2017 IEEE/ACM 39th International Conference on Software Engineering

ZENIDS: Introspective Intrusion Detection for PHP Applications

Byron Hawkins, Brian Demsky
University of California, Irvine
{byronh,bdemsky}@uci.edu

***Abstract*—Since its first appearance more than 20 years ago, PHP has steadily increased in popularity, and has become the foundation of the Internet's most popular content management systems (CMS). Of the world's 1 million most visited websites, nearly half use a CMS, and WordPress alone claims 25% market share of all websites. While their easy-to-use templates and components have greatly simplified the work of developing high quality websites, it comes at the cost of software vulnerabilities that are inevitable in such large and rapidly evolving frameworks.**

Intrusion Detection Systems (IDS) are often used to protect Internet-facing applications, but conventional techniques struggle to keep up with the fast pace of development in today's web applications. Rapid changes to application interfaces increase the workload of maintaining an IDS whitelist, yet the broad attack surface of a web application makes for a similarly verbose blacklist. We developed ZENIDS to dynamically learn the trusted execution paths of an application during a short online training period and report execution anomalies as potential intrusions.

We implement ZENIDS as a PHP extension supported by 8 hooks instrumented in the PHP interpreter. Our experiments demonstrate its effectiveness monitoring live web traffic for one year to 3 large PHP applications, detecting malicious requests with a false positive rate of less than .01% after training on fewer than 4,000 requests. ZENIDS excludes the vast majority of deployed PHP code from the whitelist because it is never used for valid requests—yet could potentially be exploited by a remote adversary. We observe 5% performance overhead (or less) for our applications vs. an optimized vanilla LAMP stack.

I. INTRODUCTION

Content Management Systems (CMS) have taken a leading role in web application development, largely because they provide a vast assortment of powerful components that are easily composed into a polished presentation with a convenient user interface. But for the same reason, even a sophisticated application may use only a small fraction of the framework code that it deploys, and vulnerabilities in the remaining code can expose the website to attacks. Table I shows that three websites hosted by our group use just 4.5-11.7% of the underlying framework. Compounding this is the highly dynamic construction of PHP applications, in which code is dynamically loaded from plain text files—and even user input strings—and fundamental program elements such as function call targets are often specified by string variables. From this perspective, the diverse and flexible functionality provided by the framework represents an important trade-off: it provides convenience for development of the website, yet creates a needless security liability for deployment of the site.

TABLE I: Application size vs. the fraction of application code that is trusted by ZENIDS. Most of the untrusted code cannot be reached by normal execution of the application in any configuration—yet can be reached by attackers as a faux entry point, or via branch manipulation.

	Source Lines		Source Files	
	Original	Trusted	Original	Trusted
WordPress	130,705	15,392	472	227
GitList	163,248	7,325	2,625	213
Doku	139,340	10,219	928	127

Instead of settling for a compromise between convenience and security, we developed ZENIDS[1] to accurately and efficiently detect malicious intrusions. Users retain the freedom to install, configure, customize and even extend any PHP application. During a short online training period, ZENIDS learns the set of execution paths that the deployed application is using. The ZENIDS monitor raises an intrusion alert when the execution of a request in an unprivileged session diverges from the set of trusted execution paths. Since ZENIDS is extremely sensitive to variations in execution, site changes of any kind could result in a high rate of false positives—for example, if a blogger writes a post having as-yet-unused formatting such as a table, the PHP code that renders the post in HTML may take different paths than for any previous post. To accommodate natural evolution in an application's usage of its underlying framework, ZENIDS selectively trusts new control flow paths that are directly associated with data changes made by privileged users. This allows developers and administrators to use any part of the framework's rich feature set, yet prevents abuse of both deployed code and dynamic control flow branches that the site is not presently using.

A. Overview

To protect a website with ZENIDS, the administrator installs the instrumented PHP interpreter with the ZENIDS extension in an otherwise standard LAMP or WIMP stack. For applications that implement user privileges, a hook must be added in the application's PHP code to notify ZENIDS of login and logout events. ZENIDS learns the set of features that site visitors are currently using by recording execution traces to a *trusted profile* for a short period of time. In monitoring mode, ZENIDS raises an intrusion alert when the execution of an unprivileged HTTP request diverges from the trusted profile.

[1]The source code of the PHP interpreter uses the prefix `ZEND_*` in honor of authors **ZE**ev and a**ND**i. Since our approach applies to interpreted languages in general, we use the more common web 2.0 moniker "Zen" for our tool.

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.29

Our experiments in Section VII demonstrate that ZENIDS detects recent attacks against vulnerable applications, yet rarely raises a false alert when deployed on the same applications receiving live Internet traffic. After configuring a WordPress site with 9 vulnerable plugins and a vulnerable theme, ZENIDS detected attempts to exploit all 10 vulnerabilities, only raising false alerts on invalid form entries. We recorded HTTP traffic to live deployments of WordPress, the GitList repository viewer, and DokuWiki for 360 days, then replayed the traffic to replica sites monitored by ZENIDS. The false positive rate was less than .01%, yet ZENIDS raised 38,076 true alerts among more than 1.5 million requests. Privileged users made changes to these sites during the experiment that would have resulted in false alerts on every request, but ZENIDS safely expanded the trusted profile, meanwhile continuing to raise true alerts on malicious requests.

A trivial implementation of ZENIDS has 10× overhead vs. a LAMP stack having typical optimizations. In Section VI we employ redundancy elimination and caching techniques to reduce overhead below 5% without compromising security.

B. Intended Usage Scenario

We envision ZENIDS being deployed both by web site administrators and by cloud providers who wish to provide an extra service to their users. Although our false positive rate is extremely small, in many scenarios automatically blocking traffic is unacceptable due to the small risk of blocking legitimate visitor requests. Thus, we have designed ZENIDS to provide users with alerts of potential attacks. Users can then manually review the alerts and either write rules to drop the malicious requests or whitelist the control flow as benevolent. Our experiments show that the vast majority of ZENIDS alerts correspond to real attacks. For higher risk deployments where training ZENIDS on live web traffic is less practical, our results show it is feasible to begin training with artificial or trusted traffic, then complete the trusted profile by manually reviewing alerts during an initial segment of live traffic.

C. Contributions

This paper makes the following contributions:

- A technique for recording a trusted profile of application features that are currently used by unprivileged visitors.
- A taint-tracking technique to safely expand the trusted profile according to changes made by trusted users, meanwhile continuing to detect anomalous requests.
- An implementation of ZENIDS that supports all features of PHP 7 and performs at low overhead on large web frameworks such as WordPress and Symfony.
- An evaluation of the performance, usability and security of ZENIDS in popular web applications facing live Internet traffic and recently reported exploits.

II. THREAT MODEL

ZENIDS is designed to defend a PHP web application against a typical remote adversary who does not have login credentials, but may attempt to open a connection on the web server using any port and protocol available to the general public. The adversary can determine the exact version of the protected PHP application and the PHP interpreter, including ZENIDS, and has obtained the complete source code. A binary attack on the PHP interpreter cannot change the execution of the script other than to crash the process.

