PEGASUS: Powerful, Expressive, Graphical Analyzer for the Single-Use Server

by

Matthew Puentes

Thesis Proposal

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

Matthew Puentes

May 2021

APPROVED:

Professor Craig A. Shue, Thesis Advisor

Professor Lane Harrison, Thesis Reader

Professor Craig E. Wills, Department Head

Abstract

In recent years, the codebases used for web development have grown due to the widespread use of content management systems such as WordPress. However, the debugging tools and infrastructure available for web developers have not yet grown to match that new size. This lack of tooling can lead to extended downtime as developers go through hundreds of files of code to find a bug once a problem has been detected, often with insufficient logging. In this paper, we seek to determine how precisely we can identify and communicate the location of a bug using the Single-use Server logging infrastructure, as well as how well we can communicate that to the user. To answer this question, we designed and implemented a debugging tool for web applications named PEGASUS, which stands for Powerful, Expressive, Graphical Analyzer for the Single Use Server. PEGASUS assists in the debugging process using information provided from the Single-use Server project, including proxy information, a full PHP call stack, and a resource guard to prevent confused deputy attacks. PEGASUS was built using a Node server, D3.js visualizations, and a Vue.js webapp to host an interactive frontend. During testing, the debugger specified the code location of a known vulnerability to just 5% of the lines in the files run by the webapp. PEGASUS also outperformed the normal debugging process in both time and simplicity, resulting in a faster and more user-friendly debugging experience.

Acknowledgements

This project would not have been possible without the support of Dr. Craig Shue, whose patience, guidance, and humor helped me through what would have been an otherwise impossible year. His work as my graduate advisor, Scholarship for Service coordinator, and professor did a a great deal for me during my five years at WPI, and for that I am exceedingly grateful. I am proud to have worked on this project under him, and he has helped my growth as a computer scientist and as a professional more than I can express.

The project also benefited from the generosity of my Cake lab peer Yunsen Lei, whom I thank for his knowledge of the Single-use Server, his work on the PHP function logger, and his help on nearly every part of this project. I'd also like to thank Dr. Lane Harrison for his feedback on my paper and his help with prototyping, as well as Beckley Schowalter for her proofreading and help through the SFS program.

I want to thank my peers Joseph Petitti, Cole Granof, Anthony Heng, and Eric Solorzano for their help, both on the thesis and otherwise, as we all completed our Masters degrees remotely. The 2020-2021 year was a difficult time, and it made it much easier having them nearby.

Finally, this was only possible because of the financial support I received from the National Science Foundation's Scholarship for Service. SFS allowed me to pursue the college career of my dreams, and I am proud to have been a part of this program during my stay at WPI.

Contents

1 Introduction

Modern web development has recently grown in scale. Rather than using small scripts to help beautify a web page or add a small interactive component, modern web design requires a full stack of code and technologies working together to create the complicated web pages that are found online. Even blogs, which in theory are just text and images, are now often created on a web development platform, such as WordPress. These platforms make the web development process more accessible, opening up more interesting options for website or blog creation to people with any level of technical skill.

However, the detailed and complex nature of these backend systems make them difficult to understand. WordPress has a fairly complicated backend, composed of over 1,000 files and directories of PHP code [21]. This, combined with the asynchronous nature of web development and the vast array of technologies that can be used to develop websites, leads to difficulty in debugging web applications.

As web technologies evolve, the tools we use to create them need to evolve as well. Debugging is one of the most apparent gaps in our development technologies, as classic software debugging tools were built for a pre-web, non-asynchronous, non-containerized era. Without debugging capabilities, developers can find it hard to respond to bugs or errors throughout their development process. This can mean hours of time spent doing debugging work without the proper tooling. Even worse, this situation could lead to security vulnerabilities, which might cause websites to be shut down for unknown amounts of time until the problems are resolved.

Some work has been done to help detect these security vulnerabilities. Tools such as Raditus [5], CLAMP [14], and the Single-use Server [8] (SUS), have all been developed to help combat common vulnerability types. While these programs are successful in mitigating these vulnerabilities, they stop at detection and isolation, lacking tooling to actually help remediate the security vulnerabilities themselves.

The Single-use Server's contributions to web application debugging are notable, as they provide specific and detailed logging for "confused deputy attacks," or attacks in which the adversary somehow tricks the more-privileged server into performing an action the user would not be allowed to do on their own. When an attacker uses an exploit to trigger a confused deputy attack, the Single-use Server uses its server-level permissions to detect that a forbidden action has occurred and logs the action as being rejected. This means that unlike typical web development, in which security issues are found through good Samaritans or other signs of a breach sometime later, the Single-use Server can use adversaries as penetration testers. This framework allows for the defects in the codebase to be exploited with no negative repercussions on the webapp as a whole and provides the logging information to determine the source of the defect. However, while these logs are good to have, having a human process thousands of lines from several different log files manually would be tedious and difficult, negating some of the benefit received by SUS in the first place. This leads to our research question for this project: using the logs available from the Single-use Server, how accurately and precisely can we determine the location of a defect in a codebase, and how easily can we communicate this to the end user?

To answer this question, we developed PEGASUS (the Powerful, Expressive, Graphical Analyzer for the Single-use Server), a forensics and debugging tool that advances the Single-use Server work done previously by Lanson et al. [8]. The program will take Single-use Server logs as input and be able to output which functions in the codebase are likely to contain the vulnerability. Additionally, it will try to communicate this information through a user interface, complete with visualizations, a feedback loop from the user adding their own data, and exploratory data access.

This program will act as a debugger that runs on top of the Single-use Server (SUS). As previously mentioned, SUS is a proposed design to help mitigate common concerns with web development security. SUS accomplishes this by containerizing each user as they authenticate with a website and ensuring that every program the user interacts with is given the permissions of that user only. This containerization prevents "Confused Deputy" attacks and significantly reduces the potential impact of bugs within the web service's code. The "bugs" in which PEGASUS will specialize are closer to security defects than to simple syntax errors, as bugs that could be found simply by trying to run the source code would not be too difficult to find. This project will utilize the pre-existing SUS logging infrastructure, which is where we will get our data for finding where the bug lies.

Our program design will be divided into two primary sections: the *Pegasus-Frontend* and the *Pegasus-Backend*. The backend will contain the logic used by the program to parse it's data sources and find where vulnerabilities might lie in the code. This process will rely on the data collected by the Single-use Server. However, as we are aiming to create a debugger for an end user rather than a filter on top of SUS's existing logging tools, we will also include a frontend which will be responsible for displaying the data composed by the backend. The frontend will have tools that allow a developer to pinpoint the source of an error and a work space that allows the developer to view and annotate the codebase they are working on.

The main benefit of PEGASUS will be an easier debugging process, costing less time and effort. While effort is difficult to quantify (we use lines of code as an approximation), time is a fairly quantifiable metric. An elongated debugging process costs human time, which is associated with an economic loss for those tasked with maintaining the website. Additionally, the downtime a website might have between detecting and fixing a bug could represent another economic loss. A 2019 study found that at least 16% of the top ten million websites are currently using WordPress [4], which is one of the platforms PEGASUS supports. We believe that 1,600,000 websites without any built-in debugging software or assistance may incur a significant loss of efficiency and time. We hope to mitigate that problem with this tool.

As with most debugging platforms, the primary source of competition is development without using a debugger at all. Often, developers simply do not use a debugger when debugging applications, as leaving their standard workflow or learning a new tool can be a hurdle they do not want to overcome. After all, if it takes an hour to learn a tool that halves one's debug time, but the bug would have only taken an hour to solve anyway, one may effectively lose time by using a debugger. In order to combat this upfront cost, a debugging tool must be both user-friendly and adaptable to any user's workflow. This means that our overall program workflow should be able to be split up, with individual tasks (such as viewing function logs, getting parameter values, or getting a program trace) easy and quick to do from any program state. While other debugging tools exist, we also have the advantage of being bundled with the Single-use Server system. When one already uses SUS, they have an implicit advantage in already having PEGASUS set up and installed.

2 Related Work

PEGASUS spans several distinct subfields, so we describe prior work in several areas. Our program is a debugging tool, so we try to identify how users actually debug a program, as well as what tools they need. Next, we explore work that shares our data source of log files and logging output, to see how its unique challenges are addressed. Finally, we address the visualization aspect of PEGASUS by looking into three different data visualization areas: how cyber security programs tend to visualize data, ways to represent data hierarchically, and evaluation methods for data visualization tools.

2.1 Understanding the Debugging Process

When creating a debugging tool, it is important to design around the actual debugging process that a user goes through. In the work of Grigoreanu et al. [6], the authors try to understand the user debugging process through the "sensemaking" model of human information gathering. While much of this paper is more in-depth than what we need to analyze, their discussion and conclusions contain certain revelations that are important to our program design. The first is that users require some sort of annotation or debugging tool with sufficiently large debugging projects. Without a tool, users can be lost in their own trail, re-treading or forgetting bugs that they have already examined or resolved. Second, the paper talks about "Information Foraging," which conceptualizes how users will try and understand the debugging problem. This paper identifies two main classes of information foraging: comprehensive and selective. The former cares more about getting context to understand the program, and the latter tries to learn only what they need to fix the error quickly. Each approach has advantages and disadvantages, but it is important that PEGASUS supports both. This means allowing all data to be accessed and understood in a potentially explorative way, while not overwhelming the user with all the data at once, so the user can focus.

2.2 Log Visualization

We examined papers that were generating visualizations from the same data source that we were, namely log files, which are very dense and data-rich but are not very human-readable. We first looked at the work of Wu et al. [22], which tackles the task of visualizing Windows log traces. Those contain a huge volume of data and produce an extremely dense graph. Wu et al. use a combination of several different visualization techniques, including a dotplot, 2-axis histograms on each side of the dot plot, and multiple bar-codes across the graph. These visualization methods produce an extremely dense graph where each pixel can represent several data points, and can easily handle the log files that could potentially have tens of millions of events. This work provides important insight for our project by emphasizing how dense visualizations can be, which is especially important when visualizing large log files. While it uses much larger files, and focuses on detection rather than remediation, its core idea of summarizing a lot of data in little space is still applicable. Additionally, this paper uses multiple layers of visuals to compound how much data they can represent.

