allow-list.

Saphire's main improvement over ZenIDS was eliminating false positives, with a minimal performance overhead. Saphire was able to prevent all 21 tested arbitrary code execution vulnerabilities, while throwing no false positives. However, one distinct weakness in Saphire that Warden attempts to address is granularity. Saphire points out that attacks could be crafted that used the allowed system calls of a specific script for malicious purposes. Specifically, because Saphire compiles a list of allowed system calls per script, there is potentially a lot of room for illegitimate actions, such as writing to the file system from a script with permission to do this.

2.3.3 Ghostrail

Ghostrail attempts to enforce CFI on web applications by securing yet another attack vector: HTTP requests [4]. Rather than enforcing server-side computation flow integrity, Ghostrail attempts to limit the GET and POST requests malicious users are able to submit to the site in such a way that they are forced to follow expected behavior. The work has been iterated on multiple times by the original researchers, but as of its most recent publication focuses on eliminating HTTP request race conditions, parameter poisoning, and unsolicited request forgeries [5, 6].

Ghostrail enforced its version of CFI by analyzing the DOM object for a given user session and keeping track of the previous HTTP request and web page accessed. It uses this information to determine whether any incoming request has expected paramaters and is a reachable request from the page the user was previously on. This acted as a sort of "static analysis," forcing users to follow links and make requests in the expected order. There are however many valid dynamically crafted requests (often created using `Ajax`

or `JavaScript`) that such static analysis would not recognize. To accommodate for this, the user mouse clicks and inputs were monitored through injected code and replicated in a controlled server side replica, or "sandbox." Ghostrail would then compare the HTTP requests generated by the trusted sandbox web page with the requests received from the user's browser. If they matched exactly then the request was considered safe and allowed.

As mentioned previously, Ghostrail has gone through many iterations, but was originally implemented as a reverse proxy using `Node.js`. When tested for effectiveness, the researchers found Ghostrail to have an average false positive rate of 17.57% with a maximum recorded rate of 60%. This means on average approximately 1 in every 5-6 valid HTTP requests were seen as malicious and blocked. The authors attribute this to the dynamic nature of web applications and customized elements unique to the web application. Some applications included features like client fingerprinting and random numbers seeded with system attributes in their HTTP requests, both of which caused the sandbox and real user to generate different requests with the same actions. The researchers also found that users experienced a round trip latency overhead of around 250-360ms per web page while Ghostrail was active. On top of this, due to Ghostrail's need to validate each HTTP request and record the client's previous location, the `Cache-Control: no-cache` header was added to every response sent to the client to keep their browser from caching web pages. This ensured any latency added due to Ghostrail was experienced for every request.

Ghostrail is one of the only formal attempts at enforcing web CFI at the HTTP request level and provides a clear example of the difficulties of adapting CFI to the dynamic field of web applications. Due to the event-driven nature of web languages like JavaScript, Ajax, and PHP the researchers found it necessary to include dynamic analysis in the form of sand boxing user interaction. In contrast, Warden aims to leverage historical execution data to enforce CFI on the PHP function level. Dynamic analysis through historical data can be implemented in a variety of ways and does not require the simulation of the sand boxing method. Additionally, by operating in the PHP layer rather than the HTTP layer, Warden does not have to accommodate for many of the issues Ghostrail encountered, such as fingerprinted HTTP requests unique to each client. Works like Ghostrail are important as they highlight the fact that web CFI can be applied to entirely separate layers of architectures, with HTTP request filtering taking the role of a "front-end" CFI and works like ZenIDS, Saphire, and Warden enforcing "back-end" CFI.

3 Design

We built Warden with three design goals in mind: **Speed**, **Administrator-Friendliness**, and **Effectiveness**. We wanted Warden to introduce as little latency as possible so that it is able to be implemented without affecting the user experience. In a similar vein if the system is not administrator friendly then it would be difficult to justify implementation. Finally we built Warden to be effective, correctly identifying anomalous execution and efficiently warning administrators.

We address the first design goal, speed, with our "layered CFI" approach, which aims to alleviate some of the overhead CFI enforcement systems by using a fast "lightweight" system which approves the majority of traffic, and only elevating to the slower "robust" system in suspicious cases. Secondly, we designed Warden to be an administrator friendly system. Security systems can quickly become cumbersome to administrators if they have a high false positive rate, that is, raise alarms about events that are actually perfectly benign. Not only do false positives waste analysts' time, but they promote an environment where security alerts are taken less seriously since they have a chance to be false alarms.

3.0.1 Threat Model

Warden is a system implemented within the SuS architecture, but is designed to be usable in traditional LAMP servers. It is intended to detect anomalous PHP execution by leveraging historical execution data. We assume an attacker acting within a compromised SuS container. The attacker has the ability to execute arbitrary PHP code, and can download, upload, and edit PHP source files on the container. We assume the SuS architecture is operating as intended, meaning the attacker is not able to perform any privilege escalation attacks and cannot influence a system outside if their container. We exclude client-side remote code execution attacks, such as Cross-Site Scripting. Warden is a server-side monitoring system, intended to detect injected or altered code running on the web server. Finally, we exclude attacks against our trusted computing base (TCB). The TCB consists of the operating system and all components of the modified SuS architecture shown in Figure 1, including the Warden log parsing and CFI processes.

3.0.2 Design Components

The Warden system adds a few new components to the SuS architecture, shown in red in Figure 1. We edited and used the PHP profiling module from prior SuS work to intercept PHP function calls and exfiltrate them out of containers as logs. The extension serialized each log and sent to a server which parsed

them and added them to a processing queue.

