Visualizing Web Application Execution Logs to Improve Software Security Defect Localization

Matthew A. Puentes, Yunsen Lei, Noëlle Rakotondravony, Lane T. Harrison, Craig A. Shue
Computer Science Department, Worcester Polytechnic Institute
{mapuentes, ylei3, ntrakotondravony, ltharrison, cshue}@wpi.edu

***Abstract*—Interactive web-based applications play an important role for both service providers and consumers. However, web applications tend to be complex, produce high-volume data, and are often ripe for attack. Attack analysis and remediation are complicated by adversary obfuscation and the difficulty in assembling and analyzing logs.**

In this work, we explore the web application analysis task through log file fusion, distillation, and visualization. Our approach consists of visualizing the logs of web and database traffic with detailed function execution traces. We establish causal links between events and their associated behaviors. We evaluate the effectiveness of this process using data volume reduction statistics, user interaction models, and usage scenarios. Across a set of scenarios, we find that our techniques can filter at least 97.5% of log data and reduce analysis time by 93–96%.

I. Introduction

Web-based applications are important and numerous. As one example, the WordPress content management system reportedly is used by over 41% of websites [22] and allows non-experts to create and update their sites in an easy-to-use fashion. However, these tools commonly rely upon a complex software stack, with multiple servers interacting to retrieve or store records, serve content, and to perform authentication. Adversaries recognize that web-based applications are valuable assets and attack them, leading to regular breaches and significant financial losses [1]. Whenever an adversary successfully exploits a vulnerability, system defenders have an opportunity to learn about the defect that makes the software vulnerable. Since adversaries may wish to continue leveraging the vulnerability, they may try to obscure the interaction with cover traffic, superfluous activity, or by corrupting records.

For defenders to pinpoint the vulnerabilities associated with successful exploits, they must first meet multiple requirements. First, they must have a mechanism to determine when an exploit occurred. This can be a tool such as an intrusion detection system or via manual labeling. Second, they must have a sufficiently detailed logging system in place that records the relevant actions in an attack in a manner that the attacker cannot corrupt [3]. Finally, they need to perform incident analysis by correlating records from distributed sensors while eliminating extraneous information.

Prior efforts have led to progress with the first two requirements. Such works have focused on techniques to isolate attacks on web infrastructures and to detect access violations [20], [14]. These systems separate users into lightweight, isolated server instances. By deploying network middleboxes between clients and server instance pools, concurrent requests from different users are demultiplexed. This per-user isolation also naturally collects logging data on a per-user basis. Our work builds upon such infrastructure to enable accurate linking between events recorded at different vantage points.

The remaining task, incident analysis, requires a scarce resource: developers' time. Prior work estimates between 35-50% of developers' time is spent performing debugging tasks [4]. With web applications, the volume of log data can be large. Furthermore, with adversary efforts to hinder comprehension, it can be difficult to unravel an attack. The current best practice is a manual (and tedious) evaluation of log files and note-taking to understand how events relate. Analysts need a visualization platform that can quickly communicate the causal links and flows between events to allow them to construct a mental model of the issue.

In this paper, we focus on the incident analysis task and ways to improve it. We ask the following research questions: *To what extent can detailed logs help developers identify the defect associated with a web application vulnerability when responding to incidents? What log data is useful in such analysis? To what extent can visualizations of function call stacks simplify analysis and what impact would it have on the time required for defect identification?*

In exploring these questions, we contribute the following:

- **Fuse and Prune Data Logs:** We implement a log fusion and distillation function for PHP web applications. We leverage existing server virtualization architectures that enforce user isolation and flag violations to trigger our log data collection process. We extract interactions related to incidents and prune unrelated activity. Using heuristics, we identify the functions most likely to be responsible for a security violation.
- **Create a Visualization and Analysis System:** We create PEGASUS (Powerful, Expressive, Graphical Analyzer for Single-User Servers), a web-based visualization system that creates a graph of function calls in PHP scripts and provides multiple views that link records from proxies that are between 1) the web client and web server and 2) the web server and database server. PEGASUS expedites code navigation, prunes irrelevant code, displays incident function call parameters, and enables developer annotation of defects. Results suggest that our approach can filter up to 97.5% of log data while still presenting the defect and its context.

- **Evaluate the System with Usability Models:** Since analyst time is our critical resource, we use Keystroke-level Model (KLM) [13] to model actions analysts perform when using our system. We record the actions an analyst performs during a debugging task for localizing vulnerabilities in plug-ins associated with the WordPress content management system. We compare traditional debugging against the PEGASUS system and found that our tools can reduce required analyst time in our tested scenarios by 93–96%.

II. Background and Related Work

Understanding and identifying the cause of a security vulnerability combines multiple sub-fields of research. Existing work studied the pattern of software code commits [5] and application contextual information [9] to provide high-level guidance on understanding software vulnerabilities. Since our work aims to localize the direct cause of a security vulnerability at the source code level, we consider more related work in causal analysis, program behavior analysis, isolation of untrustworthy execution, and visualizations of execution traces of applications.