The adversary does not know when the ZENIDS training period occurs, and is not able to access the trusted profile without first compromising the protected application. Web server authentication prevents untrusted uploads, except as permitted by the protected application itself. At the time of installation, the adversary had no opportunity to modify any files of the protected application; i.e., we assume the original installation is free of any backdoors that were specifically deployed by the adversary as part of the attack. There may, however, be backdoors in the application that are discoverable by the adversary at the time of its public source code release.

III. SYSTEM OVERVIEW

ZENIDS protects a PHP application by raising an intrusion alert when execution of an untrusted user's request diverges from known-safe behavior, as recorded in the trusted profile. Fig. 1 depicts the workflow of a basic implementation of ZENIDS. During an initial training period of configurable duration, ZENIDS records the control flow graph (CFG) of each request, which is just the union of all edges in the opcode trace, including the authentication level at each edge. At first there may be a high degree of variation among the CFGs, indicating that training is not yet sufficient and should continue. When the rate of unfamiliar control flow among untrusted requests tapers off, the CFGs are merged into the trusted profile by union (without path or context sensitivity), preserving the lowest observed authentication level at each edge. After the profile is deployed to the webserver, ZENIDS consults it to evaluate the safety of requests from untrusted users, raising an alert for edges that are not known to be safe.

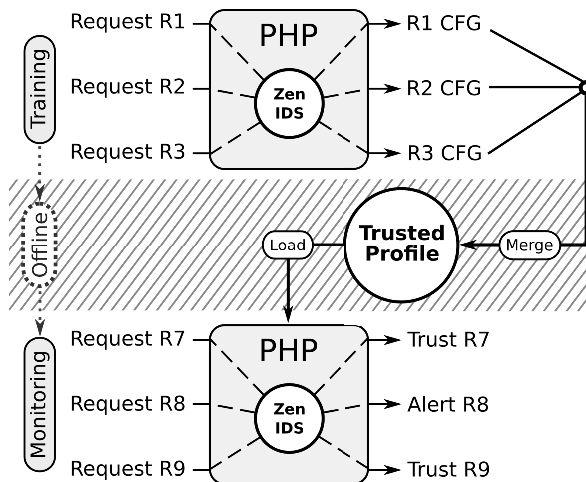


Fig. 1: Deployment of ZENIDS begins by recording the CFG of each request during an initial training period. These are merged offline into the trusted profile, which is then deployed to ZENIDS for monitoring.

A. Profiling

PHP interprets a script by first compiling it into opcode sequences (e.g., one per function) and then interpreting the sequences. Fig. 2 illustrates this, starting from (a) the source code, which is compiled into (b) an opcode sequence, then recorded by ZENIDS during execution to (c) the trusted profile. Any opcode may jump within its sequence or enter another sequence. From the source code perspective, a transition between opcode sequences represents an entry into any PHP script segment—not necessarily a function call—but for simplicity, ZENIDS models them all as ordinary calls:

a) Importing Script Files: When execution reaches an `include` or `require` statement, the PHP interpreter executes the body of the imported script—which encompasses any code outside of class and function declarations—and then adds any classes and functions to the appropriate namespaces. ZENIDS models all forms of the `include` statement as a function call to the script body of the imported file.

b) Callbacks: Built-in functions may implicitly invoke script functions, for example if the script tries to set an undefined object property, the PHP built-in accessor calls the object's `__set()` method as a fallback. ZENIDS models such callbacks with an edge from the call site directly to the callback (Fig. 2c), to avoid a nexus effect that would weaken the trusted profile. For example, in an exploit of the WordPress Download Manager plugin [1], the adversary creates a new privileged user by manipulating the target of the PHP built-in `call_user_func()`. If we allowed the built-in to become a nexus, ZENIDS would not know which call site normally (safely) reaches which callee, making the exploit undetectable.

c) Object Management: Structural methods like classloaders and destructors are invoked under complex conditions that may vary across executions of the same HTTP request. For consistency, ZENIDS creates an edge from a symbolic *system* node to the corresponding script method (e.g., `__destruct()`). If the application should directly invoke one of these methods (perhaps under adversarial influence), ZENIDS will distinguish it as a conventional call edge.

d) Dynamic Code: The PHP language provides two methods for dynamically evaluating a string as PHP code: `eval()` and `create_function()`. Since the interpreter compiles them as anonymous functions, ZENIDS models them as function calls having dynamic targets (see Section IV-B).

B. Monitoring Granularity

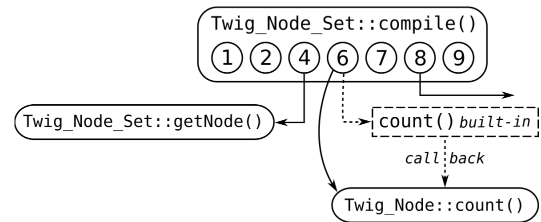
To detect the RCE exploits that are being reported against PHP applications today, it is only necessary to monitor the call graph. This is largely because there are no reported techniques for manipulating intra-procedural control flow into crossing procedural boundaries, as is possible on other platforms. Furthermore, the known compositional attacks at the inter-procedural level always involve calls that do not occur in normal execution (see Section IV-A). While it may be feasible to craft more subtle exploits, there is presently no such threat to the PHP ecosystem—and as we have shown, there are many reasons to believe that such attacks may in fact be infeasible.

```
46 public function compile(Twig_Compiler $compiler)
47 {
48   if (count($this->getNode('names')) > 1) {
49     $compiler->write('list(');
```

(a) PHP code sample from the Symfony template compiler Twig.

#	Line	Result	Opcode	Operand 1	Operand 2
0	46	-	ZEND_RECV	`$compiler`	-
1	48	-	ZEND_INIT_FCALL	`count`	-
2	48	-	ZEND_INIT_METHOD_CALL	`getNode`	-
3	48	-	ZEND_SEND_VAL_EX	`"names"`	-
4	48	`tmp #1`	ZEND_DO_FCALL	-	-
5	48	-	ZEND_SEND_VAR	`tmp #1`	-
6	48	`tmp #2`	ZEND_DO_FCALL	-	-
7	48	`tmp #3`	ZEND_IS_SMALLER	`1`	`tmp #2`
8	48	-	ZEND_JMPZ	`tmp #3`	`+25`
9	49	-	ZEND_INIT_METHOD_CALL	`$compiler`	`write`

(b) The sample (a) compiled by PHP into an opcode sequence.



(c) The trusted profile CFG of the opcode sequence (b). At opcode #6, the callee `count()` is a built-in function that either counts built-in collections such as arrays, or calls back to the object's `count()` method if it has one. ZENIDS models the callback as an edge connecting opcode #6 directly to `Twig_Node::count()` to avoid having hundreds of edges from a nexus `count()` node.

Fig. 2: Execution of a PHP code snippet under ZENIDS profiling.