Some work has also been done in the reading of generic log files. FLUKES [1] is a system by which the authors analyzed log files without knowing the structure in advance. This project uses machine learning for the generalization, as well as REGEX and D3.js to read and represent the log files themselves. These tools are applicable to PEGASUS, since they share a web-based nature. Tools such as REGEX are necessary to parse a lot of log files quickly, and the D3.js library seems to be the common standard when using JavaScript or Typescript for data visualizing.

2.3 Cyber Security Visualization

Many of the papers we examined focused on cyber security specifically. This is useful, as the goals of a cyber security visualization are similar to the goals of PEGASUS: communicating problems to a user productively.

CSight [2] is a system that seeks to debug concurrent systems with finite state machines. The log files in a concurrent system are often difficult to parse due to their asynchronous nature and possible timing differences between segments, so a more consolidated log or visualization is better for distributed programs. The CFSM (communicating finite state machine) created in this paper solves the concurrency issue by attempting to create connections between events and simplify complicated graphs into understandable trees. This paper helps emphasize how causality between events is essential for tracing how a system is executed. While PEGASUS should not have issues with asynchronous data, as SUS timestamps each of its logs from the same internal clock, communicating this concept to the user will be important.

A taxonomy of the different types of cyber security visualization has also been proposed in the work of Kasemsri et al. [7], which separates the field into scatterplots, color maps, glyphs, histograms, and parallel coordinate plot systems. It notes that there are many of non-categorized graph types that span categories, as well as combinations of those which can result in more expressive graphing types. This work shows the scope of cyber visualization and the tools at our disposal.

2.4 Data Visualization Evaluation

EEVi [18] is a proposed standardized framework for generating cyber security visualizations. This paper defines the eight roles that a cyber visualization can help accomplish, which is helpful for identifying if, and where, PEGASUS would overlap with cyber visualizations. Additionally, it emphasizes many important aspects required of a cyber visualization in order for it to be effective, including user interaction, which is a focus of PEGASUS.

Staheli et al. studied how the data visualization community evaluates its own work, specifically in the field of cyber security [19]. They surveyed 130 papers over a decade of cyber security visualization research to identify how, and how effectively, data visualization methods were evaluated. One of their conclusions is that the usage scenario is a common technique. A usage scenario is a weaker form of evaluation that is mostly a developer-led demonstration of the tool or visualization's abilities, rather than a study of how users interact with it. However, the usage scenario is one of the most popular evaluation methods in the time frame they analyzed. This is valuable information, as it reveals that one can get good qualitative data on visualization performance without needing a full user study.

Miniukovich and Angeli have studied the attractiveness of user interface aesthetics, which they claim to be an important part of a user's experience [10]. They attempt to measure these aesthetics automatically, and for websites they separated their metrics into visual clutter, color variability, contour congestion, figure-ground contrast, layout quality, symmetry, prototypicality, and ease of grouping. While PEGASUS will not be undergoing their automated analysis due to time constraints, the metrics and definitions they lay out for how aesthetics can help the user experience provide valuable feedback into how PEGASUS' frontend should look.

3 Approach

The goal of this project is to determine how effectively we can use the Single-use Server debugging tools to help users debug web applications. From goal leads to several requirements for the project. First, we will need a way to process the log data we receive as input from the Single-use Server. Second, we will need a frontend for the user to read the data from, as well as to interact with. This frontend will involve both a data visualization element and some sort of user data export system. The data visualization will help translate the dense log files into a representation that is easy for the user to understand, and the user data export system will help turn the work done, and conclusions reached by the user, back into data. Debugging is often collaborative, so a method by which users can share results from their debugging experience can help make this tool a part of a

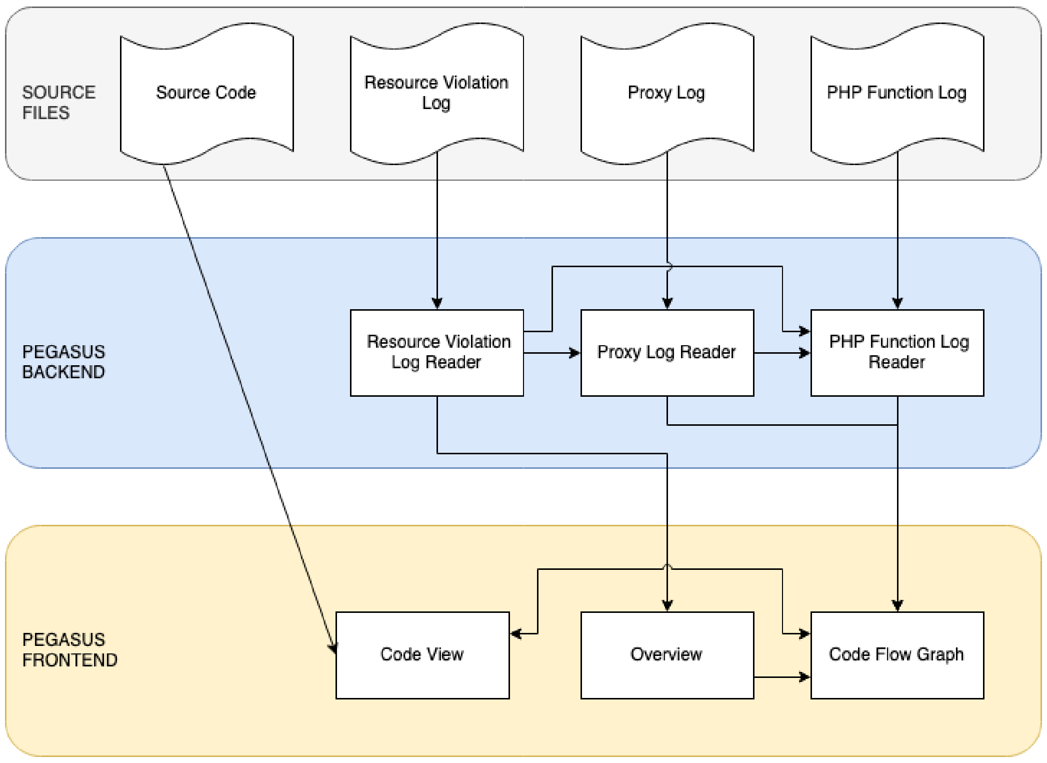


Figure 1: Overview of the Pegasus system

development workflow.

With those requirements in mind, our approach will divide PEGASUS into two components, *Pegasus-Frontend* and *Pegasus-Backend*, as discussed in Section 1. A summary of the structure of PEGASUS can be found in Figure 1.

This project is implemented on an existing Single-use Server instance that is running a WordPress Installation. For the development of SUS, an older version of WordPress with several known security vulnerabilities was used as a platform. PEGASUS was developed on the same infrastructure. The advantages to using this WordPress instance are twofold. First, it allows us to begin developing PEGASUS immediately, without re-creating a Single-use Server instance. Second, WordPress is a well-documented platform with a number of plugins that have security vulnerabilities, which we can use in our evaluation of PEGASUS.

The Single-use Server has three different logs that PEGASUS uses for its data source: the proxy log, the resource guard log, and the PHP function log. The proxy log records all the connections to and from the Single-use Server instance, and is helpful when determining when and how user flows began. The resource guard log notes every time the Single-use Server accesses a shared resource between SuS instances, such as the database of a WordPress website. This is useful for finding the time data for when an exploit takes place, which in turn helps link the violation to the PHP code

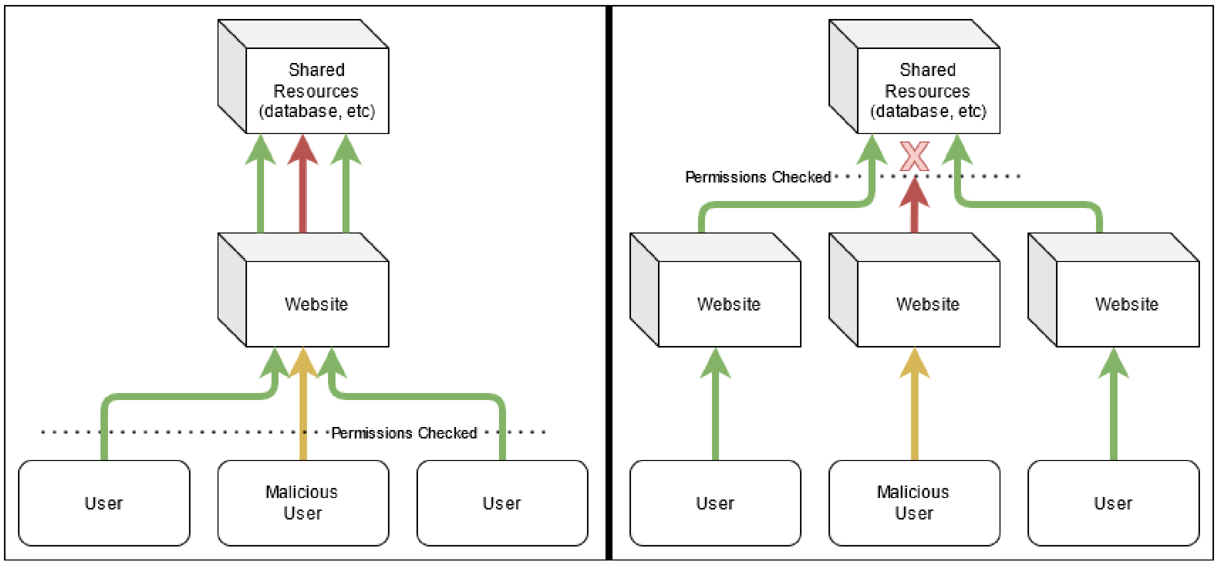


Figure 2: An illustration of the confused deputy problem and how the Single-use Server system addresses it. The red arrow represents malicious traffic using the website's permissions, and the yellow arrow represents the traffic exploiting the confused deputy.

that triggered the exploit. Finally, the primary data source for PEGASUS is the PHP function log, which logs each PHP function that is called, including when it was called, from where, and where the function was defined. All three log files are described in more detail in the implementation section.

