Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists

Alexander Bulekov* Rasoul Jahanshahi* Manuel Egele
* *Equal contribution joint first authors*
Boston University
{alxndr,rasoulj,megele}@bu.edu

Abstract

Interpreted languages, such as PHP, power a host of platform-independent applications, including websites, instant messengers, video games, and development environments. With the flourishing popularity of these applications, attackers have honed in on finding and exploiting vulnerabilities in interpreted code. Generally, all parts of an interpreted application execute with uniform and superfluous privileges, increasing the potential damage from an exploit. This lack of privilege-separation is in stark violation of the principle of least privilege(PoLP).

Despite 1,980 web app remote code execution (RCE) vulnerabilities discovered in 2018 alone [25], current defenses rely on incomplete detection of vulnerable code, or extensive collections of benign inputs. Considering the limitations of bug-finding systems, the violation of the PoLP exposes systems to unnecessarily-high risks.

In this paper, we identify the current challenges with applying the PoLP to interpreted PHP applications, and propose a novel generic approach for automatically deriving system-call policies for individual interpreted programs. This effectively reduces the attack surface (i.e., set of system-calls) an exploit can leverage to the system-calls the script needs to perform its benign functionality.

We name our implementation of this approach, Saphire, and thoroughly evaluate the prototype with respect to its security and performance characteristics. Our evaluation on 21 known vulnerable web apps and plugins shows that Saphire successfully prevents RCE exploits, and is able to do so with negligible performance overhead (i.e., <2% in the worst case) for real-world web apps. Saphire performs its service without causing false positives over automatically and manually generated benign traffic to each web app.

Keywords: interpreted language, interpreter PHP, web application, system-call, remote code execution.

1 Introduction

Interpreted languages, such as PHP and JavaScript, are the foundation of modern-day computing. This is particularly true for the web, where online social networks, eCommerce, and online news attract the attention of billions of daily users. The ensuing swaths of personal, financial, and otherwise sensitive information held by these entities, make web sites attractive targets for cyber attacks. Beyond localized leaks of information, web apps and the interpreted languages that power them have also been at the core of data breaches that affect society at large. In 2015 attackers allegedly leveraged vulnerabilities in plugins of the WordPress and Drupal web apps to leak what has become known as the "Panama Papers" [36]. As testament to this crisis, Symantec reports [10] that in 2017, one in every 13 web requests was malicious. What exacerbates the situation is that, according to W3Techs [43], nine out of ten most popular web-development languages are interpreted. Furthermore, 2017 saw a 400% year-over-year increase [24] of reported vulnerabilities in the top four most popular content management systems. All four are interpreted web apps and have attracted significant attention from attackers.

Arbitrary code-executions(ACE) are the most dangerous class of application vulnerabilities, as they allow an attacker to take complete control over the running application. The root issue that makes ACE so hazardous is the fact that modern interpreted applications do not adhere to the principle of least privilege (PoLP) [46]. An attacker's exploit executes with ambient authority and is constrained only by the operating systems' access control mechanisms.

Some at-risk projects have recognized this problem and taken steps to intentionally reduce the run-time privileges of their software. By relinquishing access to unneeded system resources and API's, the software reduces the potential impact of a vulnerability. This practice has been widely adopted by native applications such as Chrome, Firefox, Tor, QEMU, and OpenSSH, but it is not in common use by interpreted applications. The reason for this is that, by design, interpreters introduce a layer of abstraction between the program and the underlying system which manages the system resources and APIs needed by the interpreted code. This leads to the status-quo, where all interpreted scripts share the same ubiquitous privilege with respect to the system-calls they can issue.

Existing defenses to detect and mitigate code-execution vulnerabilities have been built on static taint analysis [62], the analysis of code property graphs [5], or dynamic taint analysis [22]. While static analysis is promising, the dynamic language features (e.g., dynamic includes or class auto-loading in PHP described in Sec. 4) of interpreted languages can render static analysis impractical. Dynamic analyses can, if the induced performance overhead allows, be run in an always-on mode of deployment where interpreted programs are protected against ACE vulnerabilities at runtime. To this end, ZenIDS [22] is a dynamic taint analysis tool that builds an execution profile of the PHP interpreter while processing benign requests. Once these profiles are trained for a given web apps, ZenIDS is switched into enforcement mode, where it rejects requests that violate the learned profiles. Unfortunately, such dynamic systems require training for each application with a *representative* set of known benign requests. Obtaining such a set of requests remains a known-hard and open problem that affects the utility of any learning-based defense. During the evaluation, ZenIDS raised thousands of alerts for benign crawling traffic and requests to nonexistent scripts. Furthermore, our best attempts involving an ensemble of automated crawling, unit-testing and manual-crawling techniques achieved an average coverage of 33% during our evaluation over modern web apps. The difficulty of exercising web-applications is further evidenced by [4], where after a similar ensemble of techniques applied to a different set of applications, 53.2% lines belonged to functions that never executed. Comparing our line coverage with [4]'s lines after function-debloating (an upper bound for line-coverage), we achieved similar or higher coverage for apps shared across our evaluations: phpMyAdmin, WordPress, and Magento.

Instead of relying on representative benign behavior or statically searching for vulnerabilities, this paper introduces an abstraction-aware technique for applying the PoLP to interpreted PHP applications. We treat the capability to issue a system call as a privilege. Thus, the PoLP dictates that each PHP program should *only* be allowed to invoke the system calls that it needs to function correctly. System call sandboxing techniques have long been part of the defensive arsenal of security researchers and practitioners, and they are commonly applied by native applications, such as web-browsers (Chrome) and container-environments (Docker). However, the generic design of interpreters, such as PHP, requires that the sandbox be custom-tailored for each program the interpreter executes to provide meaningful security benefits.

As existing mechanisms such as SELinux and AppArmor cannot distinguish instances of interpreters that execute different scripts, these techniques are not applicable to solve the challenges of ACE vulnerabilities in interpreted languages (see §2.2). To improve this unsatisfactory situation, this paper presents a principled approach to retrofit interpreters and the programs they interpret to adhere to the PoLP.

As described above, the layer of abstraction introduced by the generic functionality of interpreters prevents the use of existing system call sandboxing techniques. Hence, our approach analyzes the programs that might be executed by the interpreters, while being cognizant of this additional abstraction layer, and devises individual so-called system call *profiles* (i.e., the set of system-calls a program is allowed to make). This approach consists of three essential steps can be applied to most interpreted languages. Step ① analyzes the API the interpreter exposes to the application identifies the set of system-calls each API function can trigger. Step ② identifies the API functions used by each interpreted program. Combined with the knowledge from the first step, this creates the system-call profile for each interpreted program. The final step ③, enforces the system-call profile whenever the interpreter executes the corresponding program.

Saphire does not perform anomaly-detection or explicitly prevent bugs, but severely limits the potential severity of their exploitation. While this PoLP-based approach can be applied for various interpreted languages, we instantiate it in Saphire, our prototype implementation that automatically retrofits web apps written in PHP with a custom-tailored system call allowlist for each script comprising the application. Throughout the execution of a script, the allowlist applies to the interpreter and any of its child-processes. Effectively, Saphire retrofits PHP web apps to adhere to the principle of least privilege and reduces the attack surface (i.e., the set of available system-calls) that are available to ACE exploits.

We evaluate our Saphire prototype on six PHP web apps and nine plugins that, in aggregate, contain 21 known ACE vulnerabilities. Saphire prevents all these attacks from succeeding. Additionally, throughout our false-positive evaluation, based on an ensemble of techniques, the recommended configuration of Saphire raises no false alerts. Since Saphire relies on built-in features of the kernel to enforce the system-call allowlists, it causes only minor overhead to request processing times.
In summary, we make the following contributions:

- We identify that effective system-call-level sandboxing of PHP programs requires an integrated analysis over *both* the PHP interpreter and the interpreter programs.

- We propose a novel 3-stage approach to identify and enforce system-call profiles for interpreted applications and describe viable implementations of each stage (§3).

- We present Saphire as our prototype implementation of this approach for PHP applications (§4). Saphire is implemented as a PHP extension and can be deployed without modifications to the PHP runtime or the web apps it protects. To the best of our knowledge, Saphire is the first PHP security system to reason about the entire execution process, including the interpreted code, the interpreter, and all of the native libraries it relies on.

- We evaluate Saphire thoroughly for its security and performance characteristics on six popular web-apps and nine related vulnerable plugins (§5). Saphire detects and prevents exploits against all the 21 previously known ACE vulnerabilities in our evaluation data set. We installed Saphire alongside both the Apache and nginx web-servers, and confirmed that it can block exploits for both. Moreover, Saphire protects vulnerable web apps without causing false positives during benign use of the web-apps, with a low, $< 2\%$ overhead in the worst-case.

In the spirit of open science, we will open source our entire implementation along with the testing and evaluation harness.

2 Background

In this section, we describe the threat of remote code execution, explain how system-calls play a key role in vulnerability exploitation, and discuss existing exploit mitigation techniques. Finally, we describe interpreted programs, and the fundamental challenges they create for existing mitigations. These factors motivate the design of our approach, which we use as a basis for our implementation – Saphire.

2.1 Remote Code Execution Vulnerabilities

Remote Code Execution (RCE) occurs when a network-attacker gains the ability to execute arbitrary code (ACE) on a target system. Since we implemented Saphire for PHP, and PHP is used mostly for remotely-accessible web-applications, the rest of the paper talks mostly about RCE attacks, but Saphire's defense does not depend on the trasmission-medium for the attack. RCE exploits against interpreted programs generally rely on improper usage of language features. Notably, RCE exploits commonly rely on: Code Injection (OWASP [60] A1) , Insecure Deserialization (OWASP A8) or , Unrestricted File Uploads.