A. Web Data Provenance

At a high level, PEGASUS parses and builds linkage using data collected from multiple sources. The goal of such a process is essential to answer the questions about what execution leads to the security incident. Previous work proposes the concept of a Network Provenance Function (NPF) [2]. By deploying network middleboxes between components of a web stack and using domain knowledge to parse the protocol data, the linkage between an HTTP request and resource access can be established. Such an approach enables causal reasoning to locate which request leads to an exploit. Other work [11] extends the basic concept of NPF and uses server application logs to add extra context to facilitate the understanding of server execution.

These efforts allow a security incident's cause to be traced at the process or request level. Our approach offers greater localization precision by capturing the web application's dynamic behavior. Further, we visualize the pruned log data to allow analysts to explore and localize the vulnerable code.

B. Visualizing Execution Traces

A program's execution trace reveals important information about the program's behavior and prior research has examined how to visualize it. Earlier efforts focused on object-oriented programs [8], [7]. They instrument and visualize each object's dynamics and invocation of methods. This allows an analyst to gain a general understanding of a program. In addition to program comprehension, trace visualization is also used in software debugging. Mehnerä et al. proposed JaVis [15] to help with deadlock detection. JaVis instruments the entry and exit points in a synchronized method and uses a *collaboration diagram* to visualize each thread's execution traces. Analysts can identify deadlocks by finding cyclic dependencies in the resulting graph. To quantitatively evaluate the usefulness of an execution trace visualization, Cornelissen et al. [6] designed and conducted a series of controlled experiments where a participant was asked specific questions related to typical tasks in program comprehension. The answer's correctness and the task completion time are compared between users with or without the visualization tool.

Unlike prior works, PEGASUS visualizes and fuses execution data from web applications to identify security defects. Specifically, PEGASUS uses a successful attack as an illustrative example of the defect and helps an analyst identify the defective source code that enables the attack.

C. Attack Containment via Isolation Mechanisms

Multiple research efforts have explored mechanisms to isolate untrustworthy web-based execution environments. Recognizing the complexity of modern web applications, the goal in such research is to limit and contain security breaches.

In their work on CLAMP, Parno et al. [20] used Virtual Machines (VMs) to protect certain web applications. CLAMP directs each web client to a web server running in a separate VM and proxies database queries through a middlebox that is designed to enforce access privileges. Authors also note that the overheads of running multiple VMs require further innovation to enable practical deployment. In a later work, Lanson [14] proposed the Single-use Server (SuS) that uses lightweight containers to isolate web server processes, rather than VMs. This approach incorporates CLAMP's middlebox techniques while enabling better performance and scalability.

We leverage the logging infrastructure enabled by SuS as the foundation for our system, PEGASUS. In particular, we use the SuS model's isolation and middleboxes to log each web client's actions independently of other users.

III. Design: Helping Analysts Find Defects

In designing PEGASUS we adopt processes from Munzner's Nested Model [19]. Specifically, we characterize the domain, data, and data filtering processes for a web application vulnerability context, develop analyst's common actions IV related to understanding and remediation.

Web application codes are designed to be easily developed and deployed. Many applications serve users with varying roles, from anonymous (and unprivileged) users to site administrators. As a result, a web application is typically granted expansive permissions when interacting with backend resources. This allows the application to perform tasks on behalf of each type of user that is authenticated. However, this approach also makes the application responsible for determining when to use those permissions appropriately. If an adversary is able to find a defect in the application server, the adversary may cause the server to misuse its privileges. This is called a "confused deputy" attack [10]. SQL injection attacks are an example of confused deputy attacks and are consistently found to be a top security concern for web applications [21].

Both CLAMP [20] and SuS [14] use proxies to restrict queries to backend resources, such as SQL databases. By

creating a unique server instance for each web client, these architectures can determine the permissions associated with each user and enforce those in the proxy servers. In the SuS architecture, such an access violation causes the system to suspend the associated web server container, preserve the logs associated with the interaction, and alert the administrator of the system. While CLAMP and SuS play an important containment role and treat the symptoms of a vulnerability, they do not reveal the defect's cause.

With PEGASUS, we recognize that human analyst time is a precious resource and is key to remedying security vulnerabilities. We explore the context needed for analysts to understand the problem, the symptoms, and the events that precede the security violation. In this Section, we start by providing a concrete example attack. We then describe the SuS foundation and the PEGASUS components.

A. Example: WordPress Privilege Escalation

Determining the cause of a web application vulnerability can be challenging. We use a real-world example to illustrate the common obstacles faced by analysts in incident response.

Our example vulnerability, CVE-2018-19207 [18], is an improper access control defect in a WordPress plugin. The defect allows any unauthenticated user to alter WordPress global settings, which normally requires administrator-level privileges. In exploiting the defect, an attacker enables the registration of a new administrator-level account and creates such an account for their use. This attack only requires two HTTP requests with specifically tailored payloads. The only persistent indication of the attack for a site manager is the existence of another administrator account.

