Secure Software Brownfield Engineering – Sequence Diagram Identification

Aspen Olmsted
School of Computer Science and Data Science
Wentworth Institute of Technology
Boston, MA 02115
email: olmsteda@wit.edu

Abstract— **The process of securing existing "brownfield" software systems becomes challenging when trying to identify and mitigate vulnerabilities in complex and often undocumented codebases. The paper investigates the essential requirement for improved program execution flow comprehension in legacy PHP applications to support secure software development. The proposed solution utilizes the trace functionality of program execution tracing through the PHP extension to obtain detailed execution paths dynamically. The methodology generates complete UML Sequence Diagrams through automated processing of program execution trace logs. These diagrams present object and function interactions through visual representations, which developers and security analysts use as essential tools. The sequence diagrams provide a straightforward, high-level view of runtime operations, which enhances code understanding and reveals concealed dependencies and security-critical control paths. The automated visualization system helps security professionals detect potential attack vectors, verify the implementation of security controls, and identify insecure data handling practices. The research demonstrates how a debugging tool can be leveraged as a security enhancement tool for brownfield environments, enabling developers to identify vulnerabilities more efficiently without relying on manual code reviews or architectural documentation. This method offers a practical solution to enhance the security posture of legacy PHP applications.**

Keywords- cyber-security; software engineering; secure software development.

I. INTRODUCTION

In the realm of software development, there is a distinction between developing greenfield systems and maintaining and advancing existing "brownfield " applications. When it comes to greenfield projects, there's the advantage of integrating up-to-date security measures from the start. However, brownfield systems, which comprise the majority of deployed software, pose a significant challenge. These older applications, developed over years or even decades, often lack security protocols, have inadequate documentation, and carry a burden of technical debt. Businesses depend on them for their operations, but their nature and lack of clarity make them vulnerable to security risks. The process of managing and addressing these vulnerabilities in established settings proves to be quite challenging because it needs strategies that combine traditional practices with modern security needs.

The primary challenge in securing software lies in its inherently black-box nature - a term used to describe its complex and opaque structure that is difficult to comprehend from the outside perspective alone. Developers and security analysts tasked with ensuring the security of these systems often face obstacles due to the lack of up-to-date information about how the software operates. The original developers may have moved on to other projects or roles, and the design documents might have also changed. Even when missing altogether, the sheer size of the codebase can be overwhelming to navigate efficiently. The system's lack of transparency creates significant difficulties for users in detecting control pathways within the codebase and tracking data movement across software components. The system's lack of transparency creates problems for both identifying vulnerabilities that allow harmful input to enter the system and detecting accidental mishandling of sensitive information.

Traditional tools for analysis may highlight problems but often yield numerous incorrect alerts or struggle to understand the intricate context of older code bases. On the one hand, manual code inspections are comprehensive. It can be too time-consuming and costly for established applications. Dynamic analysis—observing how a system behaves during operation—provides an approach to grasping real-time attributes. However, it frequently lacks a deep understanding of function calls and object interactions, which is necessary for accurately pinpointing vulnerabilities.

This article explores the world of existing PHP applications that have been around for a while and have undergone numerous changes and updates over time, often without prioritizing security from the outset. Due to PHP's flexibility and popularity in the online world, these applications have frequently lacked a security-first approach. An immediate requirement arises for widely applicable strategies to enhance the security posture of these yet vulnerable programs.

In our study, we propose a practical method to enhance software development methods in existing PHP environments by automatically generating UML Sequence Diagrams from XDebug trace logs. XDebug is an open-source tool used by developers for debugging and profiling PHP code. The robust tracing capability of XDebug primarily serves for debugging and performance evaluation, but it also provides an opportunity for security assessment. XDebug enables the capture of runtime information about function calls, method invocations, and variable assignments, which provides visibility into program execution behavior.

Our goal is to repurpose this known developer tool to offer an approach for comprehending intricate legacy code with minimal overhead and maximum effectiveness.

Our approach includes a series of steps to achieve our goal efficiently. Firstly, we start by activating an XDebug trace for scenarios or attack situations in the existing PHP system under consideration, creating logs of the exact sequence of actions taken. Secondly, we programmatically process these logs to extract essential details, such as, function names, class methods, arguments, return values, and execution order. We transform the

extracted data into a format that works for creating UML Sequence Diagrams. The visual representations illustrate how individual objects and functions interact with each other over time, demonstrating the control flow and data movement within the software application.