This logging data provides insight into what a malicious user did while interacting with the server, as well as how they did it. Using this logging data, it should be possible to link the HTTPS request from the proxy log, the PHP function calls in order, and the resource guard violation using the timing and order of events. Connecting this information gives us a timeline of what happened, which should be our first step in communicating this data to the user. Second, more data can be gathered by looking at the log files in unison, such as by linking strings from the HTTPS request with strings found in the PHP function parameters. Both of these steps will be handled through the *Pegasus-Backend*.

3.1 Pegasus-Backend

The backend of PEGASUS will have two tasks. First, it needs to be able to parse data from the log files obtained from the Single-use Server. Second, it needs to be able to combine its data sources into an output listing the functions that are the most "suspicious," or the most likely to contain the program bug.

The data parsing is straightforward and mostly uses Regular Expressions or JSON parsing tools to parse the files. These tools read the log files into structured objects. *Pegasus-Backend* will also be able to read different parts of a file based on parameters from the AJAX requests sent by *Pegasus-Frontend*. These AJAX parameters will determine which logs are retrieved and how they are processed. The AJAX parameters and log file objects are described in more detail in the implementation section.

We will also need to put the data together into a result. The three log files should contain enough information to allow us to build a trace of the executed functions. This will involve linking the HTTPS request, resource guard violation, and PHP function call stack by time. The resource guard violation request should line up with a PHP function that accesses the SQL server, and the HTTPS request should line up with the beginning of a PHP code flow. With those three linked together, *Pegasus-Backend* can reconstruct the complete timeline of what the attacker did in order to trigger the exploit.

Further, we can derive additional information from the SuS logs. *Pegasus-Backend* can collect data such as whether the function came from a plugin, if the function name matches known builtins, or how many of its arguments are found in the initial HTTPS request. From this data, we can create the "suspicion" heuristic for the likelihood that a given function contains the code defect, in order to help the end user focus their attention on one area.

These data will be used for several things, mostly on the frontend. First, highlighting functions and lines of code in a code explorer will help the user visualize where their debugging attention should be focused. Second, functions and files that are not flagged, or have a suspicion level value of "0," can be ruled out and hidden from the user to help clean up our visualizations. Finally, the levels of suspicion will help a user incrementally expand their search as they continue the debugging process.

3.2 Pegasus-Frontend

The frontend will have two primary tasks. First, to display the data from the SuS logs and backend without overwhelming the user, in order to help them focus their debugging on areas where the bug is likely to be. Second, the user should have a way to track their own progress by allowing them to make annotations on certain lines, reflecting their progress through the debugging process.

Displaying the SUS logs requires a visualization, preferably one with user interactivity. Our goal is to provide an overview of the execution of the program, giving the user both an entry point for the debugger and a way to process the data in an explorative way.

The backend will give the frontend a suspicion level for each function representing its likelihood to contain the program's error. This information can be utilized in two ways. The first is, by augmenting the overview visualization with the suspicion level data. If we highlight functions with a color or marker that indicates its suspicion level, the user will quickly be able to filter out information that is not relevant to them. Second, the user should be able to view the source code of the web application within PEGASUS. This will not be a code editor; it is merely a view of the code with syntax highlighting, and our suspicion level highlighting, to provide the user with a way to browse through the program as they debug.

Finally, the frontend should allow some sort of annotation system, letting the user leave a trail of breadcrumbs as they debug, helping themselves or other users follow the same path they did, to prevent them from re-treading the same information. The user should then be able to comment on every line in the codebase, saving and exporting those comments for storage and preserving the data between sessions.

4 Implementation

PEGASUS is implemented as two distinct projects, *Pegasus-Frontend* and *Pegasus-Backend*. Both projects were developed locally using WebStorm and Git for version control. Additionally, a configuration file is included in order to point PEGASUS towards the log files from the Single-use Server installation.

Both the frontend and the backend are primarily implemented using TypeScript. TypeScript was chosen as the primary language for this project for two reasons. First, the author is familiar with it, so less time is needed to learn the language before the project starts. Second, the typing information is helpful for a fast development cycle, as easy typing errors, that can result in long debug times, are caught in advance. TypeScript also compiles directly to JavaScript, so in instances where TypeScript cannot be used (such as when using JavaScript-only libraries), JavaScript is used instead.

4.1 Pegasus-Backend

The backend handles three primary tasks: parsing the local single use server files, composing the data, and hosting the frontend's files so the tool can be viewed in a browser. It accomplishes all three of these tasks through its Node.js server.

The server runs an `express.js` instance in order to send files over fetch requests on localhost. Several rules are set to create a basic API, so the frontend can request scripts, pages, files and fonts from the appropriate local directories on the file system. Additionally, the backend reads the PEGASUS configuration file to determine the local location of several resources, such as the log files and the source code installation.

The log files need to be parsed, and in some cases combined, before they are sent to the frontend to be displayed. Reading and parsing large files can be very slow when done through the browser, so the server aims to do as much processing as it can before giving the data to the frontend. For each log file type, *Pegasus-Backend* has its own types and logic for reading and parsing the file.

The first log file type is the resource manager log, which records every time the website requests data from the MySQL instance, as well as what the MySQL server returned. This log allows us to find the query that the single use server rejected, and it serves as our starting point for the data. The log file is organized into entries delimited by three newline and the text "NEW ENTRY".

Each log entry has the following fields:

- **Time:** the time the resource was accessed.
- **Address:** the IP of the container requesting the resource.
- **Return Code:** the return code that the single use server replied with.
- **Changed:** the SQL String.
- **Response:** the response sent by the MySQLi instance.

The second log file type is the proxy log. The proxy log contains all of the information collected by the Single-use server proxy, which is helpful for understanding the actions the end user performed. The proxy log is a text file containing rows of data, so parsing it simply requires performing string searches to match parts of the file. Each log entry is delineated with three newline characters and the text "NEW LOG:," as well as the following fields:

- **Time:** the time the HTTP request began.
- **End:** the time the HTTP request ended.
- **User:** the user token string from the user that initiated the request.
- **Ip:** the IP address of the container.
- **Request:** the HTTP request, if any.
- **Response:** the server's response as a string.

The third, and most complicated, log file is the PHP Function log, which is a recording of every PHP function that was executed during the server's runtime. This is by far the largest log file that PEGASUS has to process, so there are several options for how the file can be parsed. The PHP Function log is formatted as a JSON file with two levels: the PID and the function ID. The PID level groups every function run under that PID, effectively grouping each run of WordPress separately. Within each PID, the function IDs are used to index each function log entry, which contains the following information:

- **Timestamp:** the time that the function was run.
- **Script name:** the name of the file that called this function.
- **Line number:** the number that the invocation of this function occurred on.
- **Definition file:** the file in which the function is defined.

- **Function start:** the line on which the function begins.
- **Function end:** the line on which the function ends.
- **Caller:** the unique ID for the function that called this function in the log.
- **Parameter:** the list of all parameters passed to this function.
- **Suspicion level:** a computed property discussed below.

If the request is for `/logs/phpcode`, the backend simply returns the entire function log as a large JSON object. This is a large value, so normally one of the parsed options is used.

If the request is for `/logs/phpcode?timestamp=[value]`, *Pegasus-Backend* will process the log before sending. This processing occurs when the given timestamp matches a resource violation error in the Single-use Server, and the frontend wants to know the call stack for that error. To find it, the entire PHP function log is first searched for functions with the name `mysqli_query`, and their time is compared to the supplied resource violation timestamp. The function with the least time between it and the resource violation error is taken to be the function that caused the incident. Only functions that occured before the violation are considered. Next, because all functions contain the ID of the function that called them, the program is able to walk up the log, taking only the functions that were called before the `mysqli_query` function. This list of functions is then reversed, to be in time-order, and sent as a JSON array.

If the request is for `/logs/phpcode?index=[value]&pid=[value]`, the process is very similar, instead of searching the entire php log, the pid and index of the function are already known, so the program just has to generate the list of functions and return the JSON array.

Pegasus-Backend also handles composing the data, adding additional information, and combining logs where required. The primary example of this is the Proxy Log and the PHP Function log. When the user requests a call stack from *Pegasus-Backend*, the applicable entry in the proxy log that represents the request sent to the server is included as the first "step" of the process. This allows a user to see the entire flow of information, from HTTP request to PHP function calls and SQL errors.

Additionally, *Pegasus-Backend* adds an additional "suspicion level" to each function in the PHP call stack. This property represents how many warning flags PEGASUS detects on a certain function, which can help the user prioritize more suspicious functions first when debugging.

When calculating the suspicion level, most of the information used comes from the PHP log entries themselves. However, *Pegasus-Backend* also incorporates information from the proxy log. The proxy log represents the beginning of the malicious traffic, so any information we can pull from it is potentially useful. However, since the request could contain nearly any string, we need to take a very general approach to seperating it into useful data. With that in mind, the process is as follows: first, the proxy log is checked to see if there is a "request" field. If there is, it is filtered so that headers unlikely to contain useful information are dropped. Those headers include POST, Host, X-Forwarded-Proto, User-Agent, Accept-Encoding, Accept, Connection, Content-Length, and Content-Type. Finally, it is split into tokens by splitting the string by known delimiters (, " { } = & :). This results in an array of "tokens" from the request.

Once we have the required data, the suspicion level is calculated in the following way:

1. Suspicion starts with a default value (+0.5).
2. If the function is known to be a basic part of WordPress (e.g., a initialization function), subtract the default suspicion (-0.5).
3. If the function comes from the `wp-content/plugins` directory, add suspicion (+1).
4. If the parameters following the function contain a string from the Proxy Tokens, add a small amount for each match (+0.1).