If the defender had detailed logging enabled for the site, they might be able to find symptions of the attack. The first symptom would appear in the query log for WordPress's database. That log would contain updates to the `user_can_register` and `default_role` values in the `wp_option` table. The second symptom would appear in the web server request log with two HTTP `POST` requests to the `/admin-ajax.php` script with the following payload:

```
payload 1={"option": "default_role",
           "value": "administrator"}
payload 2={"option": "users_can_register",
           "value": "1"}
```

Based on the similarity of strings between the HTTP and database log entries, the analyst may hypothesize that there is a causal link between the given HTTP request and the SQL action. To validate the hypothesis, the analyst could then explore the `/admin-ajax.php` script and its functions to determine if the `POST` request could cause the bad SQL query.

While this hypothesis creation and data exploration is theoretically straightforward, it might be daunting in practice. The logs of these operations grow rapidly in production websites and often involve concurrent requests, potentially across distinct users. A given SQL interaction may have hundreds or thousands of HTTP requests immediately preceding it. When multiplexed at a shared web server, it can be difficult to determine which PHP process caused a given SQL query and which client invoked that PHP script. Further, the attack payload may transform during the script's execution, so the string similarity heuristic may fail, complicating the formation of an initial hypothesis.

One of the key tasks for the PEGASUS system is to distill logs and present the analysts with the relevant causal chain of events, allowing them to skip the hypothesis creation step and jump to data exploration. With the data exploration task, we focus on providing the structure and relationship of functions as well as the specific instance of variables in an attack workflow. We now describe the key operations in PEGASUS and the components that perform each of them.

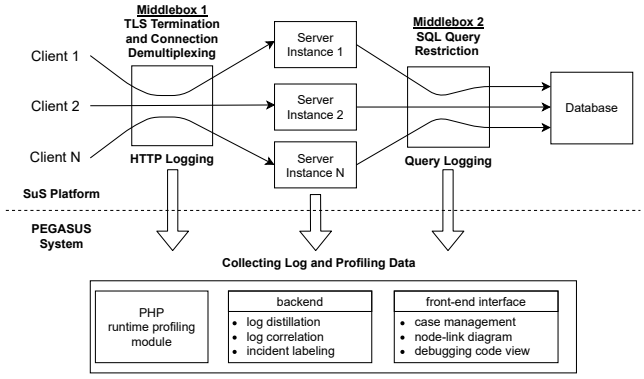

Fig. 1: The relationship between SuS and PEGASUS

B. Leveraging the SuS Model's Logging Infrastructure

We obtain a copy of the SuS system and build PEGASUS to leverage its container orchestration, client separation, and alerting functions. We consume SuS logs in PEGASUS.

Figure 1 shows the relationship between the SuS infrastructure and our PEGASUS system. While web servers are usually highly multiplexed, potentially serving thousands of simultaneous users, the SuS model instead has a unique web server instance that includes a separate server application and PHP runtime for each client. As Figure 1 (top) shows, each client is directed through a middlebox that acts as a TLS endpoint. That middlebox decrypts the communication, logs it, and directs it to the container with the appropriate web server instance. A second middlebox logs communication between the servers, enforces permissions, and acts as the detection component to alert when an access error is observed from the database server. It further notifies a coordinator system to suspend the container involved in the violation.

C. Function Call Logging in PHP

SuS's middlebox provides log data about the executions across the server component, but that base functionality does not capture the dynamic execution of the web application. To gain insight into the nature of the vulnerability, analysts need to know the code executed on the web server and the values associated with function call parameters.