C. Dynamic vs. Static Analysis

On many platforms such as C/C++, it is possible to construct a substantially accurate control flow graph on the basis of static analysis, avoiding the administrative overhead and potential inaccuracy of dynamic profiling. This approach is not viable for RCE exploits against PHP, because even though the source code is always available, there is a common idiom of calling functions by string name which defies static analysis. Since a call by name has no structural properties to limit the set of potential target functions, static analysis falls back to an over-approximation that is multiple orders of magnitude too coarse for security purposes. These call-by-name sites are quite common, for example more than 100 occur in the WordPress core alone. Section VIII-c discusses the limitations of several web defenses that rely on static analysis.

D. Monitoring Ubiquity

A common weakness among intrusion detectors is that an informed adversary can mimic trusted input, or mask those aspects of the attack that the IDS is capable of observing. For example, KBouncer [2] and ROPecker [3] use the Last Branch Record register to detect a Return-Oriented Programming (ROP) attack within the last n branches, asserting that evidence of ROP can always be found within that window. But [4] evades this "line of defense" by composing the

attack with deliberately long ROP chains. Other naïve defense techniques have been similarly defeated by simple tricks in [5] and [6]. In the context of ZENIDS, a vulnerability leveraged by a given exploit is either present in or absent from the trusted profile. If the vulnerability is absent, the vulnerable code cannot be executed without triggering an alert. For an exploit to go undetected, it must fully conform to trusted control flow paths—including the execution of the malicious payload. This approach potentially makes ZENIDS significantly more robust to new threats than prior art.

IV. SYSTEM DESIGN

ZENIDS is implemented as a PHP extension supported by 8 callback hooks instrumented in the interpreter. Fig. 3 shows the main components of the web server, as configured for our experiments, with the hooks labeled **`H1`-`H5`** (the other 3 hooks serve trivial initialization purposes). A basic implementation of ZENIDS only requires the first two hooks: **`H1`** correlates each compiled opcode sequence with its trusted profile (if the sequence is trusted at all), and **`H2`** validates each invocation of an opcode sequence by matching it to an edge in the trusted profile. Listings **`H1`** and **`H2`** present the essential functionality of these hooks in pseudo-code.

A PHP deployment is not limited to the typical configuration that we use for our experiments—it may incorporate numerous extensions, interact with external data sources and services, and be distributed across multiple servers having various architectures and operating systems. But these factors do not interfere with the ZENIDS detection mechanism. At its core, the PHP interpreter is simply a recursive iterator over opcode sequences, and to our knowledge there is no configuration that substitutes or modifies the central Opcode Execution Loop. For this reason, we expect the fundamental approach of ZENIDS to be compatible with PHP deployments of all varieties.

PHP Interpreter Hook `H1` Compile PHP code into *op_seq*

$key \leftarrow$ CANONICAL-NAME(op_seq)
if $key \in trusted_seqs.keys$ **then**
  $trusted_seq \leftarrow trusted_seqs.get(key)$
  if IS-IDENTICAL($op_seq, trusted_seq$) **then**
    $op_seq.trusted_seq \leftarrow trusted_seq$
  else
    ALERT($untrusted_op_sequence$)
  end if
end if

PHP Interpreter Hook `H2` Enter *target_seq* from $op_seq[i]$

1: **if** $op_seq[i].target_seq.trusted_seq = \text{NIL}$ **then**
2:   ALERT($untrusted_app_entry_point$)
3: **else**
4:   $key \leftarrow$ CANONICAL-NAME($op_seq[i].target_seq$)
5:   **if** $key \notin op_seq[i].trusted_targets$ **then**
6:     ALERT($untrusted_call$)
7:   **end if**
8: **end if**

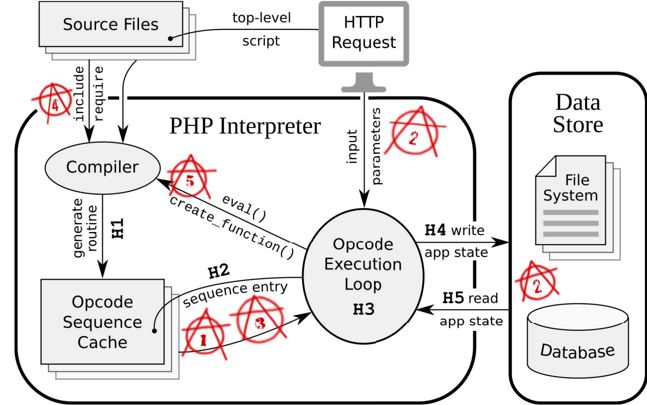


Fig. 3: Components of a typical PHP deployment along with ZENIDS hooks **`H1`-`H5`** and fundamental RCE attack vectors **`A1`-`A5`**.

A. Attacks

Fig. 3 also labels five important attack vectors taken by today's corpus of RCE exploits (**`A1`-`A5`**). The pivotal role of the Opcode Execution Loop in the PHP interpreter makes it possible for ZENIDS to detect all five vectors in hook **`H2`**:

`A1` **Call By Name:** When a function callee is specified as a string constant or variable, PHP resolves the callee using the set of functions defined dynamically during execution. An exploit of the popular WordPress Download Manager plugin creates a new user with administrator privileges by manipulating the target of just one PHP call-by-name [1]. ZENIDS raises an alert at **`H2`** on any untrusted call edge, even if the call site and the callee are in the trusted profile.

`A2` **Object Injection:** The format of serialized objects in PHP specifies both the type and content of each field, making it possible for the adversary to compose arbitrary serialized instances. Dozens of object injection attacks have been reported, such as CVE-2015-8562 against Joomla in which the adversary executes arbitrary code by fabricating a serialized session. In this scenario ZENIDS will detect untrusted edges in the payload at **`H2`**.

`A3` **Magic Methods:** PHP implicitly invokes specially named "magic methods" in certain situations, for example a call to an undefined method `$a->foo()` is forwarded to `$a->__call()` with the name of the missing callee and the arguments (as a fallback). Esser and Dahse combine *object injection* with *magic methods* to create *Property-Oriented Programming* attacks [7], [8] that can execute arbitrary code. While the approach is more complex than **`A2`**, the ZENIDS defense at **`H2`** remains the same.

`A4` **Dynamically Included Code:** The PHP `include` and `require` statements can take a string variable argument, allowing an intruder to import any file. Since ZENIDS models these as calls, **`H2`** will detect untrusted targets.

`A5` **Dynamically Interpreted Code:** PHP can execute a plain text string as code, making the input string vulnerable to attack. ZENIDS models these dynamic opcode sequences as dynamic imports and monitors them at **`H2`**.

B. Canonical Names

For **H1** and **H2** to be reliable, CANONICAL-NAME() must be consistent across executions, since ZENIDS uses it to find trusted profile entries. While it might be trivial in many languages, PHP raises several complications. To begin with, the `include` and `require` keywords are compiled as statements and executed in the opcode sequence (not pre-processed like in C/C++), making it possible for conditional branches to govern the set of imported source files. Function declarations similarly allow for conditional compilation at runtime. Combining these factors with namespaces, it is possible—and very typical—for a PHP application to define the same function identifier multiple times, often among many different files (e.g., to implement a plugin interface). ZENIDS avoids ambiguity in the canonical name using the following schemes:

- ⟨filename⟩.⟨function-name⟩.⟨line-number⟩
- ⟨filename⟩.⟨classname⟩.⟨method-name⟩.⟨line-number⟩

Dynamic code, such as from an `eval()` statement, is correlated to the trusted profile by comparing opcode content, since the origin of the code string can be difficult to determine.