This suspicion level is later reported to the user through the use of color, as detailed in the next section.

Finally, *Pegasus-Backend* can also read the source code of the program PEGASUS is debugging in order to send it to the front end for viewing. It gets the source code location from the PEGASUS configuration file, and does not do any additional processing on these files.

4.2 Pegasus-Frontend

The frontend of this project is what the user controls when they load PEGASUS in their browser.

Pegasus-Frontend is built in Vue.js, a JavaScript framework that allows a website to be built in "components": small files that contain all the HTML/CSS/JS that pertain to one element in the program. PEGASUS was designed with widgets in mind, so these components were a valuable abstraction. Vue.js's webpack integration is used to build the project into a static directory served by *Pegasus-backend*. *Pegasus-Frontend* uses Vue.js's "Single File Components," which define components in separate files, ensuring that there is no overlap and that code relevant to a component stays local to that component. Additionally, Vue has "mixins," which are TypeScript files that allow component functionality to be abstracted and inherited. In addition to the `main.ts` file used to launch the webapp, Pegsus-frontend uses 10 components and 2 mixins to implement its functionality.

The first component, `App.vue`, acts as the "root" for the rest of the project. In the header, it contains the title of the program, tabs for switching between the Overview component and workspace, a "reset" button, and the `ImportExport.vue` component.

The `ImportExport.vue` component handles the input and output of annotations, which *Pegasus-Frontend* shows on the CodeView component. In order to make PEGASUS as platform-agnostic

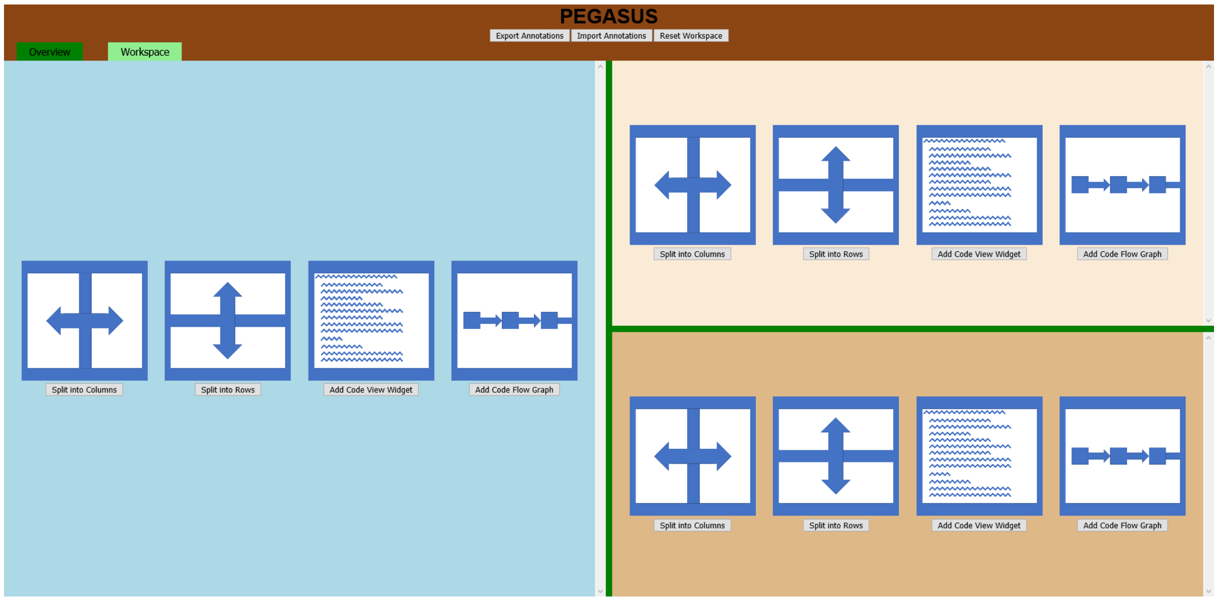


Figure 3: *Pegasus-Frontend* Overview

as possible, the annotations are encoded as a JSON file. They are exported with the filename "PEGASUS_annotations[date].json," where date is the current timestamp down to the nanosecond to prevent filename conflicts. This JSON file can then be easily imported through HTML file input fields.

The `Overview.vue` component is the last component that exists outside of the main workspace. It acts as the entry point to the program, where the user selects which error they will be debugging. In order to facilitate this, the overview component fetches the list of violations detected by SuS from the backend resourceManager node. For each element, the overview component displays information about the violation like the IP, timestamp, and attemped SQL command. Each element also has a button that, when pressed, selects that violation globally (to set the resource violation upon which PEGASUS will focus) and automatically focuses the user on the workspace tab. The user can go back and select a different resource violation at any time using the tabs on top of the screen.

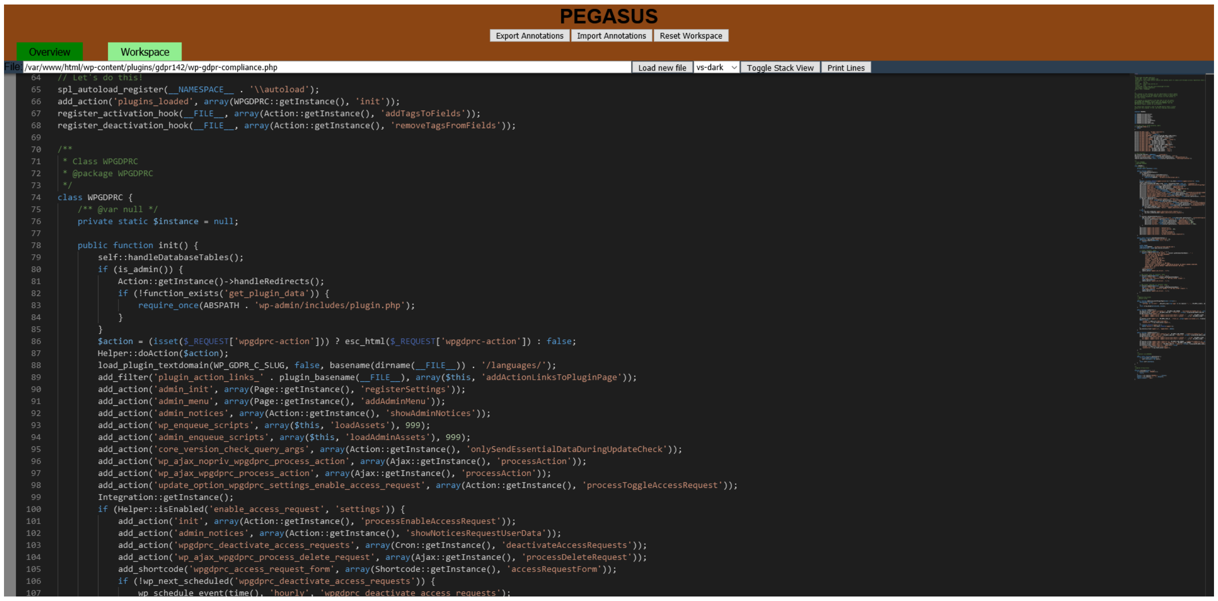
The "workspace" of PEGASUS is a tree of components that can be re-sized and split to fit the users' workflow. The workspace starts with the `EmptyContainer.vue` placeholder component, which contains the `ContainerSelector.vue` component. The `ContainerSelector.vue` component has buttons to create a new widget in the workspace, and turns itself into whatever component is selected. The first two possible components are the `ColumnContainer.vue` and `RowContainer.vue`, which handle the organization of the workspace.

The `ColumnContainer.vue` and `RowContainer.vue` components serve as the organizational structure in the workspace, splitting their parent component into two equal slices either vertically or horizontally. These slices can be re-sized by clicking and dragging a green bar, which changes the proportion of the container allotted to each half. Both components inherit from the Selector

Mixin, which holds the shared functionality between them. These components also start with another `ContainerSelector.vue` component on each side, which can be used to nest splits or create components within the resizable blocks.

The next component that can be created is `CodeView.vue`. This component acts as a read-only IDE for the codebase of the project, allowing the user to read the source code, leave annotations, and navigate through the code run by the program.

The Code View is an instance of the Monaco editor[9], a free and open-source web-based editor that acts as the basis for Microsoft's VSCode. The Monaco-Vue package is used to integrate it into a Vue component, and MonacoWebpackPlugin is used to have it all statically included within the project. The built-in language server lets us format and colorize the PHP code, making it more readable for the end user. Additionally, the Monaco "Glyph Margin" is used as a clickable space wherein the user can add "annotations" to the code they are reviewing. Annotations are then displayed as viewZones, which are displayed as their own lines, colored separately than the rest of the editor.