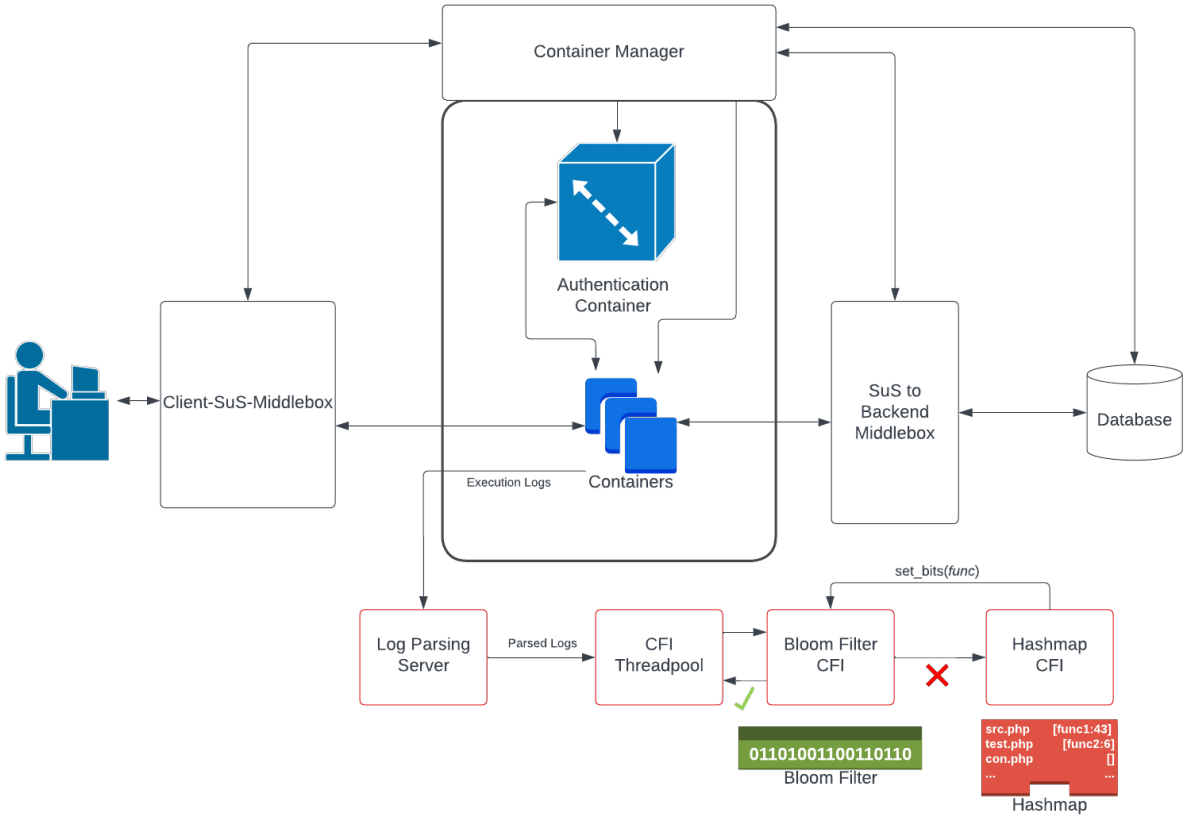


Figure 1: Modified SuS Architecture, with Warden system outlined in red

When we first instantiate SuS, a managing process creates a threadpool of CFI worker threads. As the parsing server generates logs, the manager assigns them to worker threads for investigation. Each thread then starts by running lightweight CFI (bloom filter shown in the diagram) on the function call. If the "lightweight" CFI system marks the function call as suspicious, it elevates the call to the "robust" hashmap CFI.

For each function call executed, the PHP log constructed by the PEGASUS [13] PHP extension contains callee name, caller script, line number, IP address of web server, and function parameters. Warden's CFI systems inspect the function and script names as well as the line number, giving us line number granularity. Warden could record the parameters of the function call for increased context sensitivity, though the set of trusted function calls stored by Warden would increase in size, incurring additional overhead.

The "lightweight" CFI was implemented as a bit array bloom filter, chosen for its small size and speed. The filter will check if each incoming log has been serialized in the bit array previously and elevate the log to hashmap CFI if it is not found. More information on the bloom filter data structure and rationale

can be found in the implementation section.

The hashmap CFI will then check its historical data to see if the function call is valid. If the function call is not found in the trusted set, the worker thread will investigate the WordPress source code to determine if the function call's name appears along the expected line in the PHP script. If the function call is found to be valid, the hashmap CFI updates the lightweight data structure to trust the function call, and the thread moves onto its next assignment. If the thread deems the function call invalid, the event is recorded for future investigation, although the system could be configured to halt the malicious container immediately. A comparison of the hashmap and bloom filter implementations' features can be see in Table 2

Feature	Hashmap	Bloom Filter
Line No. Granularity	✓	✓
Source Code Analysis	✓	**X**
Pre-Populated	✓	**X**
Ground Truth	✓	**X**
False Positives	**N/A**	**X**
False Negatives	**N/A**	✓
Mutex Required	✓	**X**

Table 2: Comparison of our CFI Implementations

The CFI systems can be trained beforehand with dynamic analysis. Dynamic analysis refers to running the system with benign background traffic and recording all function calls that are received, saving them as trusted for reference later when in enforcement mode. In order to avoid marking anomalous/malicious execution as trusted, system admins must perform dynamic analysis in a pre-production testing environment. Admins can utilize techniques similar to the methods discussed in the Saphire paper [12], including but not limited to:

1. Replaying HTTP requests from users sessions on a monitored browser
2. Running any test suites released with web applications.
3. Web crawling the application, with both an authenticated and unauthenticated agent
4. Releasing the web application in a pre-production manner, giving access to only trusted users

Ideally, Warden would be trained on a set of functions calls as close to the complete trusted functionality of a web application as possible. We trained our model using Apache JMeter, an industry standard for simulating web traffic and stress testing systems. We recorded HTTP requests made during

a human-driven session on the site, then replayed that traffic to simulate benign user interaction. In order to simulate Warden learning from a complete set of function calls we performed some experiments on the subset of site functionality that our simulated traffic covered.

4 Implementation

We implemented Warden in C with some utilities written in Python. The researchers in PEGASUS [13] designed a PHP extension that profiled execution by recording a log every time a function was called in the PHP runtime. A PHP environment with a modified version of the same profiling extension sends function logs to a log-catching process listening on the host machine. This tool allows Warden to operate outside of the client containers, which are not a part of our TCB. These logs are then added to a queue of logs ready for CFI processing. This is depicted in Figure 2.