Once the attacker exploits an RCE vulnerability, they leverage the exploited process to run a payload to, generally, gain access to additional resources. The operating system provides a system-call interface which processes must use to access privileged resources, such as network, file system, and process-management. Therefore, in order for the attack to be fruitful, the payload must invoke system-calls to access resources managed by the OS. For example, a simple payload may try to expose a shell which the attacker can connect to remotely. Such a payload requires, at minimum, access to the network and process-management, to spawn a shell process.

2.2 System Calls and Mitigation Techniques

An operating system manages the resources on a computer and provides user-space processes with mediated access to these resources via the system-call API. Programs and payload code alike can only communicate with the process' environment through the system-call API.

Recognizing that the system-call API is a key interface which is used by both benign and compromised processes, operating systems provide methods for limiting the system-calls accessible to an attacker who has exploited a process. For example, with Linux' `seccomp`, a process can provide the kernel with a filter, which the kernel uses to decide which system-calls to allow from the process, in the future. Once the filter is installed, it is enabled for the lifetime of the process, and it is not possible to remove restrictions. If a system-call is filtered, the kernel kills the process. `seccomp` is used for sandboxing major client and server software including, Chrome, Firefox, Tor, QEMU, and OpenSSH. The filtering is done by the kernel itself, so the overhead is negligible.

There is a wealth of both static and dynamic techniques to identify the system-calls a program relies on, for filtering purposes [15,18,20,28,31,32,42,55]. Some have even suggested generating system-call filters during a binary's build-time [17]. Past system-call filtering techniques focus on analyzing the system-calls performed by a binary program, but interpreters introduce a generic abstraction-layer between the system-call interface and the interpreted program.

Linux also supports security modules (LSM), such as AppArmor and SELinux, which add support for access control policies, including mandatory access control (MAC). MAC rule-sets can be used to explicitly limit the "capabilities" of a program, such as access to network or specific files. Security modules allow an administrator to manually secure a process, if its interactions with the OS are well-defined. Unfortunately, LSMs cannot distinguish between individual scripts executed by an interpreter. Hence, it is difficult to build a MAC rule-set for an interpreter while enforcing the PoLP for individual interpreted programs. As testament to the limited applicability of LSMs to interpreted programs, we followed an AppArmor-based hardening procedure suggested by Docker Inc, specifically for use with WordPress[1]. Unfortunately, even this tailored AppArmor ruleset does not prevent attacks against WordPress that exploit the [54] file-upload vulnerability. To improve the status-quo for securing interpreter processes at the OS-interface, our approach for system-call-filtering systems considers, *both* the interpreter and the interpreted program as a single principle.

2.3 Interpreters

Interpreted programs rely on a separate application - the interpreter, for execution. Separating the binary code in the interpreter from the actual program makes code portable and allows for straightforward implementation of advanced language features, such as reflection and dynamic-scoping. Essentially, the interpreter is a layer of abstraction, separating

[1]`https://github.com/docker/labs/tree/master/security/apparmor#step-5-custom-apparmor-profile`

the program code from the low-level details of the underlying operating system. Since interpreted programs still need access to system resources (e.g., files or network sockets), they must have a means of communicating with the kernel. To bridge the gap created by the abstraction, and provide programs with access to OS-managed resources, interpreters provide an API to the programs they execute.

For example, the PHP interpreter's built-in API provides access to the file-system, network, databases, as well as unprivileged resources such as built-in data structures and string-operations. When an API feature requires access to OS-managed resources, such as network sockets or file descriptors, the interpreter issues a system-call, which is handled by the kernel. Interpreted programs can define functions in their code, but unlike the interpreter API, these functions can never directly issue system-calls, as this would break the interpreter abstraction. Figure 1, shows how interpreted programs access resources guarded by system-calls through the built-in API.

Program dependencies are a prominent feature in many interpreters. Dependencies allow developers to organize and reuse code and encourage good software engineering practices. Dependencies impact the APIs and system-calls required by a program, since each dependency can contain its own API function-calls. In PHP, dependencies can be explicit (e.g., via `include()`), or implicit(e.g., through class auto-loading rules). Java supports implicit dependencies, by allowing the developer to specify a `ClassLoader` which dynamically resolves and loads undeclared classes. Explicit includes, can have dynamic arguments. For example, Python developers can dynamically load and execute modules using a variable path argument to `__import__()`. PHP supports both dynamic arguments to includes, and dynamic class-resolution (through the `spl_autoload_regiser()` interface).

Threat Model

Our threat-model assumes that an interpreted application running atop an uncompromised OS contains an ACE vulnerability for which the attacker has an exploit. The goal of this work is to enforce the PoLP on interpreted programs and hence restrict the capabilities (i.e., the set of available system-calls) the attacker's payload can use. Saphire is designed to restrict an attacker exploiting an ACE vulnerability in an interpreted program. As such this work does not focus on attacks which leverage a compromised interpreter to obtain arbirary-code-execution in a separate service (for example by triggering a buffer overflow in a database daemon over a network socket). As our evaluation (§5) shows, the vast majority of programs comprising real-world web apps can be confined such that existing exploits are mitigated.

3 Overview

In this section we describe our 3-stage approach for depriv-ileging interpreted programs using automatically-generated system-call allowlists. In Section 4 we detail Saphire – our prototype implementation of this approach for PHP web apps.

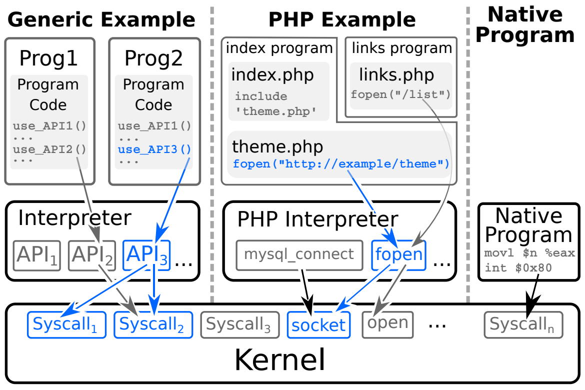

Figure 1: Left: an abstract interpreter executing Prog_2 invokes two system-calls to service a call to API function API_3. Center: a real PHP interpreter using real APIs and system-calls. Right, a program uses native code to invoke an system-call. API handlers within interpreters rely on similar native instructions. In blue, we trace an API call through the generic and PHP interpreters.

Our method of protecting interpreted applications involves collecting information about the system-calls invoked through the interpreter API, finding the interpreter API functions (e.g. `fopen` in Fig. 1) used by interpreted applications, and combining the results to enforce a tailored system-call allowlist.

To explain the process in more detail, we first describe the interpreters and programs to which our approach applies. We then explain why generating meaningful system-call allowlists requires consideration of *both* the interpreter and each interpreted program. Finally, we describe the purpose of each of the three stages, and explain how their functionality can be combined to secure programs.

3.1 Interpreters

We define *interpreted programs* as programs which require an ancillary application (i.e., an interpreter) to execute on a computer. The interpreter is, generally, an application native to the computer system – i.e., it can be directly executed within an operating system, by the hardware. Hence, interpreted applications can be portable across systems for which compatible interpreters exist. In addition to parsing and executing programs, interpreters expose an API, which allows programs to rely on the interpreter for built-in functionality. The API is composed of *functions*, which can be invoked by the interpreted program, and each of which can be implemented natively as an interpreter API *handler*. In Figure 1, $\texttt{API}_{1,2,3}$, `mysql_connect`, and `fopen` are API functions. The interpreter forms an abstraction layer between the program and the system, with a natively-implemented API bridging the gap. Thus, we make a key observation about interpreters and interpreted applications compatible with our approach: *Only the interpreter's code issues system-calls, commonly in re-*

sponse to an interpreted program invoking the API.[2] Note that some interpreters implement just-in-time(JIT) compilation, translating the interpreted program into native machine code at runtime. For interpreters with JIT-support, such as Java Hotspot and .NET CLR, the translated code still calls into a native API to invoke syscalls, so in the context of Saphire, this optimization is simply an interpreter implementation detail.

3.1.1 An API for all interpreted programs

The functions in the interpreter API must be generic so that they are useful to the wide range of interpreted programs. As a result, the interpreter provides API functions that collectively invoke a diverse set of system-calls. Therefore, we cannot create a meaningful system-call filter by simply enumerating the system-calls invoked anywhere in the interpreter.

Fortunately, individual interpreted programs depend on a small subset of all API functions provided by the interpreter, and in extension only require a small set of system-calls to execute correctly. For example, the generic `Prog1` in Figure 1 does not rely on $\texttt{API}_3$ and hence does not need $\texttt{Syscall}_1$.

Thus, during the execution of `Prog1`, it is safe to filter access to $\texttt{Syscall}_1$, even though it occurs within the interpreter binary. To enforce the PoLP, we must analyze the *joint behavior* of the interpreter and the program. Based on these insights we present a three-stage process for creating tailored system-call filters for interpreted programs.

3.2 Securing Interpreted Programs

Stage ① maps the API exposed by the interpreter to a set of system-calls invoked by each API function. In stage ②, the interpreted program is analyzed to identify the APIs it invokes. Composing this information with the map from stage ①, the output of stage ② is the list of system-calls required by the interpreted program (i.e., the system-call filter). In the final stage, the program is executed, and the system-call filter is applied to the interpreter process, protecting the program.

3.2.1 Mapping the interpreter API to syscalls

The goal of stage ① is to identify the system-calls invoked by each API function. As mentioned in Section 3.1, interpreters provide programs with access to a generic API, parts of which perform system-calls to expose system-managed resources. Generally, interpreter APIs which depend on system-calls are implemented natively, conforming to the OS-specific system-call interface (see Fig. 1).

Both static and dynamic analysis techniques can be used to map API functions to system-calls. For example, APIs can be mapped to system-calls through a static control-flow analysis of the interpreter. The analysis involves labeling the API function handlers as sources, the system-call invocations as sinks, and calculating the reachability between the two sets in the interpreter's call-graph. A dynamic analysis can be used to refine this statically-obtained mapping.

The result of stage ① is a mapping of interpreter API functions to required system-calls. This mapping is generated once, for each version of the interpreter.