C. Goals of the Basic Implementation

To focus our evaluation of ZENIDS (Section VII), the three technical sections conclude with an enumeration of their security and usability goals, notated **SG#** and **UG#** as follows:

SG1	Detect a broad variety of attacks, including known exploits and live Internet exploits.
UG1	Minimize false alerts, especially in large programs facing live Internet traffic for a long period of time.

V. SUPPORTING WEBSITE EVOLUTION

Changes to the content or configuration of the application may occasionally cause a few new execution paths to be taken, which would cause the basic implementation of ZENIDS to begin raising false alerts. Instead, ZENIDS responds to trusted changes by expanding the trusted profile to enable relevant new code paths. When ZENIDS detects that a privileged user has changed application state, for example in the database, it initiates a *data expansion event* to add corresponding new control flow to the trusted profile. Similarly, if the application generates or modifies PHP source files in a way that conforms to a trusted code generator, ZENIDS initiates a *code expansion event*. The duration of an expansion is limited to a configurable number of requests (2,000 in our experiments) to minimize an adversary's opportunity to manipulate the expansion into trusting unsafe code. Hook **H3** initiates expansions on the basis of the sets $tainted_values$ and $safe_new_sources$, which are maintained by system call monitors **H4** and **H5**.

Listing **H3** shows how ZENIDS adds edges to the trusted profile when the expansion conditions are met. The first condition initiates a data expansion (lines 4-5) at an untrusted branch decision (line 2) when at least one value in the branch predicate carries taint from an administrator's recent data change (line 3). The second condition initiates a code expansion (lines 8-9) when an untrusted call (lines 2 and 6) is made to a safe new source file (line 7). If neither expansion condition is met, and the branch is any kind of call, then an $untrusted_call$ alert is raised (line 11). During an expansion, new branches are added to the trusted profile (line 17) and taint is propagated across all assignments (line 22) and uses of tainted operands (line 25). The expansion ends (line 20) where the initiating branch joins trusted control flow (line 19).

PHP Interpreter Hook H3 Execute i^{th} opcode of op_seq

```
 1: if mode = monitoring then
 2:     if IS-BRANCH(op_seq[i]) &
            branch ∉ op_seq[i].trusted_targets then
 3:         if branch.predicate ∈ tainted_values then
 4:             mode ← expanding
 5:             trust_to ← CONTROL-FLOW-JOIN(branch)
 6:         else if IS-CALL(branch) then
 7:             if call.target_seq ∈ safe_new_sources then
 8:                 mode ← expanding
 9:                 trust_to ← i + 1
10:             else
11:                 ALERT(untrusted_call)
12:             end if
13:         end if
14:     end if
15: else                                    ▷ mode = expanding
16:     if IS-BRANCH(prev_op) then
17:         prev_op.trusted_targets ∪ {op_seq[i]}
18:     end if
19:     if i = trust_to then
20:         mode ← monitoring
21:     else if IS-ASSIGNMENT(op_seq[i]) then
22:         tainted_values ∪ {op_seq[i].predicate}
23:     end if
24: end if
25: PROPAGATE-TAINT(op_seq, i, i + 1)
```

A. Code Expansion Events

Since PHP applications often have repetitive source code, many PHP libraries provide an API to generate new source files at runtime on the basis of application-defined templates.

PHP Interpreter Hook H4 Store application state

```
if IS-TRUSTED-CODE-GENERATOR() then
    safe_new_sources ∪ {new_source}
else if IS-ADMIN(user) then
    state_taint ∪ {stored_state × user.admin_level}
end if
```

PHP Interpreter Hook H5 Load application state

```
if !IS-ADMIN(user) & loaded_state ∈ state_taint then
    tainted_values ∪ {loaded_state}
end if
```

For example, GitList uses the Symfony component Twig to define the HTML page layout for each type of repository page: file list, commit list, blame, etc. At runtime, Twig generates corresponding PHP files on demand. Incorporating this kind of dynamic code into the trusted profile is easy for ZENIDS if the code generator runs during the training period. But a demand-based code generator may run at any time—for example, in our GitList experiment (Section VII-B), crawlers found unvisited views several weeks after the 2-hour training period.

To continue trusting these known-safe code generators, each time the application persists a state change to the database or the filesystem, IS-TRUSTED-CODE-GENERATOR() in **H4** determines whether the application has just written a safe new source file. This function examines the following criteria:

1) **Call Stack:** At the API call to update the database or write to a file, does the call stack match any of the code generator call stacks recorded during the training period?
2) **User Input Taint:** If during training the application never generated source code using values influenced by user input, then ZENIDS checks whether the data for this state change carries taint from user input. This criterion tracks both directly and indirectly tainted values (and may be disabled to avoid continuous taint tracking overhead).
3) **Generator Visibility:** Hook **H4** additionally preserves a snapshot of the persisted data—if during training the application only generated source code via the PHP persistence API, then a new source file will only be trusted if it matches the last snapshot taken at **H4**.

B. Taint Tracking

The expansion events rely on propagation of taint from user input and authorized state changes. User input is tainted at HTTP input parameters and session values, while data loaded by the application is tainted in **H5** on the basis of $state_taint$. Function PROPAGATE-TAINT() in **H3** (line 25) transfers both colors of taint across assignments, compositional operations such as arithmetic and comparisons, arguments to function calls, and return values from functions. Hook **H3** also implicitly taints all assignments that occur during an expansion (line 22), since those assignments have as much of a causal relationship to the taint source as the branch itself. But ZENIDS does not implicitly taint assignments within trusted code, even if it occurs under a tainted branch, because those assignments must have already been made at some time prior to the expansion event—before taint was present on the predicate—indicating that the influence represented by the taint is not pivotal to that already-trusted branch decision.

C. Evolution Goals

This enhancement to ZENIDS aims to improve usability while maintaining the security goals of the basic implementation. In our evaluation we explore the following use cases:

UG2	Incorporate complex changes to control flow into the trusted profile without raising false alerts.
SG2	Continue to raise legitimate $untrusted_call$ alerts both during and after an expansion event.

VI. PERFORMANCE OPTIMIZATION

The overhead of branch evaluation in **H2** and taint tracking in **H3** (line 25) could make a naïve implementation of ZENIDS unusable in a high-traffic deployment. Even checking for the presence of taint in **H3** (line 3) can increase overhead by an order of magnitude. But since these expensive operations are only necessary in certain scenarios, it is safe for ZENIDS to elide them when conditions indicate that the operations will always nop. For maximum efficiency, ZENIDS implements three flavors of Opcode Execution Loop (Fig. 3), each taking only the actions necessary for its designated context:

1) **MonitorAll**: Propagates taint, evaluates all branch targets.
   - Reserved for profile expansion events (2,000 requests).
2) **MonitorCalls**: only evaluates call targets.
   - This is the default for monitoring untrusted requests.
3) **MonitorOff**: ignores everything.
   - Reserved for trusted users (negligible overhead).