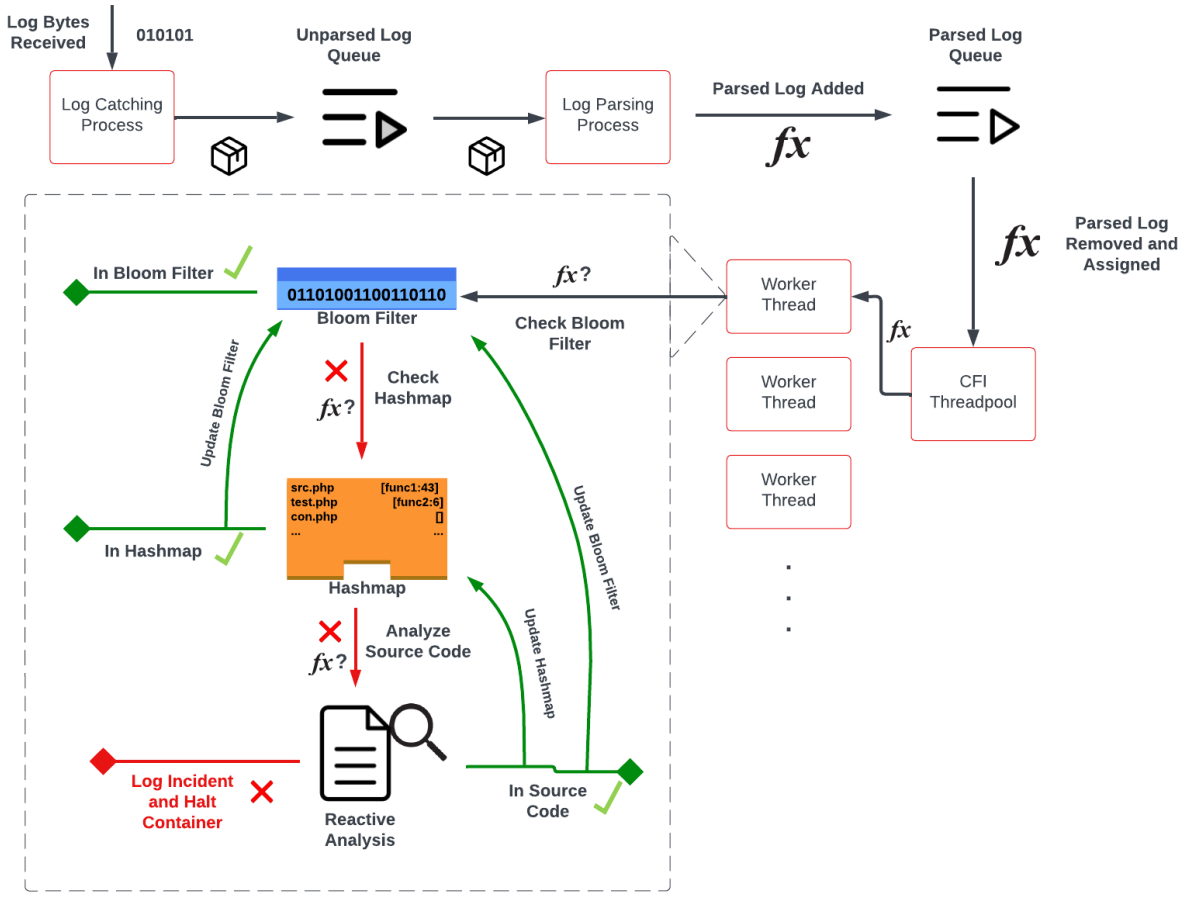


Figure 2: Warden System Implementation

The manager process is multi-threaded in order to handle the volume of incoming logs. On startup, it instantiates a threadpool of CFI worker threads. Each thread waits for the logs to be received, then removes a log and begins the CFI analysis. A high level algorithm for a Warden CFI worker thread can be seen in Algorithm 1.

Algorithm 1 Warden CFI Worker Thread

```
1:  procedure CHECKCFI
2:
3:  LightweightCFI:
4:      functionCallLog ← dequeuePHPLog()
5:      if inBloomFilter(functionCallLog) then
6:          end
7:      else
8:          goto RobustCFI
9:
10: RobustCFI:
11:     if inHashmap(functionCallLog) then
12:         updateLightweightCFI(functionCallLog)
13:         end
14:     else
15:         if reactiveAnalysis(functionCallLog) then
16:             updateLightweightCFI(functionCallLog)
17:             updateHashmap(functionCallLog)
18:             end
19:         else
20:             logIncident(functionCallLog)
21:             end
```

First, the worker runs the lightweight CFI on the function call in the log (in the diagram the lightweight implementation used is bloom filter). If the worker finds the function call in the bloom filter, it considers the log valid and the thread waits for its next assignment. If the function call is not found in the bloom filter, the thread elevates the log to robust CFI.

In order to make the robust CFI as fast as possible, a hashmap is utilized to store function calls that have been investigated previously (either during production or during the training phase) and are trusted. The script name is used as a key to hash, and the value is a linked list of function calls that are allowed. If the function call is found in the hashmap, it is considered trusted, and the bloom filter is updated so that future calls of the same kind will not be elevated to robust CFI. The thread then waits for its next assignment.

If the worker does not find the function call in the bloom filter, the hashmap CFI begins the more computationally intensive process of investigating the application's source code to determine if the log data is valid. The worker first parses the application's root directory to determine if the script filename given in

the log exists. If it does, it opens the file and scans to the line number given by the log. It then searches the line for the function call given in the log. If all of these steps are passed, the worker thread considers the log valid and updates the data structures to avoid future false positives before getting its next assignment. If the source code analysis finds that the function call is not valid, Warden logs the event. Here Warden can act as the system administrator desires. Warden could halt the SuS container, monitor it more closely, or just kill it outright in order to stop any further abnormal behavior.

4.1 Data Structures and Rationale

When deciding how to organize the multi-layered CFI system, we considered five performance goals.

1. **Security**. The Warden process responsible for identifying whether a function call is anomalous must be within our TCB.

2. **Speed**. The goal of a multi-layered CFI system to is to allow benign traffic to be quickly approved by the first layer, and only incur the speed loss when traffic is deemed suspicious.

3. **Lightweight**. The memory overhead associated with Warden scales with number of users without overwhelming the system.

4. **Low False Positive Rate**. A false positive correlates to Warden raising an alert over a function call that was not anomalous. In order to make the system as trustworthy and administrator friendly as possible, the false positive rate remains low.

5. **Controllable False Negative Rate**. A false negative correlates to Warden approving a function call that should have been marked as anomalous. The false negative rate of the system should remain as low as possible, and is ideally controllable, allowing for administrators to trade off system resources for a lower false negative rate or vice versa.

Warden handles all CFI processing and data storage outside of the container running PHP. This in turn incurs certain overheads associated with sending log data to an external process. Regardless of this, we deemed it preferable to run Warden outside the PHP runtime in order to conserve memory and keep the system within our TCB.

In the SuS architecture, each client Docker container has a unique PHP runtime instance, meaning each container will be running a unique copy of the PHP profiling extension, which do not share process memory. If Warden was to be written as a PHP extension, each container would either have to have a

unique copy of the data structures used for CFI. Allocating the memory to give each container its own copy of the CFI data structures would quickly become memory intensive, especially as SuS is applied to larger and larger applications with many clients. The duplicate bloom filters and hashmaps would also quickly lose synchronicity, as each would only learn from a single client's behavior.

Secondly, and arguably more importantly, if we implemented Warden as a PHP extension, we would have to write it to operate entirely within the container, which is not part of our TCB in the SuS architecture. If an attacker compromised the PHP runtime of the container, they could easily turn off or silence Warden. Therefore, we elected to implement Warden outside of the containers, running instead on the SuS host machine. This allows us to share single copies of our data structures between threads while operating within our trusted computing base.

The bloom filter data structure was chosen for the "lightweight" CFI due to its speed, small size, and inability to produce false positives. A bloom filter is populated using a set of "entries." The filter can then be used to test if an entry was not a member of the original set it was trained on. A bloom filter maintains an array of bits (set to 0 initially) of size $\boldsymbol{m}$, and has a predetermined number of unique hash functions, $\boldsymbol{k}$. For a given entry, $\boldsymbol{X}$, each hash function is run on $\boldsymbol{X}$ and mapped to an index in the bit array. If $\boldsymbol{X}$ is being inserted into the array, each bit it hashed to is set to 1. An example of adding $\boldsymbol{X}$ to a $\boldsymbol{m}$,$\boldsymbol{k}$ bloom filter is shown in Figure 3.

In the case that the bit of an entry's hash is already set to 1, nothing is changed. In order to test if an entry was present in the set the bloom filter was trained on, Warden simply runs each hash function on the entry and checks if each corresponding bit is set to 1. This structure allows for false negatives. A false negative would equate to checking if an entry $\boldsymbol{Y}$ was in the set $\{\boldsymbol{X}, \boldsymbol{Z}\}$ and getting the response that $\boldsymbol{Y}$ is in the set. This could happen if the union of the hashed indices of $\boldsymbol{X}$ and $\boldsymbol{Z}$ contained all the hashed indices of $\boldsymbol{Y}$, that is:

$$\forall\ h_k(Y):\ h_k(Y)\ \in\ (h(X)\ \cup h(Z)) \tag{1}$$

Where h(X) represents all hashed indices of entry X. This is visually represented in Figure 4, where a false negative's occurrence becomes clearer.