There are many benefits of utilizing reverse-engineered UML Sequence Diagrams in the realm of software engineering. These diagrams play a role in enhancing the understanding of code structures. This becomes especially valuable for developers or security experts who face deciphering intricate legacy codebases, as the sequence diagram provides a holistic overview of how various components collaborate to accomplish specific functionalities. This visual representation simplifies comprehension compared to sifting through lines of undocumented code. Furthermore, the utility of diagrams extends to revealing concealed dependencies and unforeseen interactions within the system. In systems that have undergone development or modifications (brownfield systems), functions could interact with each other in unexpected ways, or data could pass through unforeseen middle steps or components not easily recognizable at first glance. Sequence diagrams bring clarity to these hidden connections by spelling out the relationships, for a deeper examination of possible repercussions or unintentional data disclosures.

Essentially, from a security standpoint, these diagrams point out control pathways. By showing the order of actions, analysts can easily identify areas of attack, such as, points where user inputs are handled, where outside data is used, or where essential tasks are performed. They can assist in tracking how unreliable data moves, from where it enters to processing steps, exposing spots for injections or flaws, in deserialization security. Additionally, sequence diagrams help confirm that security measures are properly implemented. For example, a person can visually verify whether authentication checks are executed in real-time, if input validation processes are regularly utilized, or if authorization determinations are made before accessing resources. This automated display enables security teams to conduct efficient security evaluations, reducing the need for tedious manual techniques.

The primary objective of this study is to demonstrate that utilizing debugging tools for security analysis is not only feasible but also highly beneficial. By converting execution data into a visual display format, we create a valuable tool that can be integrated into the secure development process of legacy applications. This strategy provides a solution for companies facing security issues in their systems, enabling them to identify and resolve vulnerabilities more efficiently without incurring significant costs for re-documentation or re-engineering efforts. Our research suggests that this method offers a reliable way to enhance the security of PHP programs, ultimately contributing to a safer online environment. The paper is organized as follows. Section II describes the related work and the limitations of current methods. Section III describes a motivating example for our work. Section IV discusses the implementation of our parser. Section V discusses the creation of sequence diagrams. We conclude and discuss future work in Section VI.

II. RELATED WORK

Secure software engineering has made substantial progress through greenfield development, which enables security integration at the beginning of software development. The distinctive obstacles of brownfield systems require separate attention. This section examines relevant literature on secure software development, with special attention to research that addresses security integration in existing codebases and the application of dynamic analysis and visualization techniques.

A foundational aspect of secure software engineering is the proactive integration of security considerations throughout the software development lifecycle. Aspen Olmsted's seminal work, "Security Driven Software Development" [1], provides a comprehensive framework for embedding security into every phase of development, from requirements gathering to deployment and maintenance. This book emphasizes the importance of a security-first mindset and offers strategies for identifying and mitigating risks early. While primarily focused on new development, the principles outlined by Olmsted, such as threat modeling and secure coding practices, are equally relevant to brownfield remediation efforts. Our proposed methodology, which aims to improve understanding of existing brownfield code, directly supports the application of such security principles by making the implicit explicit.

The paper by Olmsted titled "Secure software development through non-functional requirements modeling" [2] expands on the significance of early security integration by demonstrating how Non-Functional Requirements (NFRs) serve as essential elements for software security. The paper indicates that security requirements should be treated as an NFR, which should be explicitly modeled during the initial development phase.

The precision and verifiability of security requirements can be enhanced through the use of formal specification languages such as, Object Constraint Language (OCL) and UML stereotypes in this context [3]. For brownfield systems, where NFRs may not have been formally captured during initial development, our approach of generating UML Sequence Diagrams helps in reverse-engineering the system's behavior. By visualizing execution flows, it becomes possible to infer how security-related NFRs (e.g., access control, input validation) are currently being handled, or where they are conspicuously absent. The analysis results will guide the redefinition of security NFRs and direct the remediation process.