3.2.2 Identifying API calls within an interpreted program

In stage ② we identify the API functions invoked by an interpreted program. Incorporating the mapping from stage ①, this stage determines the system-calls needed by the program. We define a program as the body of interpreted code that can be executed by an interpreter process from an entry-point. For example, in the PHP example in Fig. 1, the index program includes the code defined both in `index.php` and in the included `theme.php`. Note that a single script can be included in multiple places, and therefore belong to multiple programs.

There are two steps to identify API calls by a program:

1. Identify *all* the code comprising the interpreted program.
2. Analyze the program's code to determine the interpreter API calls it can perform.

Identifying the program's code requires a consideration of the interpreted language features which create code-dependencies. In addition to "includes", dependencies can arise from implicit sources, such as customizable auto-loading rules. Once the dependency analysis is complete, we scan the code in the program for API function calls (step 2).

As in stage ①, both static and dynamic techniques can be applied to the program. The result of stage ② is a set of API functions referenced by the interpreted program, which composed with the mapping of API calls to system-calls produces the final mapping of programs to system-calls, which is used as a allowlist in the final stage.

3.2.3 Protecting the Program

In stage ③, to protect the program, the interpreter (or program) is modified to load the corresponding allowlist, prior to execution. This dynamic protection can be facilitated by built-in low-overhead support for filtering system-calls, which is present in operating systems such as Linux, FreeBSD and Windows. The implementation of the protection depends on the execution model of the interpreter. For example, the way protections are applied may differ for programs invoked on the command-line and ones executed by a web-server. In Section 4 we describe our implementation of system-call filtering of the PHP interpreter, on Linux. Our filtering mechanism works with both the Apache and nginx webservers, as well as the standalone php-cli interface.

[2] Some interpreters provide the means for programs to execute native code within the interpreter's process(e.g., JNI for Java). Such native additions can be treated as extensions to the interpreter's API. None of the applications in our evaluation rely on this feature, so we do not implement it in our prototype.

Applying the Model to Real Interpreters

The model of interpreted languages and the three stage allowlisting process described in this section is applicable to a variety of interpreted languages. The major Lua, Perl, Python and PHP interpreters all rely on native API handlers in an interpreter binary. Additionally, runtimes which operate over an intermediate representation/bytecode, such as Java JRE, Mono, ActionScript, or Dalvik all rely on native code for APIs which is contained in a separate interpreter, or linked into individual program binaries. As such, the steps we described can be applied to a wide range of interpreted languages.

4 Implementation

We implemented the three steps outlined in the previous section for the PHP language and interpreter in our prototype – Saphire. We choose PHP due to its dominance among web apps, which are major targets of RCE attacks, and because it represents an interpreted language with advanced dynamic features. PHP is dynamically typed, with dynamic binding of function and class names, dynamic name resolution, dynamic symbol inspection, reflection, and dynamic code evaluation support. We explain how Saphire combines static and dynamic analysis techniques in stage ①. We describe the static web app analysis performed in stage ②. Finally, we detail how Saphire uses `seccomp` to sandbox the PHP interpreter on a live web app in stage ③. Figure 2 details Saphire's implementation of the three stages introduced in Section 3.

4.1 Mapping built-in PHP functions to system-calls

PHP refers to API functions as built-in PHP functions. Hence Saphire's stage ① maps built-in PHP functions to system-calls. To this end, Saphire generates an initial mapping, by performing a static call-graph analysis over the PHP interpreter. To refine the statically-collected mapping, we use Linux `ptrace` , which allows us to inspect the system-calls invoked by a running PHP process. Note that `ptrace` is only used, offline, for ① and is not used for any active defense. Moreover, Saphire blocks `ptrace` for all scripts in the web apps we evaluated, by default, as we found no built-in PHP functions that invoke the `ptrace` system-call.

4.1.1 Static analysis over the PHP Interpreter

The PHP 7.1 build we use in the evaluation relies on 55 pre-compiled, dynamic shared libraries. Since PHP generally invokes system-calls through libraries (e.g. libpthread and libc), Saphire builds a static call-graph over the interpreter and all included libraries. Note that the debugging symbols for the interpreter and the 55 libraries are readily available in the Debian repositories. Saphire uses these symbols to facilitate the analysis in stage ① and the production-binary used in stage ③ is stripped. Using the symbols, Saphire builds a call-graph, where each node is a function and each edge is a direct function call. Saphire annotes the function nodes with the system-calls they invoke. To identify the nodes corresponding to the built-in PHP function handlers, we augment `get_defined_functions()` (which lists currently defined functions) to output the address of the handler for each built-in PHP function. Similar techniques are applicable to any interpreter with a symbol table, such as Python (where functions can be enumerated with `dir()` and `global()`). Saphire performs a reachability analysis over the call-graph, where the built-in PHP function handler nodes are the sources, and nodes annoted with system-calls are sinks.

Saphire's rudimentary call-graph analysis is purpose-built to cover libraries and identify system-calls. While this step can be implemented as a static source-code analysis and might yield a more precise call-graph, the analysis needs to operate over dozens of code-bases(for the interpreter, and 55 libraries) using different languages and build-processes.

4.1.2 Refining the mapping through dynamic analysis

The reachability analysis performs an exhaustive search over the code within a PHP process, but does not handle indirect calls, which can occur in built-in PHP functions. For example, PHP's `fopen()`, can access remote files over HTTP (see Figure 1). Based on the URI, fopen sets a function pointer which specifies whether to use an encrypted HTTPS, or unencrypted connection handler. The static call-graph does not contain edges to either of these functions, which leads to an incomplete mapping of built-in PHP functions to system-calls. Furthermore, some built-in PHP functions execute external programs. `mail()` executes the `sendmail` binary. In order to apply to PoLP to the `mail`, the mapping of system-calls should contain the system-calls performed by sendmail. The static analysis over $\boxed{\texttt{CG}}$ does not reason about the system-calls that occur in external processes.

To address these issues, we extend the statically-built profile, by tracing the system-calls performed by the PHP interpreter, while executing its test-suite. The test-suite is packaged with PHP's source. We rely on a PHP extension $\boxed{\texttt{TE}}$, which exposes the name of the currently running built-in PHP function through shared memory. A companion tracer, $\boxed{\texttt{TR}}$ uses Linux `ptrace` functionality to intercept system-calls. While the interpreter is executing the test-suite, $\boxed{\texttt{TR}}$ intercepts each system-call, and examines the current PHP function, exposed by $\boxed{\texttt{TE}}$. This allows Saphire to easily detect whether the currently running built-in PHP function relies on any system-calls missing from the statically-generated mapping. $\boxed{\texttt{TE}}$ also traces system-calls in external programs called by PHP, to account for built-in PHP functions, such as `mail()` which rely on external programs. The test-suite achieves a 73.4% line coverage over the PHP interpreter, allowing Saphire to discover additional system-calls used by 137 out of 4,655 built-in PHP functions.

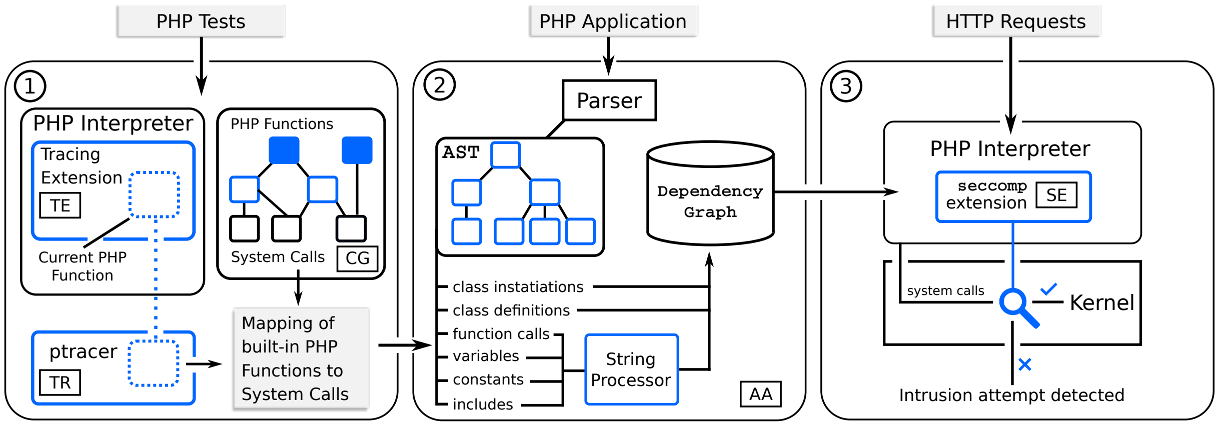

Figure 2: Saphire builds a mapping of built-in PHP functions to system-calls, acquires a list of built-in built-in PHP function calls in application scripts, and uses this information to protect the web app using `seccomp` system-call filters.

4.2 Creating system-call filters for web apps

In Stage ②, Saphire identifies each script's dependencies and determines the built-in PHP functions the interpreter can invoke while running the script. Composing this information with the mapping from ①, Stage ② outputs a set of possible system-calls invoked for each script in the web app.

To achieve this outcome, we built AA to perform a lightweight, flow-insensitive analysis, as a limited form of constant folding over strings that compose includes. AA iterates over all of the PHP files in web apps. We use php-parser [50] to parse each PHP script into its abstract syntax tree (AST). AA scans the AST for function or method calls to identify possible built-in PHP function calls. If a function call's name matches a built-in PHP function, AA infers that the script contains a call to the built-in. In the case of method calls, AA looks for all assignments of the object within the current scope to identify the class type, and checks whether the type and method combination corresponds to a built-in PHP function. To infer script-dependencies AA identifies AST nodes representing: (1) constant definitions, which frequently occur within include paths (2) class definitions/instantiations, which are essential for creating edges for auto-loaded classes, and (3) includes via the `include/require` operations. AA also identifies strings in all variable assignments, as these variables are often referenced in include statements. For each include, AA assembles an internal representation for each of these nodes, optimized for static and string content.