In terms of our work, this correlates to an illegal PHP function call appearing as valid when checked against the bloom filter. This is clearly a security concern, however the theoretical false negative rate of a bloom filter can be calculated and controlled using the the following equation [19]:

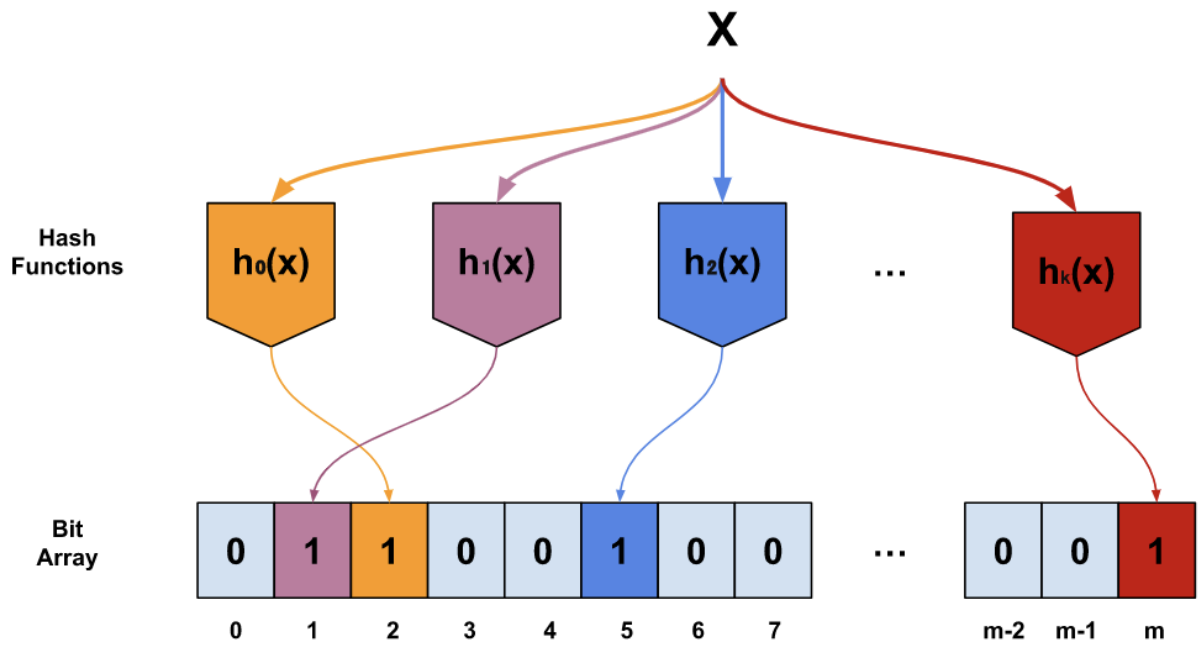


Figure 3: Entry X being added to bloom filter with k hash functions and m bits.

$$P = (1 - [1 - \frac{1}{m}]^{kn})^k \tag{2}$$

Where P is the probability of a false negative, m is the size of the bit array, k is the number of hash functions, and n is the expected number of entries that the filter is trained on. This fulfills our goal of a controllable false negative rate, as it allows the administrator to control the false negative rate of the bloom filter, trading increasing filter size in order to lower the number of anomalous functions going unnoticed.

The real advantage of using a bloom filter comes from its inability to report a false positive. A false positive occurs when an entry that was in the original set is tested and the filter reports that it was not in the set. If all bits associated with an entry's hashes are not set to 1, the bloom filter can report with 100% certainty that said entry was not in the set the filter was trained on. In the context of Warden, this means the bloom filter will only ever raise an alert and elevate a function call to robust CFI if the function call it received was not a trusted call, and will never raise a false alarm over a benign function call.

This is a significant part of making Warden an administrator friendly security system likely to be deployed in a production network. False positive alerts are sources of constant annoyance for security analysts, and can often cause confusion and clutter, possibly to the point that genuine alerts are overlooked

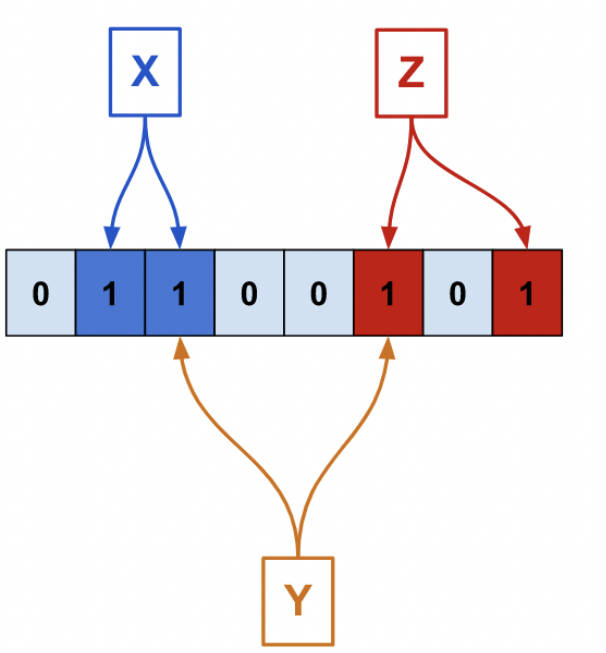


Figure 4: A false negative occurs when Y is queried on a bloom filter with {X,Z}

by overwhelmed administrators. If the rate of false negatives can be controlled and remains relatively small, some system admins may prefer to have a low false negative rate and no false positive rate instead of no false negative rate and a high false positive rate.

The bloom filter also provides the benefit of being extremely lightweight. The structure itself is composed of a bit array which can be expanded without becoming a burden on the system. The number of hash functions run for each entry is the main bottleneck in terms of latency, but the hash functions do not have to be cryptographically sound as long as they are decently sparse for index marking, so they can be chosen to be very fast. The bloom filter offers a unique set of features to CFI computation, as it has a controllable false negative rate, a 0% false positive rate, and is lightweight both in terms of memory and computation time.

Finally, we implemented a hashmap which robust CFI would check before performing source code analysis. This was done in order to alleviate the performance costs associated with the increased computation and to let the system learn dynamically generated functions in PHP. The hashmap used script names as keys and a linked list of valid function calls as values. As the system was run, it would populate the hashmap with function calls it found trustworthy, or was told to consider trustworthy by system administrators. For an application with an exhaustive set of legal function calls, this would minimize the amount of repeated computation and speed up normal traffic on the web server.