Since PHP invokes the Opcode Execution Loop via function pointer, switching is simply a matter of replacing the pointer. The two monitoring modes are further optimized as follows:

MonitorCalls The first 4 lines of **H2** are elided by lazily grafting the set of safe call targets onto each PHP opcode. ZENIDS cannot extend the 32-byte opcode struct because interpreter performance relies on byte alignment, so instead it borrows the upper bits of a pointer field (which are unused in 64-bit Linux where the user-mode address space is much smaller than 2^{64}). Specifically, a pointer into the trusted profile is copied into a cache-friendly array, whose index is packed into the spare bits of the opcode. Line 5 of **H2** is further optimized for call sites having only one trusted target: instead of a trusted profile pointer, the array entry is a direct encoding of the singleton target, allowing for bitwise evaluation. To avoid expensive string comparisons, target opcode sequences are identified by a hashcode of the canonical name.

MonitorAll Since ZENIDS maintains taint in a hashtable, the accesses required for taint propagation could become expensive under rapid iteration of hook **H3** (Fig. 3). In addition, some PHP opcodes might require a costly callback from the interpreter because they affect complex data structures or are implemented using inline assembly. Both sources of overhead are alleviated by lazy taint propagation using a queue that is locally maintained by simple pointer arithmetic.

When the interpreter copies internal data structures (e.g., to expand a hashtable), taint must be tediously transferred to the new instance. ZENIDS flags each data structure that contains taint and elides this expensive transfer for untainted structures.

A. Synchronizing Evolution

For applications that store state in a database, the persisted $state_taint$ (Listing **H4**) is updated by database triggers, since it is non-trivial to determine from an SQL string what database values may change (we assume instrumenting the database engine is undesirable). But the query for $state_taint$ can be expensive relative to the total turnaround time of a very simple

HTTP request, even when our query uses a trivial prepared statement. Instead, ZENIDS stores the request id of the last privileged state change in a flat file and updates it from the database every 20 requests. While this delay makes it possible to raise up to 20 false alerts immediately after a state change, it is unlikely because a significant state change often involves multiple HTTP requests. For example, although the WordPress permalinks feature can be enabled with just one radio button selection, it takes at least 5 asynchronous HTTP requests to process that change, and only the last one enables the feature.

B. Performance Goals

The goals of these optimizations are straight-forward. Since this significantly increases the complexity of ZENIDS, we add a usability goal to limit development and maintenance costs:

UG3 Reduce overhead to less than 5% on our benchmark applications, as measured on real HTTP traffic.
UG4 Minimize the cost of developing and maintaining a usable implementation of ZENIDS.
SG3 Maintain the existing security.

VII. EXPERIMENTAL EVALUATION

We conduct several controlled and real-world experiments to demonstrate the ability of ZENIDS to meet our three security goals (**SG1**-**SG3**) and four usability goals (**UG1**-**UG4**). We begin with a controlled experiment to detect known exploits (**SG1**) without raising false positives (**UG1**). Section VII-B demonstrates similar results in the context of real PHP applications facing live Internet traffic. In Section VII-C the same experiments additionally show the accuracy (**UG2**) and safety (**SG2**) of the evolution feature. Performance optimizations were enabled for the experiments to show that those improvements do not compromise security (**SG3**). Section VII-D concludes with an evaluation of the ZENIDS runtime overhead (**UG3**) and a discussion of development cost (**UG4**).

A. Monitoring a Vulnerable Application

We demonstrate that ZENIDS detects publicly reported exploits by configuring a WordPress site with vulnerabilities targeted by the 10 most recent WordPress exploits published on `exploit-db.com` (as of June 18, 2016), excluding data-only attacks and unavailable modules. This medley, shown in Table II, shows that ZENIDS can detect a broad variety of exploit strategies (**SG1**). For authenticity we configured and exercised at least 30% of each module's high-level features. To train the trusted profile we invited common crawlers and manually browsed the site for just a few minutes using a range of mobile and desktop browsers. For modules having forms, we entered a variety of valid and invalid data and uploaded valid and invalid files. Then we enabled the ZENIDS monitor and continued using the site to evaluate our goals:

SG1 We invoked each exploit and observed that (a) ZENIDS raised the expected alert, and (b) the POC succeeded.
UG1 False positives only occurred when entering invalid form data—all other requests were trusted by ZENIDS.

TABLE II: ZENIDS raises an alert during attempts to exploit vulnerable plugins and themes in a WordPress site. These were the most recent 10 WordPress exploits from `exploit-db.com` as of June 18, 2016, excluding data-only attacks and unavailable modules.

EDB-ID	WordPress Module	Exploit Detected
39577	AB Test Plugin	✓
39552	Beauty & Clean Theme	✓
37754	Candidate Application Form Plugin	✓
39752	Ghost Plugin	✓
39969	Gravity Forms Plugin	✓
38861	Gwolle Guestbook Plugin	✓
39621	IMDb Profile Widget	✓
39883	Simple Backup Plugin	✓
37753	Simple Image Manipulator Plugin	✓
39891	WP Mobile Detector Plugin	✓

B. Monitoring Live Applications

To demonstrate the effectiveness and usability of ZENIDS in the real world, we recorded all public HTTP traffic for one year to three PHP applications hosted by our research lab:

1) **WordPress:** Our lab website, which has 10 pages and uses the "Attitude" theme (with no posts or comments).
2) **GitList:** The public repository viewer for software developed in our lab, based on the Symfony framework.
3) **DokuWiki:** A computer science class website containing wiki-text markup, document previews and file downloads.

We conducted this experiment offline by replaying the recorded HTTP traffic to a copy of the web server and also crawling the site with utility `wget`. Table III shows the results in terms of unique false positives and total false negatives. To further substantiate the usability of ZENIDS, Table IV shows these accuracy statistics under a range of training durations.

Exploit Detection (**SG1**): ZENIDS raised 38,076 true alerts on a diverse array of attacks targeting dozens of applications and using a broad range of exploit techniques. The majority of alerts were raised in WordPress because every unrecognized URL is directed to the WordPress permalink resolver. The false negatives represent two kinds of failed attacks:

- Typical threats that were safely handled by application code, e.g., invalid logins or unauthorized admin requests.
- Exploits that (a) target applications that we did not have installed, and (b) did nothing to affect the control flow.

One especially interesting attack abused the WordPress xmlrpc API to amplify brute-force password guessing. Not only was it a real threat to our installation, it is also an example of a legitimate API that ZENIDS does not trust—and rightly so, because RPC tools are irrelevant for our simple lab website.

TABLE III: Alerts raised by ZENIDS while monitoring our lab web server for 360 days. False negatives represent safely handled attacks such as invalid logins, or attacks on applications that we do not host.