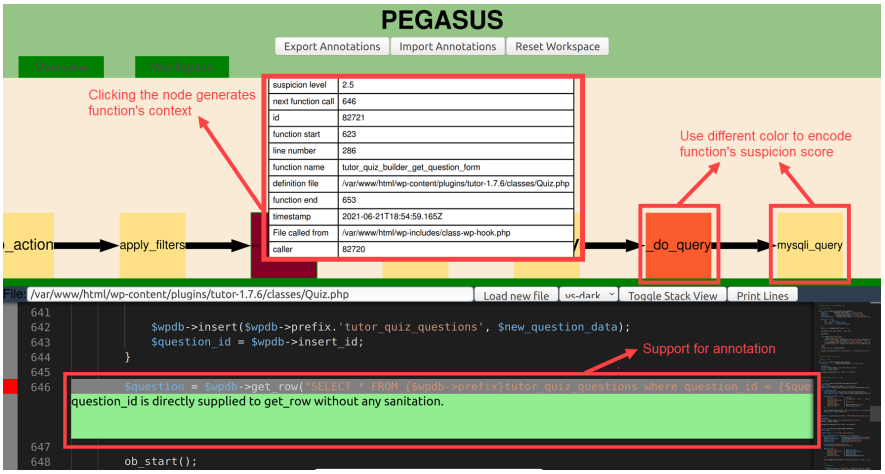

Fig. 2: Annotated screenshot of PEGASUS debugging a CVE

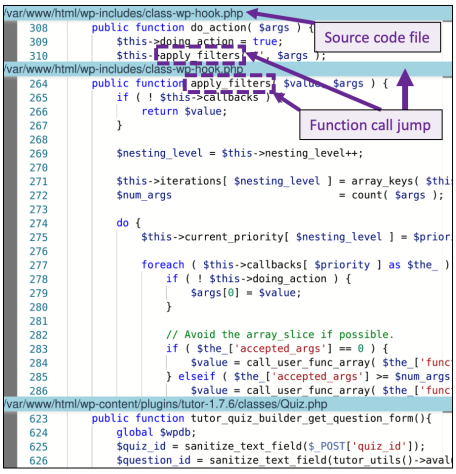

Fig. 3: Stacked code view mode

With PEGASUS, we augment the SuS data with an in-container application profiling module that logs execution of each PHP function. The module records parameters with each function invocation along with the call site (i.e., the location in which the calling function invoked the function). This module interacts with PHP's execution engine and writes to a location monitored by the PEGASUS backend. When a permission violation occurs, SuS will immediately suspend the web server container. This halts the system while some actions are still in-progress. This partial data represents the portion of the "call stack" that represents the currently-running function and its ancestors. An analyst can use this call stack to better understand how HTTP requests are processed by the web application. A visualization of this call stack can help an analyst navigate through the functions called by the attack.

D. Log Data Fusion and Distillation

To help an analyst understand a defect, we fuse disparate data sources while pruning distracting information, such as unrelated interactions or code paths that are not executed during an attack. The SuS system provides one inherent source of pruning: each log is maintained on a per-web-client basis and each client is fully isolated by containerization. When the SuS system flags an incident, it constructs incident logs only for traffic for the associated client. This is particularly important for high-traffic websites with significant amounts of unrelated traffic.

We then fuse information related to each web client. The user's requests, web application's function execution and database query depict an incident at different phases. Therefore, linking entries in these different logs helps reconstruct the whole incident. It identifies which user's requests cause functions to execute and trigger database queries. We start the linking process at the end: we identify the SQL query that caused the permission violation and link it to the appropriate node in the PHP call stack trace. To do so, we use timestamps to identify the PHP functions associated with SQL queries that immediately precede the SQL interaction. We then inspect each function's parameters and compare their strings with the SQL query to find a match.

Once we have the PHP function that issued the SQL query, we traverse its call stack. We examine the call site for each function call, allowing us to find the calling node and traverse the graph. Ultimately, we reach the root node associated with the script invoked by the web client. We again use timestamps and string matching between the PHP call stack and the HTTP proxy's logs to find the web request associated with that function invocation.

This process produces a complete causal chain of events.

E. Combining Node-Link Diagrams with Source Code

One of the goals of the visualization in PEGASUS is to empower analysts to quickly understand the flow of events, extract relevant details from the attacker's exploitation of the defect, isolate the faulty code, and annotate the issue. The process to actually correct the source code is a separate development task and is beyond the scope of this work.

In our design, we explore a directed node-link diagram. It uses nodes to represent log entries and edges to represent causal invocations between entries, such as function calls from parent to child functions. This visualization may be intuitive to analysts with experience using flow charts. The top part of Figure 2 shows the visualization in our PEGASUS tool.

The node-link visualization also provides natural interaction points for contextual information and navigation features. When an analyst clicks an edge, we overlay a table of parameters that the parent provides to the child function. When an analyst clicks a node, we signal the nearest code-view widget to scroll to that function.

Our code-view widget shows the source code of the software. PEGASUS allows panes to be arranged in a side-by-side fashion, allowing an analyst to use the Node-Link diagram for navigating and a source code viewer to explore the event.

When a user selects a node in the Node-Link diagram, PEGASUS displays the relevant code for that function in a code view widget. Importantly, unlike a traditional code viewer, the call stack logs allow us to know which lines of code

were executed in the trace. This allows us to sandwich relevant executed code regions, allowing a developer to visually see jumps between functions and eliminating distractions such as the trailing portion of functions that are not executed before the error occurs. Figure 3 shows the stacked code view that allows the analyst to see the lines of code that are invoked, in order. This allows developers to trace even complicated function calls, such as object member functions or when variables must be dereferenced to determine the invoked function (e.g., the transition between code segments two and three in Figure 3).

F. Support for Heuristics to Localize Vulnerabilities

Since call stacks include all the code that is executed before an exploit is detected, they necessarily include the software defect. This is true even if that defect is by omission, such as a failure to sanitize variables or check user permissions. Since call stacks can involve dozens of functions, a detailed analysis of the functions involved may still be arduous.

We designed heuristics that assigns a suspicion score to each function in a call stack. In these traces, each function is in a "caller" and "callee" relationship. Accordingly, heuristics can leverage these relationships where needed.