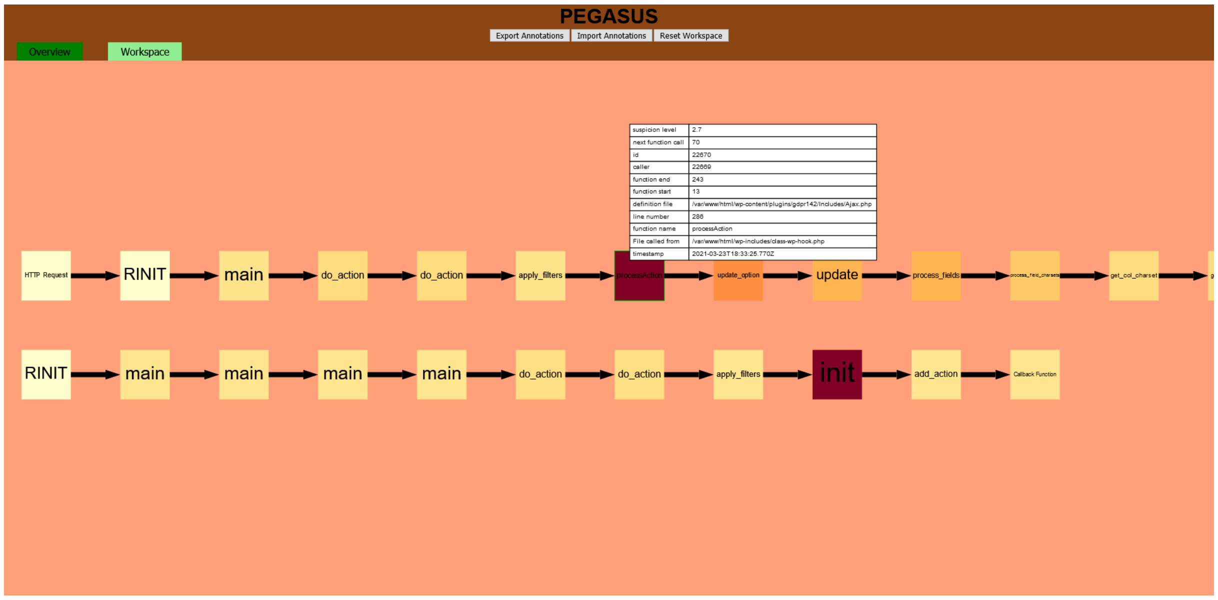
```php
// Let's do this!
spl_autoload_register(__NAMESPACE__ . '\\autoload');
add_action('plugins_loaded', array(WPGDPRC::getInstance(), 'init'));
register_activation_hook(__FILE__, array(Action::getInstance(), 'addTagsToFields'));
register_deactivation_hook(__FILE__, array(Action::getInstance(), 'removeTagsFromFields'));

/**
 * Class WPGDPRC
 * @package WPGDPRC
 */
class WPGDPRC {
    /** @var null */
    private static $instance = null;

    public function init() {
        self::handleDatabaseTables();
        if (is_admin()) {
            Action::getInstance()->handleRedirects();
            if (!function_exists('get_plugin_data')) {
                require_once(ABSPATH . 'wp-admin/includes/plugin.php');
            }
        }
        $action = (isset($_REQUEST['wpgdprc-action'])) ? esc_html($_REQUEST['wpgdprc-action']) : false;
        Helper::doAction($action);
        load_plugin_textdomain(WP_GDPR_C_SLUG, false, basename(dirname(__FILE__)) . '/languages/');
        add_filter('plugin_action_links_' . plugin_basename(__FILE__), array($this, 'addActionLinksToPluginPage'));
        add_action('admin_init', array(Page::getInstance(), 'registerSettings'));
        add_action('admin_menu', array(Page::getInstance(), 'addAdminMenu'));
        add_action('admin_notices', array(Action::getInstance(), 'showAdminNotices'));
        add_action('wp_enqueue_scripts', array($this, 'loadAssets'), 999);
        add_action('admin_enqueue_scripts', array($this, 'loadAdminAssets'), 999);
        add_action('core_version_check_query_args', array(Action::getInstance(), 'onlySendEssentialDataDuringUpdateCheck'));
        add_action('wp_ajax_nopriv_wpgdprc_process_action', array(Ajax::getInstance(), 'processAction'));
        add_action('wp_ajax_wpgdprc_process_action', array(Ajax::getInstance(), 'processAction'));
        add_action('update_option_wpgdprc_settings_enable_access_request', array(Action::getInstance(), 'processToggleAccessRequest'));
        Integration::getInstance();
        if (Helper::isEnabled('enable_access_request', 'settings')) {
            add_action('init', array(Action::getInstance(), 'processEnableAccessRequest'));
            add_action('admin_notices', array(Action::getInstance(), 'showNoticesRequestUserData'));
            add_action('wpgdprc_deactivate_access_requests', array(Cron::getInstance(), 'deactivateAccessRequests'));
            add_action('wp_ajax_wpgdprc_process_delete_request', array(Ajax::getInstance(), 'processDeleteRequest'));
            add_shortcode('wpgdprc_access_request_form', array(Shortcode::getInstance(), 'accessRequestForm'));
            if (!wp_next_scheduled('wpgdprc_deactivate_access_requests')) {
                wp_schedule_event(time(), 'hourly', 'wpgdprc_deactivate_access_requests');
```

Figure 4: The Code View

The Code View also has an alternative mode, the code execution trace. This mode uses the MonacoSnippet component, which is a small instance of the Monaco editor that only displays a single range of lines from a file. When these snippets are stacked on top of each other, the user is able to look at only the code that was actually run by the program, cutting out portions of files and functions that the program never actually ran. These ranges are calculated using the line numbers of the function calls from the PHP function call logs.

In both editor components, code is loaded by fetching the file from *Pegasus-Backend*.

The most sophisticated component in *Pegasus-Frontend* is codeFlowGraph, a graph of the execution stack received from the PHP function logs through *Pegasus-Backend* along with the proxy log entry and any callbacks that are found. Each function call is a rectangular node, with arrows between each function that has a caller-callee relationship. The colors of each node are determined via the suspicion level. More suspicious functions appear more red, so they stand out from the others. The entire graph is built as an SVG with the D3.js library, and is both zoomable and panable to allow the user to look at only the part of the graph that interests them. Clicking either a node or an arrow brings up a context menu with information. Clicking on the arrow brings up the parameters that were passed from the caller function to the callee function. Clicking on a node brings up all the information that the program has on that node, including timestamp, script name, function name, line number, definition file, function start, function end, caller, parameter, next function call, suspicion level, time, end, user, ip, request, and response. Clicking a node will also "focus" open editors on it, as explained below.



suspicion level	2.7
next function call	70
id	22670
caller	22669
function end	243
function start	13
definition file	/var/www/html/wp-content/plugins/gdpr142/Includes/Ajax.php
line number	286
function name	processAction
File called from	/var/www/html/wp-includes/class-wp-hook.php
timestamp	2021-03-23T18:33:25.770Z

Figure 5: Example usage of the code flow graph, with suspicion level color

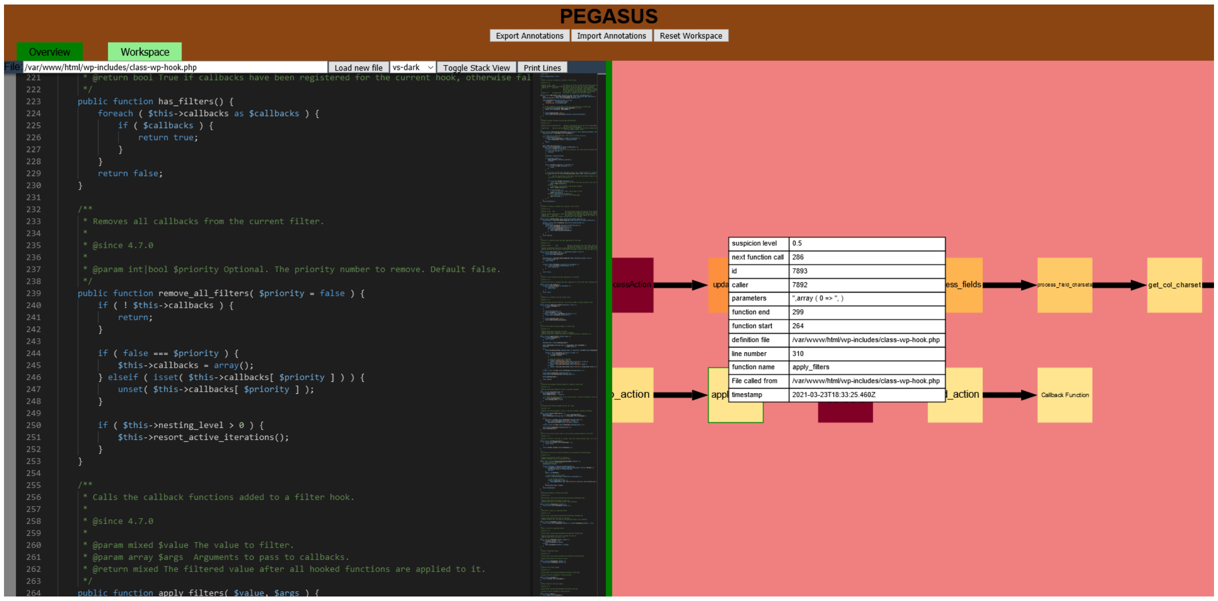
Vue has its own data propagation system, but it is insufficent for PEGASUS's use case. Vue components can only directly communicate with the components adjacent to them on the DOM tree, so interactions between components on different levels of a nested group of components requires a significant amount of boilerplate and repeated code. Because of this, PEGASUS implements its own VUE plugin named the "global data store," which keeps every component in sync and updates subscribed components. In the `main.ts` file that serves as the entry point for the entire project, a global Data object is added to every component, which allows all components to access the same information. This is useful when a large, external data source is required by multiple components, such as the call graph stack, and when components need to interact, such as when a function is selected in the code flow graph and should then be viewed in the editor component.

The global store also has update hooks, which are called whenever a value is changed in order to keep subscribed components up-to-date. When the global data store is created, these hooks are empty, but the ComponentMixin (the mixin from which all components inherit) defines an update function for each value. The Row and Column Container components also have these hooks, and propagate all changes to their children. The hooks are set as soon as the EmptyContainer adds its first component, ensuring that any component will be able to update itself when the global store is changed.

The properties in the global store are:

- **selectedFile:** which lets the code flow graph tell the code view component when to change it's focus.
- **callstack:** which allows the code flow graph to tell the stacked code view component which files, functions, and lines to show.
- **selectedTimestamp:** which allows the overview component to tell the code flow graph which timestamp to request in its fetch from *Pegasus-Backend* when requesting the call stack.
- **annotationList:** which allows every editor to have the same list of components, as well as letting them be edited by the import/export component.

Together, these components and mixins create an interlocking web of functionality that allows a reactive and personalized workspace to be created by the analyst with limited explanation or boilerplate work.