4.2.1 String representation

PHP strings can be composed of literals, and references to constants, variables, and function return values. When AA locates a node representing such a component, it notes its location. Once AA finds all nodes which compose strings, it iterates over the includes in a script. Saphire handles arguments to an includes, differently, depending on the node's type:

Literals: For literal strings, nothing needs to be done.

Constant Reference: Since Saphire keeps a record of all constants in the web app, it replaces the reference with the nodes the constant was defined with, and recurses over them.

Magic Constants: The interpreter automatically defines special constants, such as `__DIR__` and `__FILE__`, which describe the location of the current script. AA derives the script locations and filenames from the file-system hierarchy and translates them to literal values.

Variables: If the node is a reference to a variable, AA checks whether the variable is defined in the current scope. As with constants, AA replaces the reference with the nodes used in the assignment. If the variable was assigned multiple times, AA explores each possibility.

Function Calls: If the function is a known common API, such as `dirname`, `realpath`, or `strtoupper`, AA reproduces the functionality over the argument. Otherwise AA marks the result as unknown.

For each include, AA applies this procedure, recursively, until the include is composed of only literals and unknowns, and the PHP string concatenation operators (`.` and `.=`). Then AA translates the sequence of nodes into a regular expression, substituting the unknowns with regex wildcards (`.*`). Figure 3 demonstrates how AA handles includes built with multiple components. If the include refers to variables with multiple assignments, AA joins the regular expressions for each possibility with the "|" operator. AA handles relative path elements, such as `../`, by removing the preceding portions of the expression. If the immediately preceding expression is dynamic (i.e. `.*`), AA replaces all content before the relative path element with a wildcard. Once each include is represented as a regular expression, AA resolves includes by evaluating the regular expression against the paths of the PHP scripts in the web app. For each match, AA stores an edge in a dependency graph, where the nodes are PHP scripts.

Saphire handles auto-loaded classes in scripts by checking if a class with a matching name is declared in the resolved set of dependencies. If not, Saphire searches for matching class declarations in the rest of the web app and creates dependency edges to the corresponding scripts.

4.2.2 Unresolved Includes

In practice, $\boxed{\texttt{AA}}$ resolves 74% of includes to a single file, statically. Additional includes can be "fuzzy-resolved" – i.e., resolved to a subset of the files in the web app, such as all files in a subdirectory. Some include statements do not contain any information amenable to static analysis. In these cases, Saphire cannot determine a subset of PHP scripts which can satisfy an include statement. To address this, Saphire provides an option (Conservative Includes or `CI`) to resolve such includes to all scripts in the application. Enabling `CI` decreases the probability of false-positives due to missing edges in the dependency graph, but increases the number of allowlisted system-calls in scripts containing unresolved includes. We examine the effects of this option on false-positives and false-negatives in Section 5.

4.2.3 Building system-call profiles for Scripts

After identifying the built-in PHP function calls in the script files and building the dependency graph, $\boxed{\texttt{AA}}$ calculates the transitive closure of dependencies for each script, to obtain the list of built-in PHP functions called by the script or any of its dependencies. $\boxed{\texttt{AA}}$ builds the system-call profiles by replacing each of the built-in PHP functions in the list with the set of corresponding system-calls obtained in Stage ①. The output of $\boxed{\texttt{AA}}$, and Stage ② is a system-call profile (i.e., a allowlist) for each script, representing the system-calls for the built-in PHP functions used within the script, and all its dependencies. $\boxed{\texttt{AA}}$ marks each script path with its profile. The paths are relative to the root of the web app, so the output of ② is independent of the server and location of the application on the filesystem.

4.3 Sandboxing the Interpreter and Web App

The goal of stage ③(Sec. 3.2.3), is to sandbox an interpreted program when it executes. Our implementation, Saphire applies the allowlists from Stage ② to a live web app using

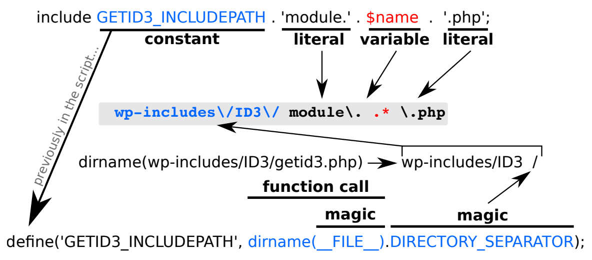
```
include GETID3_INCLUDEPATH . 'module.' . $name . '.php';
wp-includes\/ID3\/ module\. .* \.php
dirname(wp-includes/ID3/getid3.php) → wp-includes/ID3 /
define('GETID3_INCLUDEPATH', dirname(__FILE__).DIRECTORY_SEPARATOR);
```

Figure 3: Saphire inspects a WordPress include. The include references a constant defined in the script. Saphire reasons about the `dirname()` built-in API function, so it resolves the value of the constant. The variable `$name` is an argument to the function where the include occurs – Saphire cannot the possible contents, so it translates it to regex as a wildcard `.*`

`seccomp`. Specifically, Saphire deprivileges the PHP interpreter process, before it executes a web app scripts. Internally, Saphire relies on a PHP extension (labeled $\boxed{\texttt{SE}}$ in Figure 2) that invokes Linux' `seccomp` facility.

To use `seccomp`, a process provides the kernel with a filter to enforce over future system-calls made by the process. Upon startup, the PHP interpreter loads the $\boxed{\texttt{SE}}$ extension into the process. $\boxed{\texttt{SE}}$ determines which script the interpreter is about to execute, and provides the kernel with a system-call allowlist – a set of allowed system-calls. After providing the kernel with the filter, $\boxed{\texttt{SE}}$'s task is complete, since the kernel is responsible for enforcing the `seccomp` allowlist.

$\boxed{\texttt{SE}}$ is activated twice during the lifetime of the interpreter. When the PHP process is starting, it loads the $\boxed{\texttt{SE}}$ extension. $\boxed{\texttt{SE}}$ uses this opportunity to load the system-call allowlists from the disk into memory. Once the interpreter receives a request, it hands control to $\boxed{\texttt{SE}}$. $\boxed{\texttt{SE}}$ loads the allowlist for the requested script from memory, and provides it to the kernel, as a filter program. Internally, $\boxed{\texttt{SE}}$ uses `libseccomp`'s bindings to convert a set of system-calls into a allowlist [38]. PHP usually accepts web requests from a separate program - the webserver. Web-servers such as `nginx` and `Apache` implement advanced features such as reverse proxying, static resource caching, and load-balancing. When a Web-server receives a request that must be handled dynamically, it communicates with a PHP interpreter using an API, such as FastCGI. With a common `nginx` web-server using FastCGI to invoke PHP, the extension and allowlist are only loaded once, by a master process which forks workers to process requests. In our evaluation we installed Saphire's $\boxed{\texttt{SE}}$ plugin for a PHP interpreter accessible behind both major web-servers on Linux: nginx and Apache. We also deployed the same plugin for PHP's cli API, which allows executing PHP scripts from the command line (similar to Python or Perl). We did not evaluate this configuration, as the vast majority of PHP apps (and exploit targets) are web apps.

If a PHP script does not trigger `seccomp` violations, the interpreter process terminates once script execution concludes. Usually, the process cannot be reused to process other scripts, since different scripts have different system-call privileges, and `seccomp` does not allow a process to replace its system-call filters. This is a problem for interpreters that handle many short requests, since APIs such as `php-fpm` reuse the interpreter for many requests. There are two options to deal with this: (1) Configure the PHP API to only use a PHP interpreter process for a single request. While functional, this approach results in request latency, when the server is under high load. (2) Allow PHP workers to handle many requests, but ensure that each worker only handles requests for the same script. The worker loads a `seccomp` profile for the first request it receives, and this allowlist applies to all subsequent requests to the same script. For an application with many scripts, such as a CMS, dedicated workers handle scripts in high demand, and general workers handle the uncommon requests (restarting after each one). We evaluate both of these options in section 5.4.4. This issue is specific to interpreters that handle many short-lived requests, such as PHP. For longer lived executions, the one-time overhead of applying the system-call profile is negligible, but if reusing the interpreter is beneficial, routing requests to minimize Saphire overhead is a generic and effective (see Sec. 5.4.4) solution.

5 Evaluation

We evaluate Saphire's ability to mitigate remote code execution attacks on a set of popular PHP web apps and plugins. Additionally, we assess Saphire's stages, individually. Specifically, we examine the capabilities of Saphire's include-resolution, the reduction of system-call privileges due to the analysis in stage ②, and the performance of stage ③. Our experiments provide answers to three research questions:

RQ1 How precise is Saphire's dependency resolution (§5.2)?

RQ2 For each PHP script in a web app, what is the reduction in privilege/available system-calls with Saphire. How does the setting for `CI` affect the reduction (§5.3)?

RQ3 Does the retrofitted PoLP protect from known exploits, without causing false positives? How does the setting for `CI` impact the accuracy of the system (§5.4)?

5.1 Web Apps and Plugins in our dataset

We evaluate Saphire on six of the most popular PHP web apps. Our set includes the four most popular open-source content management systems: Wordpress, Joomla, Drupal and Magento. According to W3Techs, these systems comprise 70.5% of the market share among CMS systems, and 38.4% of the market for all websites [44]. Additionally, we include one of the most popular administration tools: phpMyAdmin [26], and Moodle, a popular course-management system.

In practice, administrators customize CMS deployments by installing plugins. To reflect this, we install nine vulnerable WordPress plugins: NMedia contact form, Wysija newsletter, Foxy Press, Photo Gallery, WP-Property, Reflex Gallery, Slideshow gallery, WP Symposium, WPtouch. As we are most interested in Saphire's capability to mitigate RCE attacks, we selected plugins and web app versions with the most-recently published RCE vulnerabilities and readily available proof-of-concept exploits. Additionally, to evaluate false-positives for plugins, we installed 9 of the most popular freely-available plugins. In total, our evaluation was conducted over 12 vulnerable versions of web apps, 9 vulnerable and 9 popular freely-available plugins.