The first heuristic applies string matching between the network payload and the parameters to PHP functions. If a function directly consumes HTTP payload, we assign it a higher suspicion score since it can be adversary-influenced. The second heuristic assigns higher suspicion scores to files from source code directories associated with plugins or privileged code, since such regions are often implicated in web application attacks. Our final heuristics assigns lower suspicion scores to frequently-used functions (like helper functions) and higher suspicion to relatively rare functions. Since defects are more likely to be quickly detected in common traces, we decrease the suspicion score associated with those functions.

As part of our evaluation, we test these heuristics across a set of real-world vulnerabilities. While these heuristics are simple and straightforward, they appear to hold promise. In addition, they are modular and PEGASUS users can tailor them to the analyzed web application.

IV. Implementation

We implement the PEGASUS visualization component using TypeScript and JavaScript. The PHP function logger and log fusion are implemented using C and Python respectively.

A. PHP Function Logger

To implement our PHP function logger, we leverage the function execution hook provided by the PHP execution engine. We write a profiling function that intercepts calls to user-defined and built-in function handlers to retrieve the execution context before calling the original functions. When logging, we record the called function, the function's call site (i.e., the file and line at which the function is called), the function's parameter values and the function's definition location (i.e., the file and line range that implement the function). To reduce the runtime overhead, we only instrument user-defined PHP functions (i.e., those not built-in to PHP itself) and certain database-related built-in PHP query functions. The database functions are essential for us to link the PHP's execution trace to database query. In our implementation, we use a special function, called `RINIT`, to identify the handling of user request. When that function is called, we record the PHP worker process's process ID to appropriately handle PHP's multi-process model.

B. The PEGASUS Backend: Log Fusion and Scoring

We adapt a reverse collection approach that starts with the erroneous SQL query. To link an SQL query with the responsible PHP function, we search for `mysqli_query` functions in the PHP function log and retrieve the parameter values to obtain the strings associated with queries. We connect the query log with a PHP function if there is a string match with the SQL log entry and the PHP function's parameter. Once the first PHP function is identified, we then recursively look for a PHP function's caller until we reach a global scope function. A function's caller can be identified by examining the `call site` and `definition` attribute in each function log entry. A function `X` is the caller of function `Y`, if function `Y`'s call site is within the definition range of function `X`. A global scope function can be identified if it has a special "main" function as its caller. By examining the "main" function's call site, we obtain the script's name that handles the user's request. PEGASUS tracks both the registration and invocation of callbacks and event handlers.

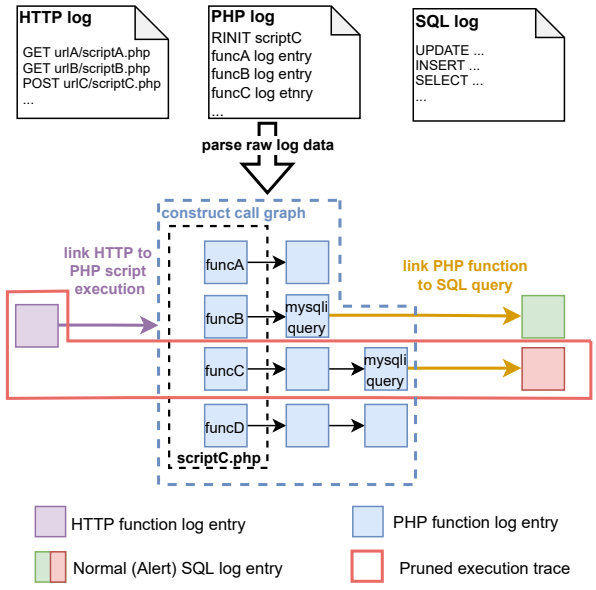

Fig. 4: PEGASUS fuses data from three log sources and distills the data into a set of execution traces associated with SQL queries that generate security alerts. This novel data set forms the basis for our visualization approach.

After completing the SQL and PHP steps, we have a function call stack that leads to a error SQL query. Our final step is to link the trace to a HTTP log entry. We do this via the HTTP request's URL. Often, as is the case for WordPress, the

script's relative path and file name is explicitly specified in the URL, which simplifies the linking process. Other applications may use URL rewriting or a single navigation script to dispatch each request. For them, we use temporal correlation.

Once we finish processing the execution trace, we apply our suspicion scoring heuristic to the linked execution trace. Each function in the execution trace starts with a default suspicion score. We use the function's name, call site and definition file to match each heuristic mentioned in Section III-F and adjust the score accordingly. The backend provides an API to the front-end to allow the display of theses scores and the trace data. The front-end encodes each node in the Node-Link graph with different color (shown in Figure 2) to denote the difference of each function's suspicion score.

C. The PEGASUS Front-End: Interactive Visualizations

When entering the system, PEGASUS shows an overview interface that displays a list of security violation incidents. Each incident includes information on its server instance identifier, the timestamp of the incident, and the rejected SQL query. From the interface, the user can select an incident of interest. Upon doing so, the frontend will direct the user into the workspace interface, acquire the relevant log data, and use that for the working set.