5 Evaluation and Results

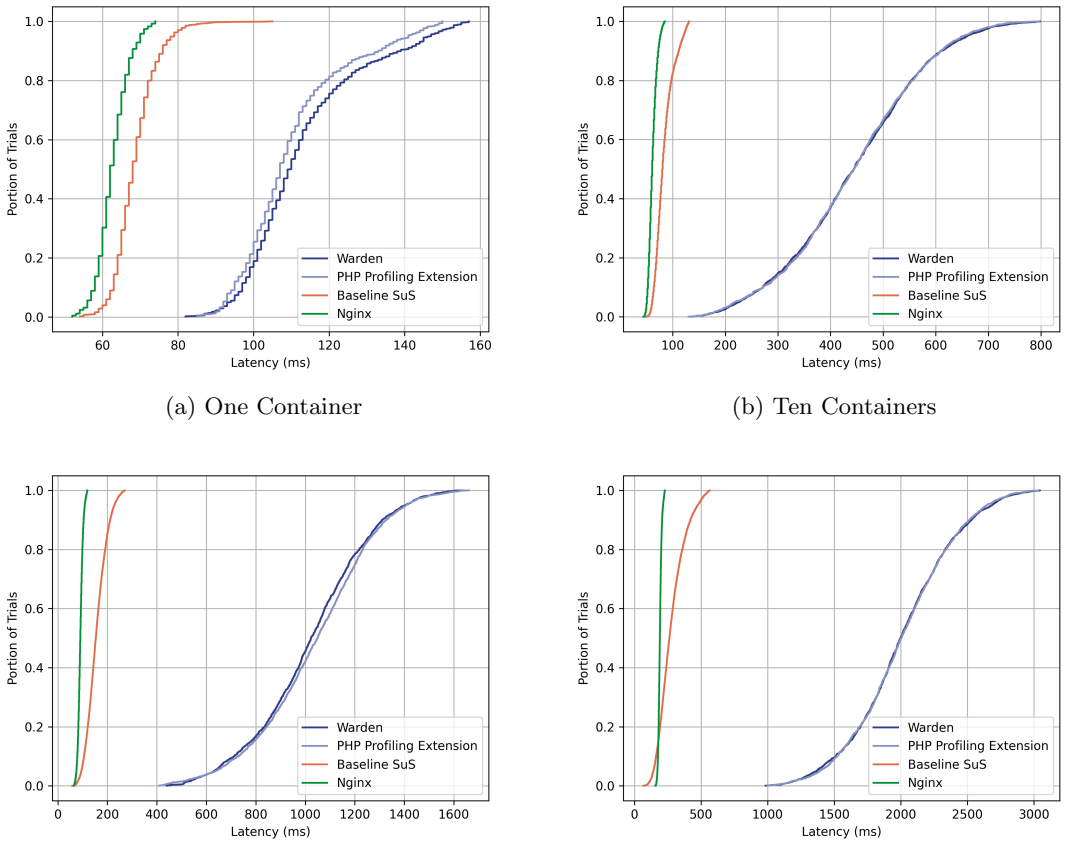
In order to evaluate Warden as a system we looked at its effect on performance when compared to the original Single-Use-Server system as well as when the PHP profiling extension is enabled. In addition to the performance of Warden we are interested in examining its effectiveness. We evaluated effectiveness by simulating different methods of server side code injection to ensure Warden recognized the anomalous execution. We also measured a theoretical false positive rate by running Warden with normal traffic and recording its behavior.

5.1 Performance

To assess Warden's performance, we conducted a series of experiments that analyzed its impact on latency, server-side CPU and memory (RAM) usage. By using Apache JMeter we were able to simulate user activity on a web-server running WordPress 5.1.1. We conducted these experiments in four phases consisting of an increasing number of simulated users, in the order of 1, 10, 25, and 50 users. The means, medians, and standard deviations of each test are found separated by testing class in the Appendix A. Our goal of determining if a multi-layered web-based CFI system is viable drove us to a particular testing methodology. We tested both SuS with Warden and SuS with only the PHP extension and log exfiltration system separately to better understand how much each component contributes to latency and overhead. Refer to Figures 5, 6 and 7 to see how the different components of Warden contribute to its performance in the four testing phases. We ran the tests on a virtualized machine with 16GB of RAM and a 2.200 GHz Intel Xeon Skylake-Server-IBRS CPU with 4 cores. We configured Warden to run with a 10 MB bloom filter and 5 hash functions. For a liberal estimated range of 1 - 5 million unique possible function calls, a theoretical bloom filter yeilds a false negative rate of 8.2e−5% to 0.139% respectively.

5.1.1 Baseline SuS

To establish a baseline, we collected performance data on the Single-Use-Server Model with Warden completely disabled. When testing with 1, 10, 25, and 50 containers, we observed average latencies ranging from 68.71-278.09ms across 7 different endpoints as well as ranges of 21.71-98.61% CPU utilization and 1985-3020MB of RAM usage. As more users were added, we observed a slight increase in latency shown in Figure 5. CPU utilization and memory consumption followed a similar trend seen in Figures 6 and 7.