The paper "Secure Software Development–Models, Tools, Architectures and Algorithms" by Olmsted [3] presents a comprehensive overview of the various elements necessary for developing secure software systems. This paper discusses multiple models for security, the tools that aid in analysis, architectural considerations for building secure systems, and the algorithms underlying security mechanisms. Our research aligns with this broader vision by introducing a practical tool-based approach (leveraging XDebug) to generate a specific model (UML Sequence Diagrams) that aids in understanding the architecture and behavior of brownfield PHP applications. The generated diagrams serve as essential inputs for security models and algorithm applications, helping analysts detect hidden vulnerabilities in complex legacy code by tracing data and control flow.

While existing literature extensively covers secure software development, a gap remains in practical, low-overhead methods tailored explicitly for thoroughly understanding the runtime behavior of brownfield applications for security purposes. Static analysis tools (e.g., SAST solutions) are effective at identifying patterns of vulnerabilities but often struggle with context and produce false positives in complex legacy code. The interaction of Dynamic Application Security Testing (DAST) tools with running applications reveals vulnerabilities, but it operates at a higher abstraction level than XDebug traces provide function-level details. Our solution enhances existing tools by creating detailed visual execution flow maps, which aid in manual security auditing, threat modeling, and vulnerability impact assessment for complex brownfield PHP applications. The re-purposing of XDebug for this task provides a unique advantage because it uses a widely available and familiar developer tool, which reduces the learning curve and integration overhead for teams working with legacy PHP systems.

III. MOTIVATING EXAMPLE

Our proposed methodology demonstrates practical utility through an example that focuses on SuiteCRM, an open-source Customer Relationship Management (CRM) system widely used by many organizations. The open-source PHP application SuiteCRM represents an excellent brownfield example, as it contains extensive complexity from multiple years of development, without complete modern architectural documentation. The complex nature of SuiteCRM's functionalities makes it challenging for new developers and security auditors to understand its security aspects. Our method of creating UML Sequence Diagrams from XDebug traces enables effective business process reverse-engineering, which improves code understanding and security analysis capabilities.

The following common user scenarios in SuiteCRM demonstrate how program traces reveal their execution flows:

1. Scenario 1: User Creates Contact

- User Action: A sales representative navigates to the "Contacts" module, fills in various contact details (e.g., name, email, phone number, address), and submits the form to save the new contact record.
- Trace Insight and Security Relevance: When XDebug tracing is enabled during this operation, the generated trace log meticulously records every function call, method invocation, and file inclusion that occurs from the moment the form submission is processed. This includes the initial handling of the HTTP POST request, validation routines, and, critically, the data persistence logic. The trace would capture calls to files such as, modules/Contacts/Save.php, revealing the sequence of operations involved in taking the submitted data and committing it to the database.
- A detailed analysis of the sequence diagram derived from this trace shows:
- Input Handling: How the raw form data is received and sanitized (or not) before processing. This is crucial for identifying potential Cross-Site Scripting (XSS) or SQL Injection vulnerabilities if input validation is insufficient or bypassed.
- Data Flow: The path of sensitive contact information (e.g., email addresses, personal details) as it moves from the web form, through various PHP functions, and ultimately to the database. This helps in understanding where data might be exposed or mishandled.
- Database Interaction: The specific functions responsible for constructing and executing SQL queries for insertion into the contacts table. Insecure practices like direct string concatenation for SQL queries would be immediately apparent, highlighting SQL injection risks.
- Workflow Triggers: If the creation of a contact triggers other business logic (e.g., sending a welcome email, updating related accounts, or initiating a workflow), the trace would show the invocation of these subsequent functions. This helps in understanding the full impact of a contact creation operation and identifying any security implications of these cascading actions (e.g., unauthorized email sending).
- Access Control: The diagram could reveal where authorization checks are performed (or omitted) before data is saved, indicating potential Insecure Direct Object Reference (IDOR) or unauthorized data modification vulnerabilities if a user can manipulate data they shouldn't.

2. Scenario 2: User Schedules a Meeting

- User Action: A user accesses the "Meetings" module, enters details such as the meeting subject, time, date, duration, and invites participants (e.g., other users, contacts, leads), then saves the meeting record.
- Trace Insight and Security Relevance: Tracing this scenario would provide a rich sequence of interactions involving the logical meeting module and its dependencies. The trace illustrates how meeting details are processed, how participants are associated, and how notifications may be generated.
- The resulting sequence diagram would be invaluable for:
- Participant Management: Understanding how participants are linked to the meeting. This is critical for assessing potential information leakage (e.g., if a user can view participants they shouldn't) or unauthorized access to meeting details.
- Cross-Module