The workspace interface is where analysts perform most of the tasks. When first entered, the interface presents a single `EmptyContainer` component. The analyst may then choose which view window to add through a `ContainerSelector` component. The selector allows the analyst to create different view windows or to split the current window. The analyst may dynamically manage windows by splitting the `ColumnContainer` and the `RowContainer` components either vertically or horizontally. This feature generates arbitrary numbers of resizable panes in the interface, allowing the analyst to view different aspects of the incident and move between perspectives and focus as needed.

A key part of the analyst's task is to examine source code. We implement this component using the open-source and web-based Monaco editor [16] which supports code formatting and highlighting to aid readability. To support code annotations, we use a clickable Glyph Margin (a vertical bar on the left side of the editor window that typically displays line numbers) in Monaco to allow analysts add comments to the code they are reviewing. As annotations are added, they are displayed as a separate view zone, which preserves the original source code line numbering in the code view (as shown in Figure 2). Within the code view, we implement an alternative stack mode that only displays the code that ran before the security violation was flagged (since code run after the incident could not have contributed to the defect). As we show in Figure 3, the stack mode splits the code view into multiple segments, each displaying only the relevant implementation of each function. The last line in each segment is the next function that is called in the execution trace. In the stack mode, a developer can examine all the code executed before issuing the query by strolling the code view window from top to bottom.

To represent the call stack and order of events in the execution trace, we use a node-link diagram that is both zoomable and panable to allow the analyst explore and focus on items of interest. Clicking on the arrows between the PHP function nodes displays the parameters used in the called function. As shown in Figure 2, clicking on a node reveals the context information (i.e., the request payload data for a HTTP node or the function's call site for a PHP function node) in a floating box. Further, to save the analyst's time in switching between log data and source code, we direct the source code view associated with the window to jump to the function definition associated with the clicked node.

V. Evaluation

To evaluate on how well PEGASUS supports analysts in navigating diverse the log data in vulnerability analysis, we pose the following evaluation questions:

1) EQ1: To what extent can detailed logs help analysts identify the defect associated with a web application vulnerability when responding to incidents?
2) EQ2: To what extent do the PEGASUS visualizations of function call stacks simplify analysis and how do they impact the time required for defect identification?

Following recommendations from prior work [23], we consider the dimensions of PEGASUS such as the feature set utility, task performance, and insight generation in our evaluation. These dimensions are essential in ensuring the effectiveness of PEGASUS's data and design in helping analysts with localizing software security defects.

A. Effectiveness of Log Distillation

We answer the first evaluation question by applying our data fusion and distillation process on real-world vulnerabilities and determining if the defect is present in the filtered result. If the defect is in the resulting data, we consider the result accurate. We compare the amount of data in the filtered result with the original volume of the relevant log data to determine the precision of our approach.

We explore how the approach works across seven vulnerabilities independently identified and recorded in the MITRE Corporation's list of common vulnerabilities and exposures (CVEs) [17]. We list these vulnerability identifiers in the first column of Table I. Given its ubiquity, we focused on vulnerabilities in the WordPress content management system. We further selected vulnerabilities that would be detected by the base SuS platform, namely those causing a privilege violation in a backend database or being flagged by the middlebox protecting that database. While the SuS platform and middleboxes can be extended to support other backend resources, such as file or mail systems, our scope with PEGASUS is to explore the utility of logs and visualizations. Accordingly, we focus solely on the issues already supported by the SuS platform.

To explore each CVE, we create a vulnerable WordPress instance inside the SuS platform in an isolated VM environment. If the vulnerability uses a WordPress plugin, we

also install the version of that plugin that is known to be vulnerable. We then run the exploit scripts associated with each CVE on the WordPress instance. These scripts typically act as web clients that send specially-crafted messages to the website. Accordingly, the scripts' execution causes the SuS platform to log HTTP, PHP and MySQL activity. In production deployment, deployers must configure the SuS middleboxes with permissions and schema constraints so that the SuS platform can automatically detect and alert upon vulnerabilities. Those efforts are out of the scope of the PEGASUS work, so to simplify our testing, we manually mark the malicious SQL queries as the start entry of our reverse data collection process (Section IV-B). For each of the tested CVEs, we examine the patch data for the CVEs to verify the actual vulnerable function, which we use as ground truth.

TABLE I: PEGASUS filters over 97.5% of logs while preserving the code needed to identify the defect.

CVE	Distillation Data Reduction Rate by	
	Log Volume (in MB)	Num. Functions
2021-24182	98.27% (1.20/60.65)	99.28% (8/1112)
2021-24183	97.95% (1.16/56.49)	99.26% (8/1083)
2020-13693	99.49% (0.17/54.48)	98.33% (19/1136)
2019-9881	99.72% (0.17/61.87)	98.19% (26/1434)
2019-9880	99.73% (0.17/63.98)	97.56% (31/1272)
2019-9879	99.81% (0.12/65.30)	98.11% (28/1480)
2018-19207	99.24% (0.15/19.19)	98.88% (9/802)

When evaluating the accuracy and precision of PEGASUS, we look at two metrics: the percentage of log data filtered in the log distillation process and the percentage of unique functions filtered during distillation. In Table I, we show the impact of the filtering. In each case, the defective source code remains in the PEGASUS output, making PEGASUS's filtering accuracy 100%. Further, the table results show that from both a raw data volume perspective and from a user-defined function perspective, PEGASUS eliminates over 97.5% of entries. On average, analysts typically would have less than 30 functions to examine after PEGASUS's attack reconstruction.