(a) One Container

(b) Ten Containers

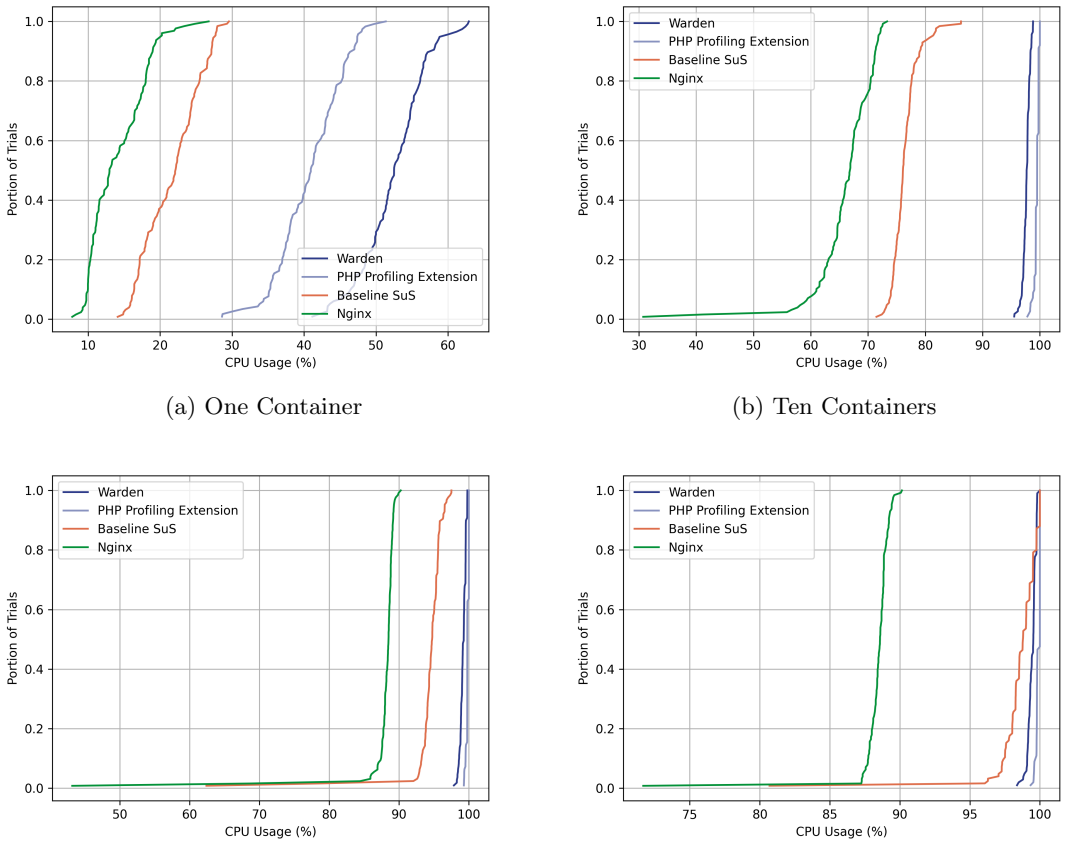
(c) Twenty-five Containers

(d) Fifty Containers

Figure 5: Latency in ms

5.1.2 PHP Profiling Extension

The next component we observed was the PHP profiling extension. As this extension works on top of the SuS architecture we expect it to perform worse when compared to baseline SuS. Seen in Figure 5 there is a significant increase in latency when the extension is introduced into the system. There are a few culprits in the extension implementation that are responsible for this. The first is the socket type used to ex-filtrate logs from within the container to Warden's manager system. We used a TCP socket in the `AF_INET` family to send the logs over the network in favor of a `UNIX` socket in order to avoid modifying the SuS Docker settings [1]. Another suspect is the lack of multi-threading in the extension to send to logs. We had initially designed the extension to exfiltrate the logs in a multi-threaded fashion through the use of the `POSIX` library function `pthread_create`, however due to complications with our environment we were unable to get `pthread_create` to run as expected and had to send the logs in series rather than in parallel, increasing latency. Other example environments have shown that `pthread_create` can work in PHP extensions. In



(a) One Container

(b) Ten Containers

(c) Twenty-five Containers

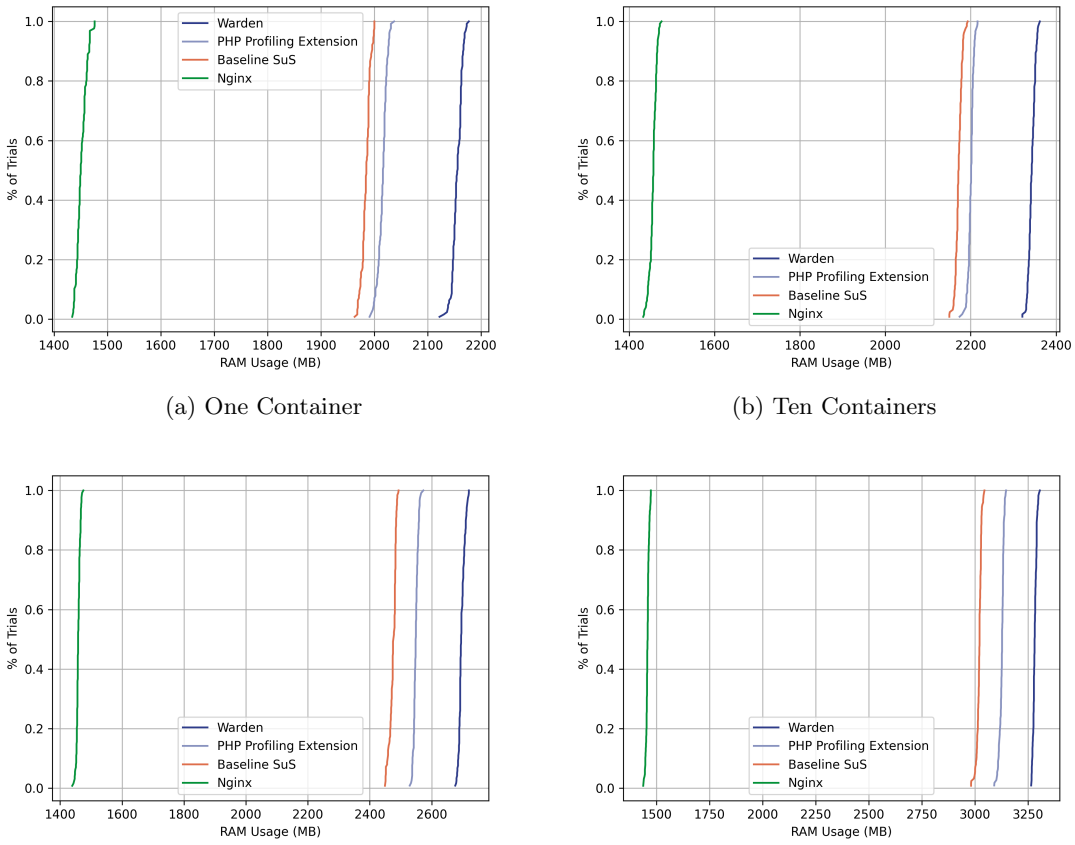
(d) Fifty Containers

Figure 6: CPU utilization in percentage

the future, an optimization could involve reconfiguring Warden to support `pthread_create` in PHP runtime extensions.

5.1.3 Warden at Full Capacity

The final component enabled was Warden's full CFI capabilities. We built the CFI to run in parallel with the users activity as to minimize the effect on latency. It is apparent from the CDFs that the curves for the PHP extension and Warden are nearly identical for latency as shown in Figure 5. The CPU was near 100% utilization for every test with 10 or more users. However, the CDF of CPU utilization with 1 user shown in Figure 6 shows the CPU usage of the PHP extension to be 42.00%, and the CPU usage of Warden to be 53.14%, implying a significant portion of Warden's CPU overhead is accumulated during the exfiltration and parsing of PHP logs. However, Warden's CFI requires significantly more memory to store the function calls in the hashmap. We decided to allow for Warden to store the function relations instead of



(a) One Container

(b) Ten Containers

(c) Twenty-five Containers

(d) Fifty Containers

Figure 7: RAM utilization in megabytes

performing static and dynamic analysis for every log the system encounters, favoring higher memory usage rather than CPU utilization.

5.2 Effectiveness

The version of WordPress that we trained Warden on (v. 5.1.1) did not have any publicly documented exploits to inject and execute arbitrary PHP code server side. Prior versions of WordPress and their associated plugins however had multiple vulnerabilities which allow for such server side code injection. Server side code injection has many forms, all of an adversary can exploit if they have Arbitrary File Download (AFD) and Arbitrary File Upload (AFU) capabilities. Both AFD and AFU vulnerabilities have been published in prior versions of WordPress and WordPress extensions. For example, in version 1.48 of the Files plugin unauthenticated users were able to upload files from their machine to the server [20]. Similarly, version 1.1.1 of the History Collection plugin neglected to filter HTTP requests in such a way that it allowed

unauthenticated users to download any file from the server [21]. The combination of AFD and AFU would allow any attacker to inject PHP scripts and modify existing scripts on a web server.