5.2 Dependency Resolution (*RQ1*)

In stage ②, Saphire scans a PHP web app to determine the built-in PHP functions which might be invoked within each script. The accuracy of the system-call profile depends on the results of this stage. One of the main challenges for Saphire is resolving the dependencies between scripts. To address this challenge, Saphire performs include and class resolution to discover the dependencies.

Table 1 presents the include resolution statistics for the web apps in our dataset, collected after Saphire's static analysis. The *literal* column shows the number of include statements with a string literal argument. The *dynamic* column shows the number of include statements with arguments that are *not* string literals. The *resolved, fuzzy-resolved and unresolved* provide a breakdown of how the static analysis in ② resolved these includes. Namely, the *resolved* column contains the number of includes resolved to a single script within the web app. The *fuzzy-resolved* column specifies the number of includes that are resolved to a subset of all web app scripts. That is, the regular expression generated by $\boxed{\texttt{SE}}$ matched to multiple scripts. On average Saphire resolves 74% includes and fuzzy-resolves 22%.

In the same table, we show Saphire's class resolution statistics. On average, 85% of classes are resolved. Unresolved classes can occur when web apps define classes dynamically. For example, for historical reasons, Joomla dynamically creates an alias for each defined class by prefixing the class name with "J" (e.g., the original `Http` class will trigger the creation of `JHttp` as an alias)[3]. As the Joomla code-base uses both notations interchangeably, Saphire detects dependencies only if the original notation is used and does not detect dependencies if classes are referred to via their alias. While this behavior could be easily emulated in the analysis by duplicating the alias-generating logic in Saphire, we chose to elide any program-dependent modifications to the system. The effect of unresolved classes is the possibility of false-positives due missing edges in the dependency graph, if the application relies on an autoloader. As we will see, the only false positives we encountered during our evaluation are caused by the Joomla idiosyncrasy described above.

5.3 System-Call Profile Size (*RQ2*)

The security benefits provided by Saphire hinge on its ability to restrict access to system-calls – specifically those that are likely to be used by attackers. In this section, we examine the reduction of attack-surface, in terms of the number of system-calls in the allowlists. For qualitative measure, Section 5.4.2 further examines the dangerous system-calls (as defined by Bernaschi et al. [8]) that exploits can still use. We collected the data presented here by running stages ① and ② on a system running Linux Kernel 4.17 which provides 333 system-calls. Figure 4 shows the number of system-calls allowed for each script in WordPress 4.6, phpMyAdmin 4.8.1, Joomla 3.7, and Drupal 7.58. The colored regions represent profile sizes with the CI option *enabled*. The bottom-most, *Available Dangerous*, region represents the dangerous system-

[3]This is implemented in Joomla's class auto-loader. If the script instantiates a class with the name `JHttp` but the auto-loader cannot find it, the loader trims the "J" prefix and looks for a class with the name `Http` instead.

Application	Includes						Classes		
	Total	Literal	Dynamic	Resolved	Fuzzy-resolved	Unresolved	Total	Resolved	Unresolved
Drupal 7.0	263	9	254	175	57	31 (11.7%)	40	30	10 (25%)
Drupal 7.5	265	9	256	174	60	31 (11.6%)	48	39	9 (18.75%)
Drupal 7.26	214	1	213	171	42	1 (0.5%)	44	34	10 (22%)
Drupal 7.57	217	1	216	172	43	2 (0.9%)	24	28	6 (25%)
Drupal 7.58	218	1	217	173	43	2 (0.9%)	35	29	6 (17.1%)
Joomla 2.5.25	348	2	346	179	149	20 (5.7%)	252	229	23 (9.1%)
Joomla 3.7	265	5	260	102	152	11 (4.2%)	481	441	40 (8.3%)
Magento	1,190	271	918	971	175	42 (3.5 %)	3,339	3,120	219 (6.6%)
Moodle	7,548	877	6,671	5,605	1,876	67 (0.9%)	2,241	2,149	92 (4.1%)
phpMyAdmin 3.3.10	753	677	76	704	33	16 (2.1%)	49	48	1 (2.0%)
phpMyAdmin 4.8.1	292	222	70	254	32	6 (2.1%)	438	402	36 (8.2%)
WordPress	1,892	517	1,375	1,747	109	36 (1.90%)	215	193	22 (10.2%)

Table 1: Dependency Resolution statistics. We break-down the static and dynamic includes for each web app and the number of include Saphire resolves precisely, and approximately. We also present similar data for class references.

calls available to each script. The *Available* region represents additional system-calls available to each script, which are not considered dangerous. Hence, the allowlist for a given file consists of the system-calls contained in these two regions. The upper-two regions represent blocklisted system-calls and dangerous system-calls, respectively. The black line represents the system-call profile sizes when the CI option is *disabled* (no dependency edges for unresolved includes).

As the graphs illustrate, Saphire generates system-call allowlists that significantly reduce the attack surface. The overall reduction of the attack surface in the number of system-calls is 80.5% on average, with the most permissive profile (i.e., the left-most script in Joomla) still removing 72% of system-calls from the allowlist. More important than the bare number of system-calls, Saphire reduces the number of available *dangerous* system calls also by 80% on average.

We note that "shelves" of system-calls occur in most of the graphs, indicating that many files require the same number of system-calls. This phenomenon is due to the fact that sets of scripts share the same dependencies. For example, Saphire finds that WordPress' `wp-includes/option.php` is included in 383(28%) of scripts. This leads to many files sharing similar system-call profiles.

When `CI` is enabled, scripts with unresolved includes include all other scripts in the web app. This results in "shelves" at the maximum profile-size, indicating that the scripts can invoke any system-call used in the entire web app. This reduces the possibility of false-positives due to missing dependency edges, but increases potential attack surface.

5.4 Defense Capabilities (*RQ3*)

We evaluate Saphire's protection against 21 remote code execution exploits. To this end, we created 12 Docker containers running vulnerable versions of the web apps in the evaluation dataset. As mentioned in Section 5.1, our WordPress installation contains 9 vulnerable plugins, for a total of 11 WordPress vulnerabilities. We attack the web apps using exploits from the Metasploit Framework [37], and consider an attack as successful if it exposes a shell to the attacker via the network. Of course, we first verified that all exploits work against unprotected versions of the web apps and plugins. In Table 2, we present the results of our experiments. Specifically, we evaluate the defense capabilities of Saphire when `CI` is enabled, and disabled.

5.4.1 Is Saphire too restrictive?

To properly apply the PoLP, Saphire should not prevent normal operation of the web apps. A false-positive for Saphire is a system-call blocked during benign execution of application code. Saphire *does not* rely on any benign web app traces to build the allowlists, but we exercise the web apps evaluate how prone Saphire is to false positives through an ensemble

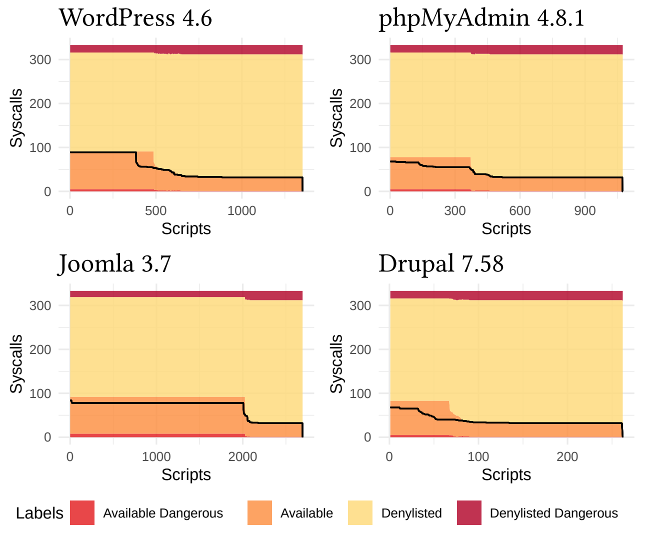

Figure 4: The sizes of the system-call allowlists for web apps in our test-set. The shaded areas represent allowlist sizes, when `CI` is *enabled* The black line shows the allowlist size when `CI` is *disabled*

		Exploits Blocked		False Positives		Benign Traces	Dangerous System Calls Available to Exploits
Application	Vulnerability	`CI` off	`CI` on	`CI` off	`CI` on	Line Coverage	
Drupal 7.0	CVE-2014-3704	y	y	0	0	†39.22%	openat, unlink
Drupal 7.5	drupal_restws_exec	y	y	0	0	†28.50%	chmod, openat, rename, symlink, unlink
Drupal 7.26	CVE-2014-3453	y	y	0	0	†37.12%	chmod, openat, rename, symlink, unlink
Drupal 7.57	CVE-2018-7600	y	y	0	0	†42.51%	chmod, openat, rename, symlink, unlink
Drupal 7.58	CVE-2018-7602	y	y	0	0	†43.63%	openat, unlink
Joomla 2.5.25	CVE-2014-7228	y	y	2	0	12.67%	chmod, openat, rename, unlink
Joomla 3.7	CVE-2017-8917	y	y	1	0	†27.95%	chmod, openat, rename, unlink
Magento 2.0.5	CVE-2016-4010	y	y	0	0	†40.61%	
Moodle 3.4	CVE-2013-3630	y	y	0	0	†28.00%	chmod
phpMyAdmin 3.3.10	CVE-2011-4107	y	y	0	0	13.72%	chmod, openat, rename, symlink, unlink
phpMyAdmin 4.8.1	CVE-2018-12613	y	y	0	0	†49.28%	chmod, openat, rename, unlink
Wordpress 4.6	11 Vulnerabilities	y	y	0	0	$^{\dagger *}$36.18%	

$*$ **Wordpress & Plugins Vulnerabilities** The WordPress vulnerabilities: WPVDB-7896, WPVDB-6680, WPVDB-6231, CVE-2014-9312, WPVDB-6225, CVE-2015-4133, CVE-2014-5460, CVE-2016-10033, WPVDB-7716, WPVDB-7118 wp_admin_shell_upload. Coverage including the vulnerable plugins is 17.88%. See Sec. 5.4.3 for evaluation over popular plugins.
Complete list of dangerous system-calls: *chmod, fchmod chown, fchown, lchown, execve, mount, rename, open(at), link, symlink, unlink, setuid, setresuid, setfsuid, setreuid, setgroups, setgid, setfsgid, setresgid, setregid, create_module*

Table 2: Exploits blocked for each configuration of Saphire. Coverage annotated with † was collected with the aid of unit-tests available for the web app.

of three complementary techniques:

- We replay browsing traces collected while one of the authors explored the web-app as a user and administrator. The traces exercise functionality available to privileged and unprivileged users.
- We crawl the application with a web crawler included in the Burp Suite. The crawler is authenticated and has access to privileged web app functionality.
- When available, we execute test-suites packaged with the web-apps. The test-suites are collections of PHP scripts, which exercise portions of the web app code.

We measure the combined line coverage of the three methods using the XDebug PHP debugger [45] and present the coverage as a percentage of total lines of PHP code (as determined by sloccount [59]) in Table 2. Our measurement accounts for possible coverage overlap between the three techniques and registers each covered line only once. However, sloccount greedily counts source lines that are not considered executable, and are consequently not tracked by XDebug. Hence, the average coverage of 33% is a strict lower bound of the true coverage that our mechanisms achieve. We selected Azad et al.'s work as, to the best of our knowledge, it is the most recent work to collect comprehensive coverage data over PHP web apps [4]. Unlike Saphire, Azad et al. collect coverage during an exploration stage to prune unused functions, thus cutting back on a web app's attack-surface. Azad et al. presents coverage as the percentage of lines in functions with any lines covered during the exploration stage. I.e. for any partially-covered functions, no lines are pruned. To mirror this technique, we calculated the percentage of lines contained in functions with *any* lines executed. According to this metric, Saphire covers 64% of WordPress, while [4] covers 57%. Additionally, we obtained the Selenium traces collected by Azad et al. and executed them on our instance of WordPress. These Selenium traces increased our coverage by 1%, without raising any false-positives. In summary, we found that our coverage is in-line with state-of-the art PHP research. Moreover, the difficulty in obtaining high dynamic coverage over web apps highlights the utility of using a static analysis for Saphire's implementation of ②, which can generate profiles even for uncovered code.

The center column-group of Table 2 shows the number of false positives for different settings of `CI`. With `CI` enabled, Saphire did not raise any false positives during our evaluation as it conservatively assumes that an unresolved include can refer to any script within the web app. While this conservative setting reduces false positives, it results in slightly larger allowlists (see the bottom two regions of the Figure 4 plots). When `CI` is disabled, system-call profile size is decreased (i.e., system-calls under the black line in Figure 4), but false positives do occur. Specifically, we encounter three false positives all within Joomla version 2.5.25 and 3.7. The reason for all three false positives is the automatic generation of aliases as explained in §5.2. Concretely, `administrator/index.php` instantiates `JHttp` which in turn relies on the built-in PHP function `curl_exec`. Although, Saphire's stage ① correctly determines that `curl_exec` requires the `getpeername` and `setsockopt` system-calls, stage ② misses the dependency introduced by instantiating the `Http` Joomla-class via its alias `JHttp`. The simple, yet Joomla-specific, modification to Saphire described above would remove these false positives. Saphire handles calls to APIs that depend on external binaries. For example, Drupal relies on the `mail()` API during user registration. Since Stage ① tracks the system-calls that the external `sendmail` binary performs, we observed no false positives from such functionality. Finally, though we did not have access to any popular PHP sites, we installed Saphire on a public web-server running WordPress. In total, our web-server received 13,261 HTTP request. Though many of these requests originated from benign crawlers, some appeared to search for unsecured API endpoints, such as WordPress' `xmlrpc.php`. None of these requests triggered Saphire alerts. We performed a manual inspection of the web-server's filesystem to confirm that it had not been compromised.

5.4.2 Payload Constraints

Table 2 also presents the effect of Saphire's script deprivilaging on the attackers using web app exploits to execute the Metasploit payload in the "Exploits Blocked" columns. Of course, adversaries are not limited to Metasploit payloads and can craft exploits which do not extend past the exploited script's system-call privileges.

To assess the impact of such attacks, we enumerate which *dangerous* system calls are present in the allowlist for scripts that contain RCE vulnerabilities with `CI` enabled. We consider a system-call dangerous if it is listed as "Threat level 1 system call" in [8]. All remaining dangerous system calls for the corresponding vulnerable files are shown in the last column of Table 2. While attackers are free to modify the exploits, they can only use the dangerous system calls listed in the table. Notably, none of the payloads can spawn new processes outside the interpreter (no `execve`). `CVE-2016-10033` is specific to Apache. Though we primarily test against nginx, we configured an Apache server with Saphire protections. The steps for configuring Saphire for nginx and Apache were virtually identical, aside from differences in the config-file syntaxes. `CVE-2016-10033`, leverages parameter injection to the external sendmail executable through PHP's mail() function. Since Saphire collected an accurate profile for the mail() and the sendmail child process, we defend against this CVE, without raising false-positives on the same page. The remaining dangerous system-calls potentially allow the attacker to tamper with website-content, but are insufficient to achieve arbitrary code-execution. Saphire protects against RCE launched via file upload vulnerabilities by default. If an attacker exploits a file-upload vulnerability, the uploaded script will have an empty system-call allowlist, as the script was not present during the stage ② analysis. Thus, the uploaded script cannot make any system-call and cannot meaningfully contribute to the attacker's goals.

Additionally, we test Saphire against a set of 40 real payloads. Though there are few php-based payload datasets readily-available, we ran the payloads in one such dataset [19] and found that every payload relies on system-calls missing from the profiles for vulnerable scripts listed in Table 2.

		False Positives		
Plugin Name	Web App	`CI` off	`CI` on	Coverage
ContactForm	Wordpress	0	0	39.14%
Yoast	Wordpress	0	0	†28.20%
Akismet	Wordpress	0	0	32.53%
WooCommerce	Wordpress	0	0	†27.93%
Classic Editor	Wordpress	0	0	40.76%
Akeeba	Joomla	0	0	14.67%
Acymail	Joomla	0	0	15.66%
Ctools	Drupal	0	0	†27.60%
Views	Drupal	0	0	†43.69%

Table 3: False positive test for popular web app plugins. Coverage marked with † was gathered with the aid of available unit-tests.

5.4.3 Analysis of Non-vulnerable Plugins

Most web apps in our evaluation dataset feature powerful plugin architectures. As such, we assess whether Saphire triggers false-positives in the plugins that leverage this infrastructure. Using the above ensemble of three methods to determine coverage, we exercise the popular plugins to assess Saphire for false positives (see Table 3). On average we achieved 31.76% line coverage which is in line with existing work focusing on Web Apps [4], though our statistics also cover plugins. Some plugins guard premium features behind paywalls. Since we did not pay for the plugins, these features contributed unreachable code, lowering the coverage we could achieve.

5.4.4 Runtime overhead

Response time is a critical metric for web-server workloads. We note that Saphire's analysis stages, ① and ②, are performed offline. Stage ③ uses a PHP extension to sandbox a web-app, by loading a system-call profile for the PHP script, at the beginning of each request. Additionally, Saphire relies on different system-call profiles for each script. Since `seccomp` does not allow Saphire to replace the system-call profile, after a process handles a request, by default, Saphire configures PHP to restart the process after serving each request. Modern web-servers, such as nginx with `php-fpm`, typically reuse PHP processes to handle multiple requests, and we consider this in our evaluation. We perform two experiments on a system, with an 8-core Intel Xeon E5-2620v2 @2.10GHz, 256GiB DDR3, running Linux 4.17, with nginx 1.14, PHP 7.1 with `php-fpm`, and MySQL 5.7.

We measure Saphire's overhead by observing the response time for WordPress' `index.php` with ApacheBench [16], over 15,000 requests, at multiple levels of request concurrency. We compare the default configuration of `php-fpm` against `php-fpm` configured to use processes for a single request. The results are presented in Table 4 and indicate that overhead is negligible at all levels of concurrency. To further generalize these results, we repeated the experiments for Apache 2.4.

Additionally, we benchmarked a *worst-case* scenario for Saphire, where the interpreter executes a trivial script. The script prints a single line of text, prior to exiting. We use ApacheBench to benchmark the trivial script across 50,000

	Concurrency	Wordpress		Trivial Script		
		Default	Protected	Default	Protected	Optimized
nginx	1	328.252	328.78 (0.16%)	0.185	1.941	0.188 (1.62%)
	2	353.982	355.776 (0.51%)	0.192	2.316	0.194 (1.04%)
	4	348.242	348.639 (0.11%)	0.264	4.347	0.265 (0.38%)
	8	361.377	363.83 (0.68%)	0.512	8.62	0.516 (0.78%)
	16	416.639	419.342 (0.65%)	0.924	18.61	0.93 (0.65%)
	32	863.932	867.932 (0.46%)	1.71	43.38	1.713 (0.18%)
Apache	1	338.75	337.42	0.201	1.91	0.204 (1.49%)
	2	368.84	370.02	0.209	2.44	0.212 (1.43%)
	4	369.48	369.55	0.236	4.53	0.233 (1.28%)
	8	372.84	372.98	0.559	8.77	0.564 (0.89%)
	16	412.47	414.42	0.954	19.02	0.961 (0.73%)
	32	872.57	877.24	1.77	42.21	1.78 (0.56%)

Table 4: Response times for requests to WordPress index.php and a worst-case, trivial script. All response times in milliseconds.

requests under default and protected `php-fpm` configurations. The results are presented in Table 4, in the first two columns under the *Trivial Script* heading. We observe, that disabling reuse of PHP workers has a severe impact on performance for the worst-case script, since each interpreter process is only active for a short time, before it must be restarted.

To avoid the performance penalty due to the `php-fpm` configuration change, Saphire takes advantage of `php-fpm`'s built-in pooling feature, and nginx URL-routing capabilities. First, an administrator specifies a set of high-demand PHP pages (this information is easily obtained from server logs). Saphire configures separate `php-fpm` pools for each specified page, and creates nginx rules to route requests to the proper pool, based on the URI. Saphire also creates a catch-all pool, where processes are not reused, for scripts that are in low-demand. Note, that the total number of `php-fpm` processes does not increase, and `php-fpm` automatically assigns and removes workers to each pool based on demand.

This configuration change enables protected `php-fpm` workers to process multiple requests, without restarting to re-apply the only installing the seccomp filter once. We present the benchmarks for this configuration in the last column of Table 4. Observe that by configuring nginx to route requests to script-specific pools, we eliminate virtually all overhead.

Artifact Availability: Saphire is open-source and available at `https://github.com/BUseclab/saphire`. We provide the Selenium traces, and vulnerable web app containers that we used to evaluate Saphire, along with instructions for reproducing the experiments. These artifacts were major components of our evaluation and we believe that they can be useful for future evaluations.

6 Limitations and Discussion

In this section, we discuss the limitations of the Saphire prototype and possible areas for future work.

eval and system: `eval()` evaluates a string as PHP code. Saphire does not consider includes, or calls to built-in PHP functions inside `eval()` arguments. `system()` executes an arbitrary shell command . None of the false-positives we observed resulted from `eval` or `system` calls, and Saphire supports execution of pre-determined external programs such as `sendmail` through the `mail()` API function. In future work, Saphire can be improved, to analyze static content in arguments to `eval` and `system`.

Mimicry: Saphire's goal is to apply the PoLP, as it relates to system-calls, to interpreted applications. This severely restricts the system-calls that the exploit and payload can rely on. In section 5.4.2, we discuss the scarcity of "dangerous" system-calls available to attackers. Even so, Saphire does not explicitly detect ACE attacks, and the attacker can attempt to craft a payload that only invokes allowed system-calls. For example, the attacker might still leverage vulnerabilities to add undesired content to content management systems.

Overwriting scripts: Saphire's system-call profiles are read-only to the PHP interpreter. If an attacker has write access to scripts on an upload path, they can, potentially, overwrite an existing script with a payload. Saphire will limit the uploaded script with the allowlist it built in Stage ②. Therefore, the attacker can overwrite a script with a larger system-call privilege-set. If scripts must be writeable, Saphire can be easily augmented to record a checksum for each PHP script during Stage ② and ensure that the checksum is unchanged, when the script is loaded in Stage ③.

Writing to Sensitive Files: Saphire aims to limit the system-calls accessible from a compromised PHP interpreter. For standard linux users, even essential system-calls, such as `open()` and `write()` can be leveraged to gain full code-execution capabilities. For example, an attacker can add malicious commands to automatically executed scripts, such as `.profile` or `.bashrc`, or append ssh-keys to `/.ssh/authorized_keys` to gain remote ssh access. We examined the possibility of such attacks by enumerating the files and directories writeable by the interpreter's user in three common configurations: 1. WordPress running in the official Docker container 2. WordPress on a Debian 10 VM installed according to instructions on the WordPress site. 3. WordPress and phpMyAdmin on a Debian 10 VM installed using Debian's APT package manager. Apart from globally writeable files and directories such as `/tmp/` and `/dev/shm/`, the `www-data` running the interpreter has access to `/var/log/php` for logging purposes and `/var/lib/nginx` (`/var/lib/apache` for the docker container) and `/var/lib/php` which contain web-server daemon lock-files and socket files. Note that `www-data` does not have write-access to its home directory, `/var/www/`, or any server configuration files in `/etc/`. The packaged web apps also run as the `/var/www` user, though since updates are managed by APT running as root, the web apps are stored in a root-owned directory in `/usr/share`. Directories that must be writeable by the web-server (such as WordPress' `wp-content`) are located in `/var/lib` with corresponding permissions.

Additionally we used ptrace to verify that the interpreter does not have access to any privileged file-descriptors(for example, ones left open after a privilege drop using `setuid/setgid`). With these file permissions, and the previously mentioned protections against overwriting scripts, we found no way for a web-server user to create or modify files to gain code-execution.

Installing plugins: When a site administrator installs a new plugin into a web app, they run stage ② on the plugin source in a safe directory, and then Saphire merges the plugin's system-call profile into the profile for the rest of the web app. Currently, ② is run manually for new plugins, removing some of the convenience of web apps that support installing plugins directly through the web-interface.

Saphire does not filter system call arguments: Saphire applies the PoLP to PHP scripts, where it considers each system-call type as privilege. This idea can be further extended to consider system-call arguments as privileges. Though system-call arguments can be derived from user-input, at run-time, , unchanging arguments can be determined statically during Saphire's stage ② and filtered in ③. In our evaluation over 21 exploits, Saphire blocked all attacks by simply filtering based-on system-call type and we leave argument-based PoLP to future work.

Line coverage of evaluated web apps: We use an ensemble of human-driven and automatic techniques, as well as unit-testing to test for false-positives. In our evaluation, we achieve an average line coverage of 33.28% which is in line with similar work [2–4, 6, 14, 35]. Unlike these works, our web apps contain large plugins. Since phpMyAdmin 3.3.10 and Joomla 2.5.25 do not include test-suites, their coverage is significantly lower. Another factor limitting possible coverage is the fact that web apps often rely on small fractions of large frameworks. For example our WordPress installation contains the wp-property plugin, which includes TCPDF (a PDF generator). The wp-property code does not reference TCPDF anywhere, so this idle code (39k lines, or 11% of our WP install) is likely unreachable. Additionally, we found that 14.66% of uncovered WordPress code is only executed during installation/update (which is performed offline, with Saphire). A further 16.28% of the code was specific to the SimplePie and ID3 components and accessible, only by providing specially formatted RSS and Audio File inputs. Additionally, less than 1% of lines were specific to the Windows platform. Since we performed our evaluation on Linux, we could not exercise these lines.

Applying Saphire to other interpreters: In this paper we present a framework for allowlisting system-calls in interpreters, on a per-script basis. Based on our experience implementing Saphire for PHP, we identified the interpreter characteristics that are required to apply Saphire to other interpreters:

1. Each interpreted program should be executed in its own instance of the interpreter.

2. Each program can only invoke system-calls by calling into a native interpreter function, or a foreign-function interface addon. There must be a dispatch table mapping built-in function names within the interpreter to the native implementations of those functions.

3. The interpreted language should be amenable to an inter-procedural analysis to statically identify dependencies between scripts.

To the best of our knowledge, these requirements are satisified for:

1. A Python interpreter running a program composed of multiple python scripts. The program executes within a single Python process (potentially with multiple threads). It can only rely on syscalls implemented in the standard libraries included by the scripts (open(), read()...), or FFI libraries.

2. A server executing multiple Node.JS microservices. Each individual service can be restricted to only the system-call it requires. In Node.JS, programs usually call into the Node.JS api using calls such as `fs.open`, which in-turn call C++ code registered in the dispatch table.

3. Any classic CGI-based interpreted web-app using a language such as Perl or Lua. CGI launches a separate interpreter process for each request, which can be protected by a Saphire-flavored approach, as long as there are sufficient means to analyze to code.

7 Related Work

As system-calls guard access to sensitive OS-managed resources, there is abundant research related to system-call based sandboxing, focused on restricting resources available to an application [15, 18, 20, 28, 31, 42, 48, 55], intrusion detection systems [23, 39, 49, 51, 56, 61] and confining Linux containers [29, 32, 57]. Janus [55] relies on system-call interposition with *ptrace* to intercept and filter dangerous system-calls, according to defined policies. Plash [48] restricts a process by executing it in a chroot environment with a set of instrumented system-calls, relying on an RPC server. Systrace [42] generates system-call policies interactively, with input from the user. Systrace requires the user to manually modify the policies for applications which pass non-deterministic arguments to system-calls [42]. Ostia [18] and REMUS [9] rely on user-specified rules to filter system-calls.

Unlike our approach which is completely automatic, [9, 18, 42, 48, 55] require user involvement in profile generation. *N*-gram-based allowlists make decisions based on whether sequences of system-calls were observed during benign execution [15, 49, 56], but rely on representative sets of benign executions. Janus, Systrace, Ostia, and the *N*-gram-approaches incur significant overhead, which makes them impractical [33].

Unlike Saphire, prior filtering approaches do not tailor system-call profiles to individual interpreted programs. Since interpreted programs are prime targets for attackers today, this is a major limitation.

SELinux [34] leverages role-based access control and multi-level security to implement Mandatory Access Control and enforce restrictions on data for user roles. AppArmor [11] restricts a program's access to files and capabilities according to a profile. Both SELinux and AppArmor require an administrator to manually specify, or dynamically collect a security profile, which is non-trivial. AppArmor and SELinux's protection cannot distinguish between the execution of different programs by an interpreter. FMAC [41], creates an access profile based on benign inputs to a program, and uses it to deny restrict access to files. MAPbox [1] allows a user to manually specify a list of acceptable application behaviors, each of which corresponds to a sandbox configuration. Boxmate [29] confines an Android app to the set of resources it accessed during a training stage. Boxmate blocks any access to resources that was not accessed during training. Wan et. al [57] extends Boxmate to Linux containers , by recording a list of the accessed system calls during automatic testing and using this as a allowlist for filtering system-calls in Linux containers. Boxmate [29] and [57] confine processes and are not fine-grained enough to identify the execution of different scripts by an interpreter.

Static Analyses: Several proposed approaches focus on detecting vulnerabilities in the source code of web apps, statically, [5, 12, 13, 30, 53, 58, 62]. These approaches use taint analysis to track unsanitized data to find potential vulnerabilities. Zheng et al. [62] rely on a SAT solver to find connections between user controlled inputs and critical PHP functions and identifies eight previously unknown vulnerabilities. Backes et al. [5] detect injection vulnerabilities using code-property graphs. Dahse et al. [13], track data-flows to detect injection vulnerabilities. RIPS [12] conducts taint analysis by building a control flow graph for PHP files in web apps to find injection vulnerabilities. Several approaches target specific classes of injection vulnerabilities, including cross-site scripting [30], SQLi [58], and Denial of Service(DoS) [53]. SaferPHP [53] uses taint analysis and symbolic execution to find DoS vulnerabilities by examining the semantics of web app code. Due to dynamic features of Web Apps, it is difficult to statically build accurate CFGs. Saphire's flow-insensitive analysis of web apps is lightweight compared to approaches that require comprehensive CFG analysis to pinpoint vulnerabilities.

Dynamic Analyses: WASP [21], Diglossia [52], SCRIPTGARD [47] and ZenIDS [22] rely on dynamic analysis to detect specific categories of vulnerabilities. WASP [21] uses syntax evaluation and positive tainting for tracking trusted data to detect and prevent SQLi attacks on web apps. WASP requires the developer to specify trusted sources and trust policies, manually. Likewise, Diglossia [52] uses dynamic taint analysis to detect and prevent SQLi attacks. SCRIPTGARD [47] tracks the flow of trusted data using data flow analysis to find inconsistencies in sanitization functions of ASP.NET web apps. ZenIDS instruments the PHP interpreter to collect and merge CFGs for trusted executions during a training stage. To defend a web app against RCE, ZenIDS traces the control flow throughout the execution of incoming requests and raises an alert if a control flow transition does not exist within the trusted profile. ZenIDS relies on a representative set of known benign requests, for each protected web app. Saphire does not require dynamic CFG exploration or dynamic expansion stages, opting for low-overhead system-call-based enforcement.

Hybrid approaches: WebSSARI [40] builds CFGs and instruments web apps with guards, preventing insecure information flows. SQLBLock [27] deploy a hybrid static-dynamic analysis to protect web apps against SQLi attacks. Saner [7] uses static and dynamic techniques to verify sanitization routines, by tracing inputs through web apps. The defense of such techniques hinges on their ability to precisely and consistently pinpoint vulnerable code. Saphire uses dynamic analysis to refine the mapping in stage ①, but the goal of the combination is to build a profile of system-calls for each script, rather than to find vulnerabilities.

8 Conclusion

We identify that interpreters add a layer of abstraction, reducing the practicality of many deprivileging techniques. We propose a novel solution for generating and applying system-call filters to interpreted applications. Saphire is our prototype implementation of the system, for PHP web apps. During an offline analysis of the PHP interpreter and web app, Saphire determines the system-calls required by each script. Then, Saphire uses `seccomp`, to filter unneeded system-calls. In our experiments, we find that Saphire blocks 21 real-world web app exploits, without raising false-positives for benign use. System-call allowlists for vulnerable scripts contain a minimal amount of system-calls identified as dangerous. Saphire's extension only needs to activate once per request and does not cripple performance.

Acknowledgements

We thank our anonymous reviewers for their insightful comments and feedback. This work was supported by the Office of Naval Research (ONR) under grant N00014-17-1-2541.

References

[1] Anurag Acharya and Mandar Raje. Mapbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Conference on Security Symposium*, 2000.

[2] Nuno Antunes and Marco Vieira. Benchmarking vulnerability detection tools for web services. In *IEEE International Conference on Web Services*, 2010.

[3] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Daniel Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 2010.

[4] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.

[5] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *IEEE European Symposium on Security and Privacy*, 2017.

[6] Young-Min Baek and Doo-Hwan Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[7] Davide Balzarotti, Marco Cova, Viktoria Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *IEEE Symposium on Security and Privacy*, 2008.

[8] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. Enhancements to the linux kernel for blocking buffer overflow based attack. In *Proceedings of the 4th Annual Linux Showcase & Conference*, 2000.

[9] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. Remus: A security-enhanced operating system. *ACM Transactions on Information and System Security*, 2002.

[10] G. Cleary, M. Corpin, O. Cox, H. Lau, B. Nahorney, D. O'Brien, B. O'Gorman, J. Power, S. Wallace, P. Wood, and Wueest C. Internet security threat report. Technical Report 23, Symantec Corporation, 2018.

[11] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. Subdomain: Parsimonious server security. In *Proceedings of the 14th USENIX Conference on System Administration*, 2000.

[12] Johannes Dahse and Thorsten Holz. Simulation of built-in php features for precise static code analysis. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[13] Johannes Dahse and Thorsten Holz. Static detection of second-order vulnerabilities in web applications. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[14] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proceedings of the 21st USENIX Conference on Security Symposium*, 2012.

[15] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *IEEE Symposium on Security and Privacy*, 1996.

[16] Apache Software Foundation. ab - apache http server benchmarking tool. `https://httpd.apache.org/docs/2.4/programs/ab.html`, November 2018.

[17] Jessie Frazelle. A rant on usable security. `https://blog.jessfraz.com/post/a-rant-on-usable-security/`, October 2018.

[18] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed System Security Symposium*, 2004.

[19] Mattias Geniar. Code obfuscation, php shells & more: what hackers do once they get passed your (php) code. `https://github.com/mattiasgeniar/php-exploit-scripts`, March 2014.

[20] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th USENIX Conference on Security Symposium, Focusing on Applications of Cryptography*, 1996.

[21] William G.J. Halfond, Alessandro Orso, and Panagiotis Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 2008.

[22] Byron Hawkins and Brian Demsky. ZenIDS: Introspective intrusion detection for php applications. In *Proceedings of the 39th International Conference on Software Engineering*, 2017.

[23] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6, 1998.

[24] Imperva. The state of web application vulnerabilities in 2017. `https://imperva.com/blog/the-state-of-web-application-vulnerabilities-in-2017/`, October 2018.

[25] Imperva. The state of web application vulnerabilities in 2018. `https://imperva.com/blog/the-state-of-web-application-vulnerabilities-in-2018/`, October 2018.

[26] IDG Communication Inc. phpmyadmin. `https://www.pcworld.com/article/233948/phpmyadmin.html`, October 2018.

[27] Rasoul Jahanshahi, Adam Doupé, and Manuel Egele. You shall not pass: Mitigating sql injection attacks on legacy web applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 445–457, 2020.

[28] Kapil Jain and R Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of the Network and Distributed System Security Symposium*, 2000.

[29] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. Mining sandboxes. In *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[30] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, 2006.

[31] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *Proceedings of the 22nd USENIX Conference on Annual Technical Conference*, 2013.

[32] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. SPEAKER: Split-Phase Execution of Application Containers. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

[33] Cullen Linn, Mohan Rajagopalan, Scott Baker, Christian S. Collberg, Saumya K. Debray, and John H. Hartman. Protecting against unexpected system calls. In *Proceedings of the 14th USENIX Conference on Security Symposium*, 2005.

[34] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, 2001.

[35] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[36] Keiren McCarthy. Panama papers hack: Unpatched wordpress, drupal bugs to blame? `https://www.theregister.co.uk/2016/04/07/panama_papers_unpatched_wordpress_drupal`, October 2018.

[37] Metasploit. metasploit. `https://www.metasploit.com`, June 2019.

[38] Paul Moore. libseccomp. `https://github.com/seccomp/libseccomp`, November 2018.

[39] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous system call detection. *ACM Transactions on Information and System Security (TISSEC)*, 2006.

[40] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Security and Privacy in the Age of Ubiquitous Computing*, 2005.

[41] Vassilis Prevelakis and Diomidis Spinellis. Sandboxing applications. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*, 2001.

[42] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Conference on Security Symposium*, 2003.

[43] Q-Success. Usage of server-side programming languages for websites. `https://w3techs.com/technologies/overview/programming_language/all`, October 2018.

[44] Q-Success. Usage Statistics and Market Share of Content Management Systems for Websites, November 2018. `https://w3techs.com/technologies/overview/content_management/all`, November 2018.

[45] Derick Rethans. Xdebug: debugger and profiler tool for PHP. `https://xdebug.org/`, November 2018.

[46] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 1975.

[47] Prateek Saxena, David Molnar, and Benjamin Livshits. Scriptgard: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, 2011.

[48] Mark Seaborn. Plash: tools for practical least privilege. `http://plash.beasts.org`, June 2019.

[49] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, 2001.

[50] Vadym Slizov. php-parser. `https://github.com/z7zmey/php-parser`, June 2019.

[51] Anil Somayaji and Stephanie Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Conference on Security Symposium*, 2000.

[52] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 20th ACM SIGSAC conference on Computer*, 2013.

[53] Sooel Son and Vitaly Shmatikov. Saferphp: Finding semantic vulnerabilities in php applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for ecurity*, 2011.

[54] WPScan Team. Foxypress 0.4.1.1-0.4.2.1 - arbitrary file upload. `https://wpvulndb.com/vulnerabilities/6231`, June 2019.

[55] David Wagner. Janus: An approach for confinement of untrusted applications. Technical report, University of California at Berkeley, 1999.

[56] David Wagner and Drew Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[57] Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining sandboxes for linux containers. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017.

[58] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[59] David A. Wheeler. `https://dwheeler.com/sloccount/sloccount.html`, 2004.

[60] Dave Wichers. Owasp top-10 2013. *OWASP Foundation, February*, 2013.

[61] Cong Zheng and Heqing Huang. Daemon-guard: Towards preventing privilege abuse attacks in android native daemons. In *Proceedings of the First Workshop on Radical and Experiential Security*, 2018.

[62] Yunhui Zheng and Xiangyu Zhang. Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *35th International Conference on Software Engineering (ICSE)*, 2013.