While 30 functions may be reasonable for an analyst to explore, the heuristic approach from Section III-D may help an analyst start with the functions most likely to be relevant to correcting a defect. We now examine how accurate and precise the heuristics can be in labeling the functions to help developer prioritize their task.

TABLE II: Accuracy and precision of heuristics to highlight functions most likely to harbor the defect. For CVE-2019-9879 to CVE-2019-9881, we manually verified the the cause by comparing the vulnerable version with patched version.

CVE	Ranking of Vulnerable Function	Cause Verification
2021-24182	1	[25]
2021-24183	1	
2020-13693	2	[12]
2019-9881	7	manual verification
2019-9880	9	
2019-9879	7	
2018-19207	1	[26]

In Table II, we show the results of our heuristic labeling. For the CVEs we examined, we ranked the most vulnerable functions. Our heuristics specifically looked for functions associated with the plugins path in WordPress. In doing so, PEGASUS further narrows the scope of vulnerable functions. For more than half of the CVEs, the defect was found in the top two most suspicious functions.

The PEGASUS approach can filter the data analysts must examine. In some cases, that examination can be further aided by heuristics to help highlight the functions most likely to harbor defects, allowing analysts to prioritize their efforts.

B. Quantifying Analyst Effort

We evaluate the visualization components in PEGASUS by quantifying the number of actions an analyst must perform in identifying and annotating a defect. While the underlying log data and source code associated with each vulnerability varies, we find that the process of localizing a vulnerability for a web application requires repetition in developer actions.

TABLE III: KLM operations, their times, and their symbols.

Operation	Symbol	Time (s)
Pointing to a target	p	1.10
Keystroke	k	0.20
Mentally preparing for an action	m	1.35
Filling in a text field	T	2.32
Scrolling	S	3.96
Button Press or Release	b	0.10

Therefore, to quantify such a process accurately, we first use a Keystroke-level Model [13] to define basic common mouse and keyboard actions a user usually performs. As shown in Table III, we use different symbols to represent constituent actions and show the time required in performing each action. Then in Table IV, we use these basic actions to compose and define the common actions performed in PEGASUS.

TABLE IV: Common actions performed in PEGASUS

Action Code	Description	KLM
A1	Open overview interface	mpbb
A2	Open workspace interface	
A3	Select security violation	
A4	Split window	
A5	Add "Flow Graph" widget	
A6	Add "Code View" widget	
A7[i]	Click the left side "arrow" connecting the i^{th} node in the flow graph to examine corresponding function's parameter	
A8[j]	Click the j^{th} node in flow graph to examine the j^{th} function's implementation code in code view	
A9[k]	Click the (k-1)th node in the flow graph and scroll in the code view to examine the k^{th} function's call site	mpbbS

With these common actions defined, we then explore analyst effort by recording an analyst performing the analysis workflow for CVE-2018-19207. For comparison, we record the same workflows for the analyst with legacy tools. In these recordings, the analyst is given the same log data. In the

PEAGSUS scenarios, the debugging tool is PEGASUS itself with the log data loaded. For the non-PEGASUS scenario, the analyst uses command line tools like `vim`, `vscode`, and `grep`. These tools provide the basic searching and code navigation functionality present in PEGASUS, and are commonly used in the broader security analyst toolchain [24].

As a result, we found that with PEGASUS, an analyst spends 33 to 104 seconds (whether examine the function sequentially or rely on suspicion scoring) locating the vulnerable function for CVE-2018-19207. In the non-PEGASUS case, the same task requires around 405 seconds to complete. From the recordings, we found that in actions A1 through A6, an analyst can easily set up the analysis environment in a single integrated interface that combines both the code and the corresponding log data. The convenient setup saves the time spent in locating related logs and code, which are usually performed in a non-PEGASUS scenario. Through actions A7 to A9, an analyst can easily determine the causal relationship between logs generated by different application components or by different functions. Importantly, an analyst does not need to spend time identifying the code responsible for the log entry since our interfaces already fuses that information.

VI. Concluding Remarks

In our work, we find that fusing and distilling log data from multiple sources can significantly reduce the log volume associated with a security incident. By providing a visualization that shows causal links between events in malicious execution traces and fusing the log data with the underlying source code, we find analysts can save significant time in locating the root cause for a web application vulnerability. The heuristics we use in our evaluation are simple and based on usage frequency, historical vulnerabilities in plugins, and string similarity. The suspicion scoring component is designed to be easily extended and could be enhanced with insight from code analysis studies. As with code "linting" tools for debugging purposes, highlighting instances of poor programming practice may help analysts quickly identify the cause of vulnerabilities.