File: /var/www/html/wp-includes/class-wp-hook.php

```
     * @return bool True if callbacks have been registered for the current hook, otherwise fal
     */
    public function has_filters() {
        foreach ( $this->callbacks as $callbacks ) {
            if ( $callbacks ) {
                return true;
            }
        }
        return false;
    }

    /**
     * Removes all callbacks from the current filter.
     *
     * @since 4.7.0
     *
     * @param int|bool $priority Optional. The priority number to remove. Default false.
     */
    public function remove_all_filters( $priority = false ) {
        if ( ! $this->callbacks ) {
            return;
        }

        if ( false === $priority ) {
            $this->callbacks = array();
        } elseif ( isset( $this->callbacks[ $priority ] ) ) {
            unset( $this->callbacks[ $priority ] );
        }

        if ( $this->nesting_level > 0 ) {
            $this->resort_active_iterations();
        }
    }

    /**
     * Calls the callback functions added to a filter hook.
     *
     * @since 4.7.0
     *
     * @param mixed $value The value to filter.
     * @param array $args  Arguments to pass to callbacks.
     * @return mixed The filtered value after all hooked functions are applied to it.
     */
    public function apply_filters( $value, $args ) {
```

suspicion level	0.5
next function call	286
id	7893
caller	7892
parameters	'',array (0 => '',)
function end	299
function start	264
definition file	/var/www/html/wp-includes/class-wp-hook.php
line number	310
function name	apply_filters
File called from	/var/www/html/wp-includes/class-wp-hook.php
timestamp	2021-03-23T18:33:25.460Z

Figure 6: Pegasus Frontend with two widgets sharing data

5 Evaluation

The evaluation of PEGASUS is composed of two parts. First, we evaluate the extent to which the program gets accurate and precise results by gathering metrics on its output. Second, we evaluate the extent to which the program is successfully communicating its results to the user. To do so, we compare the results from PEGASUS and a from a user using not using PEGASUS.

Vuln	Lines in Files	% reduction	Lines in Functions	% reduction	Lines run by program	% reduction	total % reduction	Suspicion Ranking of Error
GDPR Plugin	8668	97.71%	926	89.32%	838	9.50%	90.33%	1 (2.9)
CVE-2021-24182	7016	98.15%	487	93.06%	306	37.17%	95.64%	2 (2.7)
CVE-2021-24183	7016	98.15%	446	93.64%	330	26.01%	95.30%	2 (2.7)

Table 1: Defect Location Experimentation results.

PEGASUS user		Non-PEGASUS user	
KLM time	Video time [15]	KLM time	Video time [16]
31.48s	34.94s	456.18s	859.12s

Table 2: User Experience experiment results.

5.1 Automatic Localization of Defects

The accuracy and precision of the results are measured by two primary metrics, obtained by running the debugger on a set of known WordPress vulnerabilities. Those metrics are the number of lines the program flags, and the suspicion value assigned to the function that contains the line with the actual error. These represent precision and accuracy, respectively; more lines of code mean that the program was not able to narrow down the options as well, and the suspicion heuristic represents how sure the program is that a specific line contains the error.

Our process for obtaining our quantitative data is as follows. First, we identify a known WordPress vulnerability with an error that can be narrowed down to a line number. The vulnerabilities trigger the Single-use Server protection against the Confused Deputy problem, as illustrated in figure 2. We then modify our Single-use Server installation to use any version of WordPress or plugins that are required. Next, the Single-use Server is run, the exploit is triggered either manually or through an exploit script, and then the Single-use Server is shut down. After running, we get the number of lines in the WordPress installation with the command `wc -l wordpress/**/*.php`, which sums the line count for each PHP file in the WordPress codebase.

Then, we launch PEGASUS. From PEGASUS, we get the following data: the number of lines present in all the files run by the program, the number of lines present in all the functions run by the program, and the number of lines present that program actually ran, i.e., the amount in the callstack. These numbers will help us understand how PEGASUS reduces the amount of code an analyst has to consider. These numbers are obtained through a debugging option we placed in PEGASUS. Finally, we can look at the suspicion level for the functions PEGASUS displays and learn the ranking of the function that contains the error. This gets us a metric for precision and accuracy (i.e., the number of lines SUS shows out of the total lines in the project and the ranking of the applicable suspicion level, respectively).

5.1.1 Vulnerability 1: Van Ons GDPR plugin

The first vulnerability we tested is in the Van Ons WP GDPR Compliance WordPress plugin (aka wp-gdpr-compliance) in versions before 1.4.3 [20]. This plugin has an exploitable code defect wherein an incorrect callback is given to an operation, allowing a user to bypass controls, gain elevated permissions, and execute arbitrary SQL code. The specific line is located within the `init()` function of the `WPGDPRC` class in the file `wp-content/plugins/gdpr142/wp-gdpr-compliance.php`. On line 96, the `wp_ajax_nopriv_wpgdprc_process_action` action is incorrectly assigned to the `processAction` function; the `processAction` function assumes that the code has already confirmed the user's privileges, but the action is only called when the user does not have privileges (hence the "nopriv" in the function name).

The code used to perform the exploitation is shown below.

```
import requests
import re
import urllib3
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
url = "https://192.168.122.150/"
session = requests.Session()
wp_home_res = session.get(url, verify=False)
if wp_home_res.status_code is not 200:
    print("GET␣Request␣response␣was␣not␣200,␣aborting")
    exit(1)
ajax_security = None
m = re.search('\"ajaxSecurity\":\"(.+?)\"}', wp_home_res.text)
if m:
    ajax_security = m.group(1)
else:
    print("ERROR:␣Could␣not␣find␣ajax␣security␣token,␣aborting")
    exit(1)
cookies = dict(wp_home_res.cookies)
print(cookies)
wp_role_data = {}
```

```
wp_role_data["action"] = "wpgdprc_process_action"
wp_role_data["security"] = ajax_security
wp_role_data["data"] = "{\"type\":\"save_setting\",\"append\":false,
    \"option\":\"users_can_register\",\"value\":\"      1\"}"
wp_role_res = session.post(url + "wp-admin/admin-ajax.php", data=wp_role_data,
    cookies=cookies, verify=False)
if wp_role_res.status_code is not 200:
    print("POST Request response was not 200, aborting")
    print(wp_role_res.text)
    exit(1)
```

As seen in table 1, when the exploit script is run, PEGASUS correctly narrows the potential locations down to 8 files totaling 8,668 lines. It further narrows to 17 functions totalling 926 lines, and then further narrows to 838 lines that were actually run by the program. This represents a 90.33% reduction in the amount of code that the analyst would need to review to identify the defect. Additionally, the `init()` function containing the error, as referenced above, is labelled as the most suspicious function, with a suspicion level of 2.9 (shown in table 1). These results indicate high precision and accuracy.

5.1.2 Vulnerability 2: Tutor LMS Plugin CVE-2021-24182

The second vulnerability used to evaluate PEGASUS is CVE-2021-24182 [12], which is a known bug in the WordPress plugin Tutor LMS. In this plugin, an AJAX action called `wp_ajax_tutor_quiz_builder_get_answers_by_question` is created so authenticated users can get the answers to questions. This plugin has an exploitable code defect wherein a SQL statement does not have its input sanitized, leading to the potential for a SQL UNION attack. This results in an attacker adding a "UNION" statement to the SQL request string, which results in the program returning more than it is supposed to. The specific line is located within the `tutor_quiz_builder_get_answers_by_question()` function of the `Quiz` class in the file `wp-content/plugins/tutor/classes/Quiz.php`. The unsanitized SQL statement is on line 872 and is used when calling the function `$wpdb->get_row()`.

The exploit is abused on the SUS installation by first connecting to the website normally, signing into a account without permissions, and copying the SUS and WordPress cookies from the browser using the developer tools. Those cookies are then put into a text file representing a request to the affected AJAX endpoint. Finally, a tool called `sqlmap` is used to automatically perform the SQL UNION attack on the server, retrieving all the tables it can. These attacks break the permissions of the user, causing SUS to log the error and allowing PEGASUS to operate on the log files. The sqlmap command used is `sqlmap -r ./tutorexploit.txt --dbms=mysql --technique=U -p question_id --dump`.

The `tutorexploit.txt` file used to perform the exploitation is shown below. The cookies field has been removed for readability.

```
POST /wp-admin/admin-ajax.php HTTP/1.1
Host: https://192.168.122.150
```

```
Content-Length: 96
Accept: */*
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/537.36 ...
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Origin: https://192.168.122.150
Referer: https://192.168.122.150
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: [Cookies]
Connection: close

action=tutor_quiz_builder_get_answers_by_question&question_id=1&question_type=1
```

As seen in Table 1, when the exploit script is run, PEGASUS is able to correctly narrow the potential locations down to 7 files totalling 7,016 lines. It further narrows to 14 functions totalling 487 lines, and then to 306 lines that were actually run by the program. This represents a 95.64% reduction in code. Additionally, the `tutor_quiz_builder_get_question_form()` function that contains the error, as referenced above, is labelled as the second-most suspicious function, with a suspicion level of 2.9 (shown in table 1). These results indicate high precision and accuracy.

5.1.3 Vulnerability 3: Tutor LMS Plugin CVE-2021-24183

The third vulnerability is CVE-2021-24183 [13], which is similar to Vulnerability 2. The same plugin and lack of SQL sanitization leads to another UNION attack, this time on the endpoint `wp_ajax_tutor_quiz_builder_get_question_form`. The specific line for this defect is located within the `tutor_quiz_builder_get_answers_by_question()` function of the `Quiz` class in the file `wp-content/plugins/tutor/classes/Quiz.php`. The unsanitized SQL statement is on line 646 and is used when calling the function `$wpdb->get_row()`.