5.2.1 Testing Code Injection

In order to test Warden's effectiveness against the different code injection methods, we separated such attacks into different "classes" of code injection based on the criteria Warden monitors: PHP script name, function name, and line number of function calls. The classes represent different kinds of code injections and different levels of "stealthiness." Stealthiness refers to the adversary attempting to hide the injected code amongst normal execution, whether that be by using valid script names, function names, or line numbers in their injected code. We decided that our "simulated" attacker would have the proof of concept goal of creating and writing to a server side file as a Proof of Concept (POC). The code injection classes included:

1) Exploiting the `include` and `require` keywords. PHP scripts can use the built-in functions include and require to execute code from another script before executing their own, similar to the `import` statement in Python. Attackers with the ability to inject code or file paths can use these functions to cause a benign PHP script to call a malicious PHP script of their choice. This is called a Remote File Inclusion (RFI) attack. CVE 2021-24472 documented an RFI vulnerability in both a WordPress plugin and theme which would have allowed for this kind of code injection [22]. To simulate this class of code injection, we modified `index.php` to include a malicious script placed on the server.

2) Injecting a malicious PHP script. AFU vulnerabilities such as the one mentioned previously allow an attacker to inject a malicious PHP script to a server's home directory. By then visiting the associated URL, the attacker could cause the server to execute the code. In order to test this class, we uploaded our POC script and visited the newly created URL.

3) Replacing a valid script with a malicious script of the same name. Should an attacker attempt to replace a valid script with their own functionality, Warden should still detect the injection and log the function calls that the imposter script makes. To test this class, we replaced `index.php`'s functionality with our POC script instead of loading the site.

4) Injecting a function call into a valid script. An attacker may not remove any functionality from a valid script, but rather append to it in an attempt to hide their malicious function calls in amongst the normal execution of the script. To test this class, we injected code from our POC script into index.php in such a way as to allow the site to load normally.

5) Injecting code without custom functions into a valid script. Should an attacker have knowledge of Warden's presence, they may attempt to inject code into a script that does not make any function calls, so as to not alert Warden. This code could for example reassign the value of some local or global variables in order to change the execution of conditional statements. Warden may still catch this injection as it monitors execution at a line number granularity, meaning any valid function calls that were moved due to the injected code would be marked as suspicious. To test this class, we added variable declarations into `wp-login.php`, moving function calls in the rest of the script down a line.

6) Replacing a valid script with valid function names called on the right line numbers. Finally, assuming an attacker with complete knowledge of the server source code and how Warden monitors execution, they may write a malicious script with a valid name and define malicious functions with the same names as valid functions in that script. If they ensure all functions are only called on the line numbers they are expected, they should be able to fool Warden.

The results of our simulated code injections can be seen in Table 3.

Injection Class	Detected
1) include()	✓
1) require()	✓
2) Script Injection	✓
3) Script Replacement	✓
4) Function Injection	✓
5) Wrong Line Number	✓
6) Hidden Injection	**X**

Table 3: Effectiveness of Warden Detection

Warden was able to correctly identify malicious code injection in every case except for when the injected script used valid script names, function names, and line numbers, as was expected. However, in order to execute an attack of this class, an attacker would have to replace the functionality of a valid script on the server, which would likely be noticed and investigated quickly. This class could be made significantly harder to exploit if Warden was configured to also take parameters passed to function calls into account since the PHP logs include parameters. This would significantly increase the size of Warden's trusted set of function calls however, and further complicate obtaining a complete set of functionality. Future work could further restrict the attacker by tracking previously inspected function calls to ensure functions are being executed in the expected order in each script.

5.2.2 False Positives

We trained Warden using a combination of static and dynamic analysis, analyzing WordPress source code and simulating trusted traffic on the site. However, the task of exhaustively enumerating a web application's functionality is complex and difficult, as execution branches can change depending on a number of factors including browser, user authentication level, state of application caches, and external sites. Ideally, Warden would be trained on a complete test suite of the site's functionality. Depending on the application the developers may have such a test suite available, or other methods to improve dynamic analysis, such as those discussed in Section 3.0.2.

In order to test Warden's false positive rate when trained properly, we simulated traffic on a subsection of WordPress' functionality that we'd trained and tested on. We used Apache JMeter to constrain the end user's browser type and authentication level. We then recorded the number of logs seen, number of logs elevated to hashmap CFI, and number of false positives raised by Warden. Out of the 10,003 requests made, 32,386,535 logs were seen, 12 logs were elevated to hashmap CFI, and 0 false positives were raised.

As was expected, Warden's multi-layered CFI model allowed the vast majority of traffic ($> 99.99\%$) to be quickly approved by the lightweight bloom filter, only elevating $3.7 \times 10^{-7}\%$ of traffic to the slower and more computationally intensive hashmap CFI. From this we conclude that Warden, or another system trained using historic execution data would be able to perform with a very low false positive rate if trained on a complete enough training set. Any attempt to generate the training set through simulation would need to be diverse in its level of user authentication, method of contacting the site (browser, command line, Apache JMeter), and time of requests in order to account for the complex conditional execution of large web applications.

6 Conclusion and Future Work

In this work we introduce Warden, a multi-layered approach to PHP web application CFI that aims to leverage historical execution data to identify anomalous activity while quickly approving benign traffic. In our testing site (WordPress 5.1.1), Warden only incurred constant memory overhead ($\sim$ 200 MB) when compared to baseline SuS, regardless of the number of users. We show that a historically trained multi-layered approach to CFI is able to correctly identify anomalous traffic similar to common Remote Code Injection and Execution attacks without raising many false positives, given that it is trained on a large enough training set.

We found that acquiring a complete training set for complex web applications such as WordPress can be difficult due to the dynamic nature of web applications. Functionality surrounding caching/decaching, user-authentication functions, and user-driven execution (i.e. `eval` and `call_user_func`) further complicate collecting a complete set of function calls.

While Warden did add significant latency overhead to the SuS system, we found that the vast majority of this overhead was accumulated during the creation, extraction, and parsing of PHP function logs from the container's PHP runtime to the Warden manager process running on the host. This implies that Warden's CFI functionality may add very little overhead if implemented alongside a more efficient data extraction method, though this hypothesis remains to be tested.

We made many cursory optimizations throughout the development of Warden, decreasing overhead and simplifying systems. By simply altering the policy Warden used to wake up worker threads, CPU usage with 1 container dropped by 20%. Future work could certainly make similar advancements, implementing a more complex time-based approach to waking and sleeping worker threads, decreasing CPU utilization even further. As described above, performance in terms of both CPU usage and latency would likely improve drastically if a method was found to make the extraction and parsing of logs (both on the PHP extension and Warden side) more efficient and reliable. Complications in our testing environment prevented us from using threading in the PHP extension. This is likely a cause of a significant portion of the latency overhead and could certainly be investigated and fixed with future work.

Many web CFI technologies such as Warden, Saphire, and ZenIDS rely on building a trusted set of benign activity, whether it be in terms of PHP function calls, PHP opcode or system calls [12][18]. Future work may investigate the feasibility of exhaustively enumerating *or* simulating arbitrarily complex web applications' functionality. Both enumerating and simulating functionality would provide CFI systems with the trusted set they need through static or dynamic analysis respectively. Finally, future work could integrate Warden's multi-layered system into other web CFI implementations, swapping out the rough hashmap CFI described in this paper for a more advanced system such as those described in the Saphire and ZenIDS papers.