In our future work, we will examine PEGASUS's effectiveness in helping developers save analysis time via user studies. In addition, we will explore other web applications and code bases to assess the generalizability of our findings. We examine WordPress in our evaluation due to its widespread use on the Internet, enabling PEGASUS to have a significant impact. Likewise, WordPress's popularity provides ample well-documented vulnerabilities for our evaluation. At the same time, PEGASUS can be easily expanded to other applications running on the SuS platform that use SQL and PHP.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. 1814402 and 1815587.

References

[1] Accenture. Accenture 2019 cost of cybercrime study. https://www.accenture.com/_acnmedia/PDF-96/Accenture-2019-Cost-of-Cybercrime-Study-Final.pdf, 2019. Accessed June 17th, 2021.

[2] A. Bates, W. U. Hassan, K. Butler, A. Dobra, B. Reaves, P. Cable, T. Moyer, and N. Schear. Transparent web service auditing via network provenance functions. In *International Conference on World Wide Web*, p. 887–895, 2017.

[3] M. Bellare and B. S. Yee. Forward integrity for secure audit logs. Technical report, University of California at San Diego, November 1997.

[4] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen. Reversible debugging software: Quantify the time and cost saved using reversible debuggers. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

[5] G. Canfora, A. Di Sorbo, S. Forootani, M. Martinez, and C. A. Visaggio. Patchworking: Exploring the code changes induced by vulnerability fixing activities. *Information and Software Technology*, 142:106745, 2022. doi: 10.1016/j.infsof.2021.106745

[6] B. Cornelissen, A. Zaidman, and A. van Deursen. A controlled experiment for program comprehension through trace visualization. *IEEE Transactions on Software Engineering*, 37(3):341–355, May 2011.

[7] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, p. 326–337, 1993.

[8] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Software Visualization*, pp. 151–162, 2002.

[9] A. Di Sorbo and S. Panichella. Exposed! a case study on the vulnerability-proneness of google play apps. *Empirical Software Engineering*, 26, 07 2021. doi: 10.1007/s10664-021-09978-0

[10] N. Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, Oct. 1988. doi: 10.1145/54289.871709

[11] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates. Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis. In *Network and Distributed System Security Symposium*, 2020.

[12] R. Karger. Analysis of cve-2020-13693. https://blog.raphael.karger.is/articles/2020-05/CVE-2020-13693, 2020. Accessed June 17th, 2021.

[13] D. M. Lane, H. A. Napier, R. R. Batsell, and J. L. Naman. Predicting the skilled use of hierarchical menus with the keystroke-level model. *Human-Computer Interaction*, 8(2):185–192, 1993.

[14] J. P. Lanson. Single-use servers: A generalized design for eliminating the confused deputy problem in networked services. Thesis, Worcester Polytechnic Institute, May 2020.

[15] K. Mehner. JaVis: A UML-based visualization and debugging environment for concurrent java programs. In *Software Visualization*, pp. 163–175, 2002.

[16] Microsoft. Monaco editor. https://microsoft.github.io/monaco-editor/, 2021. Accessed April 28th, 2021.

[17] MITRE. CVE. https://cve.mitre.org/, 2021. Accessed June 17th, 2021.

[18] MITRE. CVE-2018-19207. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-19207, 2021. Accessed June 24th, 2021.

[19] T. Munzner. A nested model for visualization design and validation. *IEEE transactions on visualization and computer graphics*, 15(6):921–928, 2009.

[20] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. Clamp: Practical prevention of large-scale data leaks. In *30th IEEE Symposium on Security and Privacy*, pp. 154–169. IEEE, 2009.

[21] Positive Technologies. Web applications vulnerabilities and threats: statistics for 2019. https://www.ptsecurity.com/ww-en/analytics/web-vulnerabilities-2020/, 2019. Accessed June 17th, 2021.

[22] Q-Success / W3Techs. Usage statistics and market share of wordpress. https://w3techs.com/technologies/details/cm-wordpress, 2021. Accessed June 25th, 2021.

[23] D. Staheli, T. Yu, R. J. Crouser, S. Damodaran, K. Nam, D. O'Gwynn, S. McKenna, and L. Harrison. Visualization evaluation for cyber security: Trends and future directions. In *Workshop on Visualization for Cyber Security*, p. 49–56. ACM, 2014.

[24] R. S. Thompson, E. M. Rantanen, W. Yurcik, and B. P. Bailey. Command line or pretty lines? comparing textual and visual interfaces for intrusion detection. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, p. 1205, 2007.

[25] Wordfence. Several vulnerabilities patched in tutor lms plugin. https://www.wordfence.com/blog/2021/03/several-vulnerabilities-patched-in-tutor-lms-plugin/, 2020. Accessed June 17th, 2021.

[26] Wordfense. Privilege escalation flaw in wp gdpr. https://www.wordfence.com/blog/2018/11/privilege-escalation-flaw-in-wp-gdpr-compliance-plugin-exploited-in-the-wild/, 2018. Accessed June 17th, 2021.