The `tutorexploit.txt` file used to perform the exploitation is shown below. The cookies field has been removed for readability.

```
POST /wp-admin/admin-ajax.php HTTP/1.1
Host: https://192.168.122.150
Content-Length: 96
Accept: */*
X-Requested-With: XMLHttpRequest
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_4) AppleWebKit/537.36 ...
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
Origin: https://192.168.122.150
Referer: https://192.168.122.150
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: [Cookies]
Connection: close

action=tutor_quiz_builder_get_answers_by_question&question_id=1&question_type=1
```

As seen in Table 1, when the exploit script is run, PEGASUS correctly narrows the potential locations down to 7 unique files totalling 7,016 lines. It to 14 functions totalling 446 lines, and then further narrows to 330 lines that were actually run by the program. This represents a 95.30%

reduction in code. Additionally, the `tutor_quiz_builder_get_answers_by_question()` function that contains the error, as referenced above, is labelled as the second-most suspicious function, with a suspicion level of 2.9 (shown in table 1). These results indicate high precision and accuracy.

5.2 Keyboard-level model and Recordings

To evaluate the effectiveness of the communication, we will create a recording of a developer going through the debugging process with and without PEGASUS. We will apply a Keyboard-level model (KLM) [3] to the recording to reduce bias and noise and create a usage scenario. The two recordings of the debugging process are of a user debugging the Van Ons GDPR plugin vulnerability as described above. In one recording, the user is using PEGASUS; in the other, the user only has simple tools with which to view the logs and source code. The non-Pegasus tools will be text editors like `vim` and command-line tools such as grep and the node REPL. These tools which were chosen because the author is faster with those tools and these tests are designed to give the non-PEGASUS tools as much benefit as possible. These recordings will assume that the user is skilled with the tools they use; familiar with the bug; and can read quickly. These skills will ensure that the main item being timed is the interaction with the tool itself in both cases. From there, a Keyboard-Level Model will be applied to the steps the user took throughout the video in an attempt to measure the actual complexity of the workflow and eliminate personal bias from the recordings.

5.2.1 Recordings

To create our bugfixing videos, we identified the mandatory steps a user would have to take in order to find the vulnerability, even with all our previously stated assumptions. As soon as the code containing the bug is visible, we count the bug as identified. The mandatory steps are for both users: the one using PEGASUS and the one who is not not using PEGASUS.

5.2.2 PEGASUS Workflow

1. Click button for the Resource Guard violation
2. Set up workspace
    (a) Click split pane
    (b) Click code view on panel 1
    (c) Click code flow graph on panel 2
3. On the code flow graph, click the function highlighted as the darkest red
4. See bug in code, which should be visible in the code view
5. Click the annotation zone to the left of the code

6. Write annotation describing bug
7. Export annotation and save it locally

5.2.3 Manual Workflow ("Full" exploration)

1. Get timestamp from Resource Guard log
    - (a) Search Resource Guard log for response code of "-1"
    - (b) Copy timestamp
2. Find Proxy Log entry with matching (or temporally near) timestamp
3. Find PHP `mysqli_query` function with the same (or temportally near) timestamp
    - (a) Translate timestamp to UNIX format in Node REPL
4. Explore `parsed_php_log` file in reverse by searching caller ID
    - (a) Open source code for every file, read each
5. Upon reaching the `do_action` function that calls a dynamic function, switch code flows to the `do_action` flow
6. Advance from the `do_action` function by searching callee ID
7. Open the filename, observe the bug
8. Create a comment describing the bug

5.2.4 Keyboard-level Model

Operation	Symbol	Time (s)
Pointing to a target	p	1.1
Keystroke	k	0.2
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	0.1

Table 3: KLM operations, their times, and their symbols.

Next, we created a KLM [3] based on the recordings to calculate the time taken by the above workflows in a quantifiable and consistent way. While the recorded videos can be affected by the user's knowledge of the program, workflow quirks, and personal working preferences, a KLM is a useful way to approximate how the average expert user would experience a workflow, helping reduce the noise of real recordings. The operations used, and their required time, can be seen in Table 3. This list comes from two sources. The first set of actions are keystrokes, pointing the mouse cursor, and mentally preparing oneself. These are from Allen Newell's paper [3] that proposed the idea of a

KLM. The second set contains more complicated "compound" actions described by Jeff Sauro [17], including filling in a text field and scrolling through text. These are used for simplicity's sake, as many of the actions taken by the user, especially in the non-PEGASUS case wherein Vim is used, can be accomplished in many similar ways. Rather than figure out the most optimal possible use case, these approximations are used for longer text inputs and actions like scrolling with no similar pre-defined operation. Finally, the "b" operation is created to differentiate between mouse clicks and key-presses; it has the same time as to be faithful to the standard KLM model [3].

The created KLMs can be found in the appendix. The PEGASUS user's results are in table 4 and the non-PEGASUS user's results are in table 5.

Overall the KLM demonstrates that using PEGASUS results in a significantly faster debugging experience than manually going through the files. The KLM takes significant liberties with the manual process, assuming that the user is able to start tasks and process information quickly (which explains why the KLM is faster than the actual recording), but the PEGASUS method still takes less than 7% of the time.

In table 2, we show times for both the length of the recording during the bugfixing process and the amount of time calculated via the KLM model. The videos used are linked through their citations in the table.

5.3 Usage Scenario

Our research showed that usage scenarios are a popular way for a data visualization to be evaluated [19], as they do not require the in-person experiments that user studies do. For our usage scenario, we will walk through the process of debugging Vulnerability 1 with PEGASUS and analyze its effectiveness at solving visualization tasks. We have two usage scenarios: the best case, in which the user quickly identifies the bug; and the worst case, in which the bug is one of the lower-rated functions in terms of suspicion.

Scenario 1:

1. the user opens PEGASUS in response to SUS reporting an error
2. the user selects the only resource guard violation: an SQL update query done without permission
3. the user splits the screen into two panes
4. the user opens a code view in the left pane by clicking the "Code view" button or icon
5. the user opens a code flow graph in the right pane by clicking the "Code flow graph" button or icon
6. the user notices a function which is more red than the others
7. the user clicks on the red function, bringing up the code on the code view pane
8. the user can see where the function begins and ends on the code view, and looks through the code to find the bug

9. the user identifies a line with a potential SQL injection attack
10. the user leaves an annotation on that line by clicking the annotation bar to the left of the code view
11. the user exports that annotation as a JSON output for future reference

Scenario 2:

1. the user opens PEGASUS in response to SUS reporting an error
2. the user selects the only resource guard violation: an SQL update query done without permission
3. the user splits the screen into two panes
4. the user opens a code view in the left pane by clicking the "Code view" button or icon
5. the user opens a code flow graph in the right pane by clicking the "Code flow graph" button or icon
6. the user notices four functions which are more red than the others
7. the user selects the most red function by panning the screen to it and clicking on it (this brings it up the code on the code view pane as well)
8. the user reads through the code and determines that only one or two lines are complex enough for a bug
9. the user leaves annotations on those lines to indicate that they are possible, but unlikely, candidates by clicking the annotation margin to the left of the code view
10. the user repeats steps 7-9 for the other four functions
11. after exhausting the highlighted functions, the user moves to the "code execution trace" mode of the code view by clicking the "code execution trace" button on the code view title bar
12. the user reads the entire codeflow line-by line and identifies the error
13. the user leaves an annotation on the line containing the error by clicking the annotation bar to the left of the code view
14. the user exports that annotation as a JSON for future reference

These scenarios illustrate two different use cases, one in which PEGASUS greatly assists the user by focusing them to the correct destination, and one in which the user is initially misled by PEGASUS's guesses and instead has to use PEGASUS's ability to aggregate relevant information to find the bug themselves. In both cases, only a few steps are required from the user, and the user interface allows them to move smoothly between tasks. The main source of redundancy was in the initial setup; it takes three clicks to go from a blank workspace to the split code view and code flow graph, which are usually what the user wants. Additionally, we could have PEGASUS automatically navigate to have the most-suspicious function selected in the code flow graph and the code view as soon as the resource guard violation is selected.

6 Discussion

In this section, we will discuss the limitations of this project and potential future work with PEGASUS.

6.1 Limitations

PEGASUS automatically detects dynamic functions; this is a functionality of WordPress wherein a plugin can define a hook for users to access, allowing WordPress to run the user's custom code on certain events. These functions are not defined as source code in advance, but are dynamically loaded from strings passed through the plugins. PEGASUS has support for these dynamic functions and can track whether bugs occured in them, but does not display the actual text from these dynamic functions. The text can still be viewed as a parameter being passed into the function, but the code view does not show the code.

Many of the Wordpress plugin vulnerabilities used to evaluate PEGASUS are officially registered in the MITRE "Common Vulnerabilities and Exposures" (CVE) database. However, there is no need to restrict our evaluation just to formal CVEs. While this paper was originally written with them in mind, it has come to the author's attention that the vulnerability used to exploit the Van Ons GDPR plugin is not, in fact, a CVE. While the Van Ons GDPR plugin does have an arbitrary code execution CVE from the same version number as our actual exploit [11], the exploit used during our evaluation is actually a completely separate bug. For this reason, we have worded this paper to discuss only "WordPress vulnerabilities," when generally this means CVEs for WordPress plugins. This also shines a spotlight on one of PEGASUS' primary use cases; these novel CVEs through its defect location system, making attribution of exploits much easier for the end user.

6.2 Future Work

The scope of PEGASUS starts with the log files that are exported from the Single-use Server. These log files are generated automatically by SuS, further, the PEGASUS configuration file must point to the location of each file so the program can read them. However, this process involves running several scripts and knowing the location of each of these log files in the fairly large SuS codebase. A future goal for PEGASUS would be to have better tooling and automatically generate and move these files on startup. Further work could explore automatic generation, pre-processing, or even running PEGASUS automatically in the event of an error.

PEGASUS is built entirely on top of the Single-use Server and, as such, can only work on platforms where the Single-use Server can run. While the Single-use Server is theoretically applicable to any server architecture, the only implementation at the time this work started was the WordPress installation. Several aspects of PEGASUS' design take the WordPress environment into account, such as *Pegasus-Backend* accounting for dynamic functions and the automatic filtering of the “/var/www/” path off of filenames. These minor features would be easy to add to other implementations. Future work could help ensure that the small WordPress-specific features are migrated or removed.