The sub-field of web application CFI is still relatively unexplored compared to its compiled counterpart. While Warden in its current state is not feasible for deployment to larger web servers, we conclude that the principles of using multi-layered CFI to improve performance and leveraging historical data to accommodate for the dynamic environment of the internet are promising tools that future works can utilize and improve on.

Appendices

A Test Statistics

Average CPU Usage (%)				
User Count	**Nginx**	**Baseline SuS**	**PHP Ext.**	**Warden**
1	14.10	21.71	42.00	53.14
10	66.11	76.47	99.47	97.93
25	87.84	94.40	99.79	98.93
50	88.42	98.61	99.85	99.51

Median CPU Usage (%)				
User Count	**Nginx**	**Baseline SuS**	**PHP Ext.**	**Warden**
1	12.97	22.12	41.58	52.74
10	66.88	76.10	99.52	98.05
25	88.50	94.75	99.78	99.03
50	88.59	98.80	99.80	99.56

CPU Usage Standard Deviation (%)				
User Count	**Nginx**	**Baseline SuS**	**PHP Ext.**	**Warden**
1	3.91	3.94	4.23	3.74
10	5.40	2.33	0.33	0.67
25	4.42	3.37	0.20	0.43
50	1.60	1.86	0.16	0.26

Average RAM Usage (MB)				
User Count	**Nginx**	**Baseline SuS**	**PHP Ext.**	**Warden**
1	1452	1985	2015	2155
10	1456	2172	2200	2343
25	1459	2475	2549	2697
50	1458	3020	3126	3282

Median RAM Usage (MB)				
User Count	**Nginx**	**Baseline SuS**	**PHP Ext.**	**Warden**
1	1450	1985	2016	2156
10	1457	2172	2201	2343
25	1458	2477	2549	2695
50	1459	3022	3128	3282

RAM Usage Standard Deviation (MB)				
User Count	**Nginx**	**Baseline SuS**	**PHP Ext.**	**Warden**
1	9.60	8.13	8.61	9.28
10	8.32	7.73	6.75	7.96
25	5.89	10.50	7.47	9.49
50	7.38	8.63	11.01	10.61

Average Latency (ms)				
User Count	**Nginx**	**Baseline SuS**	**PHP Ext.**	**Warden**
1	62.74	68.71	110.05	113.09
10	61.88	83.98	442.95	443.84
25	91.13	156.35	1038.03	1022.34
50	191.73	278.09	2012.7	2011.62

Median Latency (ms)				
User Count	**Nginx**	**Baseine SuS**	**PHP Ext.**	**Warden**
1	62.00	68.00	107.00	110.00
10	61.00	81.00	445.00	445.00
25	91.00	153.00	1049.00	1023.00
50	191.00	263.00	2004.00	1998.00

Latency Standard Deviation (ms)				
User Count	**Nginx**	**Baseline SuS**	**PHP Ext.**	**Warden**
1	4.15	5.56	13.95	15.54
10	7.73	17.40	127.49	129.16
25	10.04	40.22	232.39	228.07
50	12.48	99.00	377.65	384.37

References

[1] C. Bell, K. Gumienny, and N. Odell, "Developing single use server containers." https://web.wpi.edu/Pubs/E-project/Available/E-project-032620-210114/unrestricted/MQP_Containers.pdf, 2020.

[2] J. P. Larson, "Single-use servers: A generalized design for eliminating the confused deputy problem in networked services." https://digital.wpi.edu/downloads/vm40xv24x.

[3] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Comput. Surv.*, vol. 50, apr 2017.

[4] B. Braun, C. Gries, B. Petschkuhn, and J. Posegga, "Ghostrail: Ad hoc control-flow integrity for web applications," in *ICT Systems Security and Privacy Protection* (N. Cuppens-Boulahia, F. Cuppens, S. Jajodia, A. Abou El Kalam, and T. Sans, eds.), (Berlin, Heidelberg), pp. 264–277, Springer Berlin Heidelberg, 2014.

[5] B. Braun, P. Gemein, H. P. Reiser, and J. Posegga, "Control-flow integrity in web applications," in *Engineering Secure Software and Systems* (J. Jürjens, B. Livshits, and R. Scandariato, eds.), (Berlin, Heidelberg), pp. 1–16, Springer Berlin Heidelberg, 2013.

[6] B. Braun, C. v. Pollak, and J. Posegga, "A survey on control-flow integrity means in web application frameworks," in *Secure IT Systems* (H. Riis Nielson and D. Gollmann, eds.), (Berlin, Heidelberg), pp. 231–246, Springer Berlin Heidelberg, 2013.

[7] "Address space layout randomization." https://www.ibm.com/docs/en/zos/2.4.0?topic=overview-address-space-layout-randomization, Apr. 8 2021.

[8] H. Sidhpurwala, "Security technologies: Stack smashing protection (stackguard)." https://www.redhat.com/en/blog/security-technologies-stack-smashing-protection-stackguard, Aug. 20 2018.

[9] "Position independent executables (pie)." https://www.redhat.com/en/blog/position-independent-executables-pie#:~:text=Position%20Independent%20Executables%20(PIE), Nov. 28 2012.

[10] C. Zhang, M. Niknami, K. Z. Chen, C. Song, Z. Chen, and D. Song, "Jitscope: Protecting web users from control-flow hijacking attacks," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, pp. 567–575, 2015.

[11] I. M, M. Kaur, M. Raj, S. R, and H.-N. Lee, "Cross channel scripting and code injection attacks on web and cloud-based applications: A comprehensive review," *Sensors*, vol. 22, no. 5, 2022.

[12] A. Bulekov, R. Jahanshahi, and M. Egele, "Saphire: Sandboxing PHP applications with tailored system call allowlists," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 2881–2898, USENIX Association, Aug. 2021.

[13] M. A. Puentes, Y. Lei, N. Rakotondravony, L. T. Harrison, and C. A. Shue, "Visualizing web application execution logs to improve software security defect localization," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 1183–1190, 2022.

[14] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), p. 901–913, Association for Computing Machinery, 2015.

[15] C. Canella, S. Dorn, D. Gruss, and M. Schwarz, "Sfip: Coarse grained syscall-flow-integrity protection in modern systems." https://arxiv.org/abs/2202.13716, Feb 2022.

[16] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *2009 Ninth IEEE International Conference on Data Mining*, pp. 149–158, 2009.

[17] J. Bacon, D. Eyers, T. F. J.-M. Pasquier, J. Singh, I. Papagiannis, and P. Pietzuch, "Information flow control for secure cloud computing," *IEEE Transactions on Network and Service Management*, vol. 11, no. 1, pp. 76–89, 2014.

[18] B. Hawkins and B. Demsky, "Zenids: Introspective intrusion detection for php applications," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 232–243, 2017.

[19] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2012.

[20] SpaceHen, "Wordpress plugin download from files 1.48 - arbitrary file upload." https://www.exploit-db.com/exploits/50287, Sept. 13 2021.

[21] Kuroi'sh, "Wordpress plugin history collection 1.1.1 - arbitrary file download." https://www.exploit-db.com/exploits/37254, June 10 2015.

[22] "Cve-2021-24472." https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-24472, Jan. 14 2021.