	Requests		Intrusion Alerts				
			Total	False Positives		False Negatives	
	Total	Training		Unique	Rate	#	Rate
WordPress	248,813	595	36,693	3	.00001%	16679	.07%
GitList	1,629,407	298	1,744	1	<.000001%	0	0
DokuWiki	24,574	3,272	253	3	.0001%	0	0

These experiments not only show that ZENIDS detects a broad range of attacks, but also that it integrates effectively into a diverse set of applications representing an important cross-section of today's PHP landscape. These frameworks serve a large percentage of the world's HTTP traffic and support millions of websites ranging in significance from personal homepages to household names in the Alexa top 100.

Minimize False Alerts (`UG1`): ZENIDS only raised 507 false alerts (3 unique) against WordPress in the entire year. A broken link triggered 502 of them (2 unique) at the "guess permalink" function—one might argue that these are true alerts, since they do reflect an error in the site.

Improvements to ZENIDS could potentially eliminate all the false positives in DokuWiki. Half were caused by the addition of a tarball download to the wiki, which does not trigger an expansion event because new control flow occurs before the tainted tarball is loaded from the filesystem. ZENIDS could enable this expansion by tainting operands at untrusted branches and, after the request completes, checking backwards for privileged user influence. The remaining false positives were caused by crawlers reaching new pages, which could be avoided by blocking suspicious requests from crawlers.

We experienced just one false positive in over 1.5 million GitList requests despite training the trusted profile in just 2 hours—a total of 298 requests—highlighting the simplicity of our dynamic approach vs. a traditional static analysis. Our trusted profile for GitList covers 62 closures and several dynamically generated PHP files, along with 33 distinct magic methods reached from 54 call sites (excluding constructors and destructors), and 327 callbacks from PHP built-in functions. There were 25 callbacks to closures, which are especially challenging for static analysis, yet easily identified at runtime.

Our first two goals imply that, ideally, the duration of the training period should not significantly increase false positives (`UG1`) or false negatives (`SG1`). Table IV shows that although ZENIDS can be trained for a very short period of time, the results are similar for much longer training periods.

C. Evolution

Each of our three applications experienced one expansion event during the experiment. The largest event occurred in WordPress when the site administrator enabled permalinks (at request #53,310), which has the following effects on the site:

- Visitors may request any page by name. For example, the original "ugly" URL http://ourlab/?p=18 is now also reachable as http://ourlab/publications_page/.
- Requests for the original URL forms /p= or /page_id= are rewritten by WordPress to the permalink URL.
- Visitors subscribing to a comment feed can use the permalink form of the feed URL (which was requested by crawlers even though comments were disabled).

A smaller data expansion occurred in DokuWiki after a change to the layout of the start page and the wiki menu bar. In GitList, a code expansion incorporated new view templates into the trusted profile as they were dynamically generated.

TABLE IV: The duration of the training period has minimal impact on the accuracy of ZENIDS alerts. False positives in DokuWiki could potentially be avoided by improvements to ZENIDS. False negatives represent attacks that failed to have any effect on control flow.

	Training Requests	Intrusion Alerts				
		Total	False Positives (unique)		False Negatives	
			#	Rate	#	Rate
WordPress	595	36,693	507 (3)	.002% (.00001%)	16,679	.07%
	1,188	35,826	507 (3)	.002% (.00001%)	17,546	.07%
	2,070	35,767	507 (3)	.002% (.00001%)	17,605	.07%
	4,128	35,727	507 (3)	.002% (.00001%)	17,645	.07%
GitList	298	1,744	6 (1)	$<$ .000001%	0	0
	862	1,744	6 (1)	$<$ .000001%	0	0
	12,010	1,744	6 (1)	$<$ .000001%	0	0
	72,068	1,744	6 (1)	$<$ .000001%	0	0
DokuWiki	1,462	364	224 (55)	.009% (.002%)	0	0
	3,272	253	101 (3)	.004% (.0001%)	0	0
	4,133	253	101 (3)	.004% (.0001%)	0	0
	5,559	217	65 (2)	.003% (.0001%)	0	0

Safe Profile Expansion (`SG2`): Learning to trust the new WordPress permalinks feature was the most risky of the three expansion events, because the feature involves directing all unrecognized URLs to WordPress for pattern matching. Since this includes any exploit attempts, a weak implementation of ZENIDS might mistakenly trust malicious control flow as part of the expansion event. But manual analysis confirms that all 144 newly trusted calls were strictly necessary to support the permalinks feature. In fact, ZENIDS raised 114 intrusion alerts during the expansion, including 9 attempts at known WordPress plugin exploits (CVE-2015-1579 and [9], [10], [11]), 2 invalid login attempts, 3 attempts to register new users, 6 requests for disabled features, and 36 unauthorized requests for administrator pages. Following the expansion, ZENIDS continued to raise alerts on thousands of malicious requests, many of which used a valid WordPress permalink URL form.

The GitList expansion incorporated several new views into the trusted profile, each having more than 100 SLOC. In DokuWiki the expansion added 145 new SLOC. We did not experience enough attacks targeting GitList or DokuWiki to make an empirical case for the safety of those expansions, but manual analysis confirms that every added call was strictly necessary for ZENIDS to trust the newly enabled features.

The Twig template engine in GitList conforms to all three characteristics of a disciplined code generator, indicating that ZENIDS successfully detected the three corresponding criteria in hook `H4` when the new views were generated.

Sufficient Profile Expansion (`UG2`): No false positives occurred in any of the three features that initiated expansion of the trusted profile. The WordPress permalinks feature supports many URL variations and a complex resolution mechanism, and although it was heavily exercised for the 9+ months during which permalinks were enabled, ZENIDS trusted all the new code paths on the basis of the 2,000 request expansion period. The new GitList views and DokuWiki menu layout also activated new URL forms with additional URL resolution, and these were fully enabled after the expansions completed.

D. *Performance* (**UG3**)

We evaluated the performance of ZENIDS by replaying a contiguous segment of the recorded HTTP traffic from the experiment in Section VII-B. To avoid bias we selected a segment having a representative frequency of expansion events (though none incorporated new control flow). The web server is an Intel Xeon E3-1246 v3 with 32GB RAM and a solid state drive, configured with a LAMP stack running on Ubuntu 14.04: Apache 2.4.7, MySQL 5.5.49, and PHP 7.1.0 alpha 3.

To show that ZENIDS performs well in a real deployment scenario, we configured optimizations that would typically be used for high-traffic PHP websites. For example, our deployment uses the popular `opcache` extension, which alleviates compilation overhead by caching the opcode sequences. We also chose the latest stable build of PHP which includes significant core optimizations such as global register allocation for the script instruction pointer and frame pointer. We enable all optimizations in `opcache` and use gcc optimization flags `-O3 -march=native` to obtain peak performance for the baseline (vanilla) configuration of the PHP interpreter.

After configuring ZENIDS with the same trusted profile that we reported in Section VII-B, we replayed the HTTP requests synchronously to isolate the PHP interpreter's execution time from process and thread scheduling factors. Table V shows the overhead is less than 5% for all our applications.

Instrumentation Overheads (**UG4**) The cost of developing and maintaining ZENIDS is an important factor in its overall performance as a practical security tool. Although there is significant effort involved, this burden becomes progressively lighter as the user base grows, since the work only needs to be done once for each version of the PHP interpreter.

The ZENIDS PHP extension consists of 20KLOC of code, and we additionally instrumented 8 hooks for a total of 317 lines of code in the PHP interpreter source. For applications that store state in a database, ZENIDS requires a database schema to contain the *state_taint* that is used by hooks **H4** and **H5** to support data expansions. The schema consists of two small tables, plus one trigger per application table.

For PHP applications having an authentication scheme, the login function must be instrumented with callbacks to `set_user_level($level)`, which is provided by the ZENIDS extension as a PHP built-in function. The `$level` argument is an integer indicating the new authentication level, for example in WordPress we use the application's `role_id`. Placing the callbacks was simple in both WordPress and Doku: for each application, we inserted three callbacks in one source file immediately following a log statement indicating successful login or logout (GitList has no authentication).

E. *Verifiability*

The ZENIDS prototype is open-source and can be found at http://www.github.com/uci-plrg/zen-ids. Since the data used in our experiments contains private user information, we are not able to publicly release it. However, our repository provides instructions for conducting similar experiments in ZENIDS.

TABLE V: Runtime overhead of ZENIDS vs. an optimized vanilla LAMP stack, measured as the geometric mean of 10 runs.

	WordPress	GitList	DokuWiki
Runtime Overhead	4.1%	4.6%	4.5%

VIII. RELATED WORK

Many recent approaches to the problem of intrusion detection are successful against specific categories of vulnerabilities, such as cross-site scripting (XSS) or SQL injection (SQLi), but none of them has proven effective against remote code execution (RCE) exploits. For example, the WordPress plugin MuteScreamer [12] was once commonly used to protect WordPress sites, but it only supports a manual blacklist of request patterns, making it vulnerable to mimicry and incapable of defeating a zero-day attack. The comprehensive coverage of the trusted profile makes it possible for ZENIDS to surpass some limitations of these otherwise successful techniques.

a) Input Filtering: For applications having a relatively systematic public interface, exploits can be detected with high accuracy by observing patterns in user input. Commonly deployed tools are Snort [13] and Apache ModSecurity [14], which block known attacks based on a manually maintained blacklist. But for today's complex and highly dynamic PHP applications, this approach performs poorly because (a) a single vulnerability can be compromised using distinct crafted inputs, (b) the frequency of blacklist updates would increase by several orders of magnitude (for example, wordpress.org currently offers over 46,000 plugins, most of which are continually in development), and (c) blacklist approaches cannot defeat zero-day attacks, yet new exploits against WordPress alone are reported almost daily.

A variant known as *anomaly detection* relies instead on a whitelist of normal application behaviors, but the whitelist can be difficult to construct. One approach [15] requires an expert to manually define a set of policies, which also must be maintained to accommodate ongoing customization, configuration changes, and even application state changes. Several alternative techniques formulate the whitelist in simpler terms—for example, n-grams of packet bytes [16], [17], or properties of HTTP request attributes such as string length, characters sets, token sets [18], [19]—and learn the whitelist by observing a set of known-normal requests (obtained by manual analysis). Many of these approaches are subject to mimicry or evasion tactics because the whitelist is only indirectly related to program semantics. Another weakness is that the training time is typically quite long—from 8% up to 50% of the experiment's reported request set in successful techniques. Recent investigation into these approaches no longer mentions web applications, suggesting that the increasingly dynamic interaction between browser and server inhibits convergence of the whitelist. Pure machine learning approaches have mediocre results on artificial benchmarks such as the DARPA datasets [20], and have rarely been used in practice, where they typically perform much worse [21].

b) Introspective Anomaly Detection: A different n-gram approach observes the internal execution of the application on the server and learns a whitelist of short system call sequences [22]. This approach turns out to be highly susceptible to mimicry—isolating the sequence of system calls by name (only) yields a vague representation of the application that allows the attacker far too much flexibility. Similar approaches based on finite state automata [23], [24] were difficult to apply in practice (no experiments were reported).

c) Static Analysis: Instead of attempting to detect exploits at runtime, several techniques focus on finding vulnerabilities offline so the application can be secured by simply patching. Conventional static analysis has been unable to find many important attack vectors in stateful, session-based applications, so [25] uses an SMT solver along with advanced formulation of constraints on string dataflow and program logic to find several new RCE exploits in large PHP applications. While this approach has some success, the authors admit many vulnerabilities cannot be found this way. For example, the implementation has limitations at dynamic constructs such as variable array indices, which are very common in PHP—in the version of phpMyAdmin reported in the experiment, 25% of all array accesses use a variable index.

Other approaches are primarily effective against XSS and/or SQLi vulnerabilities. To improve coverage of the static analysis, [26] partitions it into 3 levels of granularity: intra-block, intra-procedural and inter-procedural. Both WebSSARI [27] and Saner [28] combine static and dynamic analysis to detect missing authentication checks; WebSSARI automatically patches the code, while Saner verifies each result by dynamically generating an exploit. Pixy [29] employs taint tracking to find vulnerable handling of string variables. The authors of all these techniques suggest their approaches may also work well for other kinds of exploits, including RCE—but they also admit limitations at dynamic statements such as `include` with a variable operand, which are a primary targets of RCE exploits. Another limitation of these approaches is that they disregard client-side JavaScript, making it difficult for them to find all the application entry points.

d) Dynamic Analysis: Exploit detection techniques that share our dynamic approach are highly specialized to particular kinds of vulnerabilities other than RCE. ScriptGard [30] instruments both the client and server to dynamically patch faulty authentication code. Noncespaces [31] randomizes sensitive values to prevent client-side tampering. Diglossia [32] and SQLCheck [33] employ efficient taint tracking to detect SQLi. To our knowledge, ZENIDS is the first such dynamic instrumentation approach to focus specifically on RCE exploits.

A. Defenses for Other Platforms

Some important defenses that have been developed for other platforms cannot be directly applied to RCE exploits in PHP, yet are still relevant to the ZENIDS approach.

A large body of CFI research has focused on reducing overheads [34], [35], [36], [37] through minimizing dynamic checks. Unfortunately, researchers found these simplifications open the application to attacks [5], [38], [39], [40], [41]. In response, researchers have developed newer, more restrictive approaches [42], [43] with higher overheads. Some compiler-based approaches achieve low overhead by protecting only forward control flow edges [44], [45]. OpaqueCFI [46] combines randomization with simple range checks, which weakens the constraints to improve runtime efficiency, but in a way the adversary cannot easily reason about. Several other CFI approaches target specific attacks [47], [48], [49], [3], [2], such as ROP, but may not be robust to new types of attacks. Recent work has shown that many of these approaches are vulnerable to attack variants [4], [6]. ZENIDS is able to maintain a usable level of performance while comprehensively monitoring inter-procedural control flow at runtime against the trusted profile.

Software diversification [50], [51] introduces uncertainty for the attacker by generating a physically distinct yet functionally identical version of a compiled binary for each user, but is not able to deter exploits that manipulate source-level semantics.

The logging framework BlackBox [52] employs learning to isolate potential exploits of COTS binaries, building a whitelist by observing real executions. ZENIDS is similarly able to isolate anomalies in the execution of dynamically specified branch targets, dynamically generated code strings, and PHP files that did not exist at the time of initial deployment.

The Android permission model is coarse grained, making it possible for adversaries to access resources that are allowed by the permissions but are irrelevant to the application. In [53], an offline dynamic analysis mines the application for normal resource usage, and a runtime sandbox limits the application to those refined permissions. This approach differs from ZENIDS in that it focuses on constraining resource abuse.

Clearview [54] uses learning to patch software errors. A key difference is that Clearview focuses on generating repairs, relying on external mechanisms for error detection, whereas ZENIDS focuses directly on detecting attacks.

IX. Conclusion

We presented ZENIDS, a tool for identifying malicious activity in PHP applications. During a short learning phase, ZENIDS observes a set of normal executions and encodes those application behaviors in a trusted profile. ZENIDS then raises an intrusion alert when it observes untrusted application behaviors. ZENIDS employs taint-tracking to enable new application behaviors that result from trusted state changes, including privileged user activity and application code generators. Our results show that ZENIDS can protect real world PHP applications from known exploits and can detect a number of real world attacks initiated by actual adversaries.

X. Acknowledgments

We thank Prateek Saxena, Per Larsen, Konrad Jamrozik, Ùlfar Erlingsson, Fabian Gruber, Cyril Six, the PLDI Student Research Competition committee, and the anonymous reviewers for their helpful comments. This work was supported by the National Science Foundation under grants CCF-0846195, CCF-1217854, CNS-1228995, and CCF-1319786.

REFERENCES

[1] C. Viviani, "WordPress Download Manager 2.7.4 - remote code execution vulnerability," https://www.exploit-db.com/exploits/35533/, 2015.

[2] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *SEC*, 2013.

[3] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attack," in *NDSS*, 2014.

[4] E. Göktas, E. Athanasopoulos, M. Polychroniakis, H. Bos, and G. Portokalidis, "Size does matter - why using gadget chain length to prevent code-reuse attacks is hard," in *USENIX Security*, 2014.

[5] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX Security*, 2014.

[6] N. Carlini and D. Wagner, "ROP is still dangerous: Breaking modern defenses," in *USENIX Security*, 2014.

[7] S. Esser, "Utilizing code reuse or return oriented programming in PHP applications," in *Black Hat, USA*, 2010.

[8] J. Dahse, N. Krein, and T. Holz, "Code reuse attacks in php: Automated pop chain generation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, 2014.

[9] CXSecurity, "WordPress revslider arbitrary file upload, download & cross site scripting," https://cxsecurity.com/issue/WLB-2015060136, 2015.

[10] F. Ng, "A story of cyber attack and incident response," https://www.linkedin.com/pulse/story-cyber-attack-incident-response-freeman-ng, 2015.

[11] CXSecurity, "Wordpress formcraft plugin file upload vulnerability," https://cxsecurity.com/issue/WLB-2016020136, 2016.

[12] WordPress, "Mute screamer—wordpress plugins," https://wordpress.org/plugins/mute-screamer/.

[13] snort.org, "Snort - network intrusion detection & prevention system," https://www.snort.org/.

[14] Trustwave SpiderLabs, "ModSecurity: Open source web application firewall," https://www.modsecurity.org/.

[15] D. Scott and R. Sharp, "Abstracting application-level web security," in *Proceedings of the 11th International Conference on World Wide Web*, ser. WWW '02, 2002.

[16] K. Wang, G. Cretu, and S. J. Stolfo, "Anomalous payload-based worm detection and signature generation," in *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'05.

[17] K. Wang, J. J. Parekh, and S. J. Stolfo, "Anagram: A content anomaly detector resistant to mimicry attack," in *Proceedings of the 9th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'06, 2006.

[18] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03, 2003.

[19] W. Robertson, G. Vigna, C. Kruegel, and R. A. Kemmerer, "Using generalization and characterization techniques in the anomaly-based detection of web attacks," in *In Proceedings of the 13 th Symposium on Network and Distributed System Security (NDSS)*, 2006.

[20] M. L. Laboratories, "DARPA intrusion detection data sets," https://www.ll.mit.edu/ideval/data/.

[21] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10, 2010.

[22] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, ser. SP '96, 1996.

[23] F. B. Schneider, "Enforceable security policies," vol. 3, no. 1, Feb. 2000, pp. 30–50.

[24] G. Wassermann and Z. 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*, ser. PLDI '07.

[25] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013.

[26] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06, 2006.

[27] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th International Conference on World Wide Web*, ser. WWW '04, 2004.

[28] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. SP '08, 2008.

[29] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, ser. SP '06, 2006.

[30] P. Saxena, D. Molnar, and B. Livshits, "SCRIPTGARD: Automatic context-sensitive sanitization for large-scale legacy web applications," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11, 2011.

[31] M. Van Gundy and H. Chen, "Noncespaces: Using randomization to defeat cross-site scripting attacks," vol. 31, no. 4. Elsevier Advanced Technology Publications, Jun. 2012, pp. 612–628.

[32] S. Son, K. S. McKinley, and V. Shmatikov, "Diglossia: Detecting code injection attacks with precision and efficiency," in *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, 2013.

[33] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06, 2006.

[34] C. Zhang, T. Wei, Z. Chen, L. Duan, S. McCamant, L. Szekeres, D. Song, and W. Zou, "Practical control flow integrity & randomization for binary executables," in *Proceedings of IEEE Symposium on Security and Privacy*, 2013.

[35] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *USENIX Security Symposium*, 2013.

[36] U. Erlingsson, S. Valley, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI '06, 2006.

[37] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nrnberger, and A. reza Sadeghi, "MoCFI: A framework to mitigate control-flow attacks on smartphones," in *Proceedings of the Network and Distributed System Security Symposium*, 2012.

[38] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *CCS*, 2015.

[39] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *S&P*, 2014.

[40] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX Security*, 2015.

[41] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *CCS*, 2015.

[42] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: Cryptographically enforced control flow integrity," in *CCS*, 2015.

[43] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive CFI," in *CCS*, 2015.

[44] D. Jang, Z. Tatlock, and S. Lerner, "SafeDispatch: Securing C++ virtual calls from memory corruption attacks," in *Proceedings of the 2014 Network and Distributed System Security Symposium*, ser. NDSS '14, 2014.

[45] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Úlfar Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *USENIX Security*, 2014.

[46] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity," in *NDSS*, 2015.

[47] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11, 2011.

[48] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, "Drop: Detecting return-oriented programming malicious code," in *Proceedings of the 5th*

International Conference on Information Systems Security, ser. ICISS '09, 2009.
[49] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12, 2012.
[50] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *S&P*, 2014.
[51] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Librando: Transparent code randomization for just-in-time compilers," in *CCS*, 2013.
[52] B. Hawkins, B. Demsky, and M. B. Taylor, "BlackBox: Lightweight security monitoring for COTS binaries," in *Proceedings of the 2016 Symposium on Code Generation and Optimization*, ser. CGO '16, 2016.
[53] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *Proceedings of the 2016 International Conference on Software Engineering*, 2016.
[54] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *SOSP*, 2009.