PEGASUS’ research goals involved gathering, processing, and communicating data from the Single-use Server. While the goal of communication is aided by solid visual design, it was deemed a non-necessary component of the project. Therefore, aspects such as the color palate, button size, and visual design were not fully explored. However, the impact of visual design is important when it comes to human-computer interaction [10], and cannot be ignored. Future work could update the design and improve aesthetics to assist user understanding and ease-of-use.

The current annotation system is not as user-friendly as it could be. Requiring the user to click an unlabelled grey bar is not very elegant or user-friendly; time constraints prevented much development being done on top of the Monaco editor. Future work could improve the Monaco editor integration, by adding a more full-featured plugin, moving the feature out of the editor and into the PEGASUS HTML, or adding a tutorial to help make it more understandable.

The KLM and user study both revealed some redundancy in PEGASUS’s user experience. Some parts of the debugging process could be sped up fairly quickly, such as by automatically setting up the code view and code flow graph side-by-side, selecting the most suspicious function, and scrolling to the highlighted line all on startup. Additionally, the code flow graph could scroll and pan to center on the selected function or most suspicious function when a shortcut or button is pressed, and the code flow graph could toggle between its views with a shortcut as well. These speedups would improve the user’s comprehension and usability, and thus the user experience.

6.3 Concluding Remarks

Debugging web applications is difficult. It is hard to discover when a code defect has been created, and even if it has been exploited, it can be difficult for a developer to find the location without combing through the entire codebase. Tools such as the Single-use Server are useful for getting information about how and when a defect is exploited. Even then, the amount of data processing required makes the defect location process time-consuming, which costs development time, reduces website up-time, and puts potential victims at risk. Our goal was to determine how accurately and precisely we could locate the defect in a codebase using the SuS logging information, and how well we could communicate that to the end user.

To answer this we developed PEGASUS, which has a log-processing backend and a web-based frontend to allow for easy and quick navigation of the log information and codebase. We used D3.js to develop a visualization that helps the user focus on where the defect is likely to be, as well as a Monaco editor instance to allow the user to view the codebase and annotate lines, creating a simple data output to assist in the debugging process.

PEGASUS performed well in our evaluation, and was able to precisely pinpoint the defects to around 5% of the total lines in the files run by the webapp, which is less than 0.25% of the overall codebase. Additionally, our accuracy rate was 100%, with the actual error appearing in the top 2 most suspicious functions every time. PEGASUS also demonstrated its ability to communicate this data to the user, outperforming non-PEGASUS debugging workflows in both time and complexity. While the testing infrastructure for PEGASUS was a WordPress website, we discuss how the tools it uses could be applied to any webstack that can take advantage of the Single-use Server PHP callstack tracing. From this research, we conclude that the logging information presented by the Single-use Server represents a significant advantage for debugging web applications and that this information can be quickly and reliably presented to the end user.

References

[1] Monther Aldwairi and Hesham H Alsaadi. Flukes: Autonomous log forensics, intelligence and visualization tool. In *International Conference on Future Networks and Distributed Systems*, pages 1–6, 2017.

[2] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. In *International Conference on Software Engineering*, pages 468–479, 2014.

[3] Stuart K. Card, Thomas P. Moran, and Allen Newell. The keystroke-level model for user performance time with interactive systems. *Commun. ACM*, 23(7):396–410, July 1980.

[4] Ionut-Cosmin Cernica, Nirvana Popescu, and Bogdan Tiganoaia. Security evaluation of wordpress backup plugins. In *International Conference on Control Systems and Computer Science (CSCS)*, pages 312–316. IEEE, 2019.

[5] Raymond Cheng, William Scott, Paul Ellenbogen, Jon Howell, Franziska Roesner, Arvind Krishnamurthy, and Thomas Anderson. Radiatus: A shared-nothing server-side web architecture. In *ACM Symposium on Cloud Computing*, pages 237–250, 2016.

[6] Valentina Grigoreanu, Margaret Burnett, Susan Wiedenbeck, Jill Cao, Kyle Rector, and Irwin Kwan. End-user debugging strategies: A sensemaking perspective. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 19(1):1–28, 2012.

[7] Rawiroj Robert Kasemsri. A survey, taxonomy, and analysis of network security visualization techniques. Master's thesis, Georgia State University, Atlanta, GA, December 2006.

[8] Julian P Lanson. Single-use servers: A generalized design for eliminating the confused deputy

problem in networked services. Master's thesis, Worcester Polytechnic Institute, Worcester, MA, May 2020.

[9] Microsoft. Monaco editor. `https://microsoft.github.io/monaco-editor/`, 2021.

[10] Aliaksei Miniukovich and Antonella De Angeli. Computation of interface aesthetics. In *ACM Conference on Human Factors in Computing Systems*, pages 1163–1172, 2015.

[11] MITRE. CVE-2018-19207. Available from MITRE, CVE-ID CVE-2018-19207., 2018.

[12] NIST. CVE-2021-24182. Available from NIST, CVE-ID CVE-2021-24182., 2021.

[13] NIST. CVE-2021-24183. Available from NIST, CVE-ID CVE-2021-24183., 2021.

[14] Bryan Parno, Jonathan M McCune, Dan Wendlandt, David G Andersen, and Adrian Perrig. Clamp: Practical prevention of large-scale data leaks. In *IEEE Symposium on Security and Privacy*, pages 154–169. IEEE, 2009.

[15] Matthew Puentes. Pegasus evaluation recording 1. Posted to youtube.com, 2021.

[16] Matthew Puentes. Pegasus evaluation recording 2. Posted to youtube.com, 2021.

[17] Jeff Sauro. Estimating productivity: Composite operators for keystroke level modeling. In Julie A. Jacko, editor, *Human-Computer Interaction. New Trends*, pages 352–361, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[18] Aneesha Sethi and Gary Wills. Expert-interviews led analysis of EEVi—a model for effective visualization in cyber-security. In *2017 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 1–8. IEEE, 2017.

[19] Diane Staheli, Tamara Yu, R Jordan Crouser, Suresh Damodaran, Kevin Nam, David O'Gwynn, Sean McKenna, and Lane Harrison. Visualization evaluation for cyber security: Trends and future directions. In *Workshop on Visualization for Cyber Security*, pages 49–56, 2014.

[20] Mikey Veenstra. Privilege escalation flaw in WP GDPR compliance plugin exploited in the wild. `https://www.wordfence.com/blog/2018/11/privilege-escalation-flaw-in-wp-gdpr-compliance-plugin-exploited-in-the-wild/`, 2018.

[21] Wordpress. The wordpress codebase, 2019. `https://make.wordpress.org/core/handbook/contribute/codebase/`, Last accessed on 2020-10-27.

[22] Yongzheng Wu, Roland HC Yap, and Felix Halim. Visualizing windows system traces. In *International Symposium on Software Visualization*, pages 123–132, 2010.

A Appendix

Description of step	Required Operations	Total time (s)
Click on ”Select Violation” button	MPBB	1.3
Click on ”Split into Rows” button	MPBB	1.3
Click on ”Add Code View Widget” button	MPBB	1.3
Click on ”Add Code Flow Graph” button	MPBB	1.3
Click on most red node in Code Flow Graph	MPBB	1.3
Scroll until you see the highlighted line	SM	5.31
Click on annotation sidebar	MPBB	1.3
Click on text field on annotation popup	MPBB	1.3
Type comment	TT	2.32
Click on ”submit” button on annotation popup	MPBB	1.3
Click on ”Export annotations” button	MPBB	1.3
Total		31.48

Table 4: KLM simulation of PEGASUS workflow.

Description of step	Required Operations	Total time (s)
run tmux command	MK[tmux]	2.15
split tmux window	MKK[B%]	1.95
navigate to log directory	MTT	3.67
switch to source code tmux pane	MKKK	1.95
navigate to source code directory	MTT	3.67
switch to log tmux pane	MKKK	1.95
open resource guard log in Vim	MTT	3.67
search for -1	MK[/-1]K	2.15
copy timestamp	PBB	1.3
exit vim	MK[:q]	1.75
open node REPL	MK[node]K	2.35
convert timestamp format	MTT	3.67
exit node REPL	MKK	1.75
Open proxy log in vim	MTT	1.35
Search for timestamp	MKKK	1.95
copy proxy timestamp	PBB	1.3
exit vim	MK[:q]	1.75
open node REPL	MK[node]K	2.35

convert timestamp format	MTT	3.67
exit node REPL	MKK	1.75
open PHP log in vim	MTT	3.67
search for closest mysql_query function	MTT	3.67
copy caller ID	PBB	1.3
search for caller ID	MTT	3.67
copy definition file	PBB	1.3
switch to source code tmux pane	MKKK	1.95
open source code in vim	MTT	3.67
search source code for function	MTT	3.67
scroll through and read code	MscrollM	6.66
exit vim	MK[:q]	1.75
switch to log tmux pane	MKKK	1.95
repeat above 9 steps for 12 functions		311.04
copy action name from do_action parameters	PBB	1.3
search for action name	MTT	3.67
search for corresponding add_action function	MTT	3.67
copy definition file	PBB	1.3
switch to source code tmux pane	MKKK	1.95
open source code in vim	MTT	3.67
search source code for function	MTT	3.67
scroll through and read code	MscrollM	6.66
exit vim	MK[:q]	1.75
switch to log tmux pane	MKKK	1.95
copy caller ID	PBB	1.3
search for caller ID	MTT	3.67
copy definition file	PBB	1.3
switch to source code tmux pane	MKKK	1.95
open source code in vim	MTT	3.67
search source code for function	MTT	3.67
scroll through and read code until bug is found	MscrollM	6.66
Enter insert mode in vim	MKK	1.75
Leave comment detailing bug	MTT	3.67
save and exit vim	MK[:wq]K	2.15
Total		456.18

Table 5: KLM simulation of Manual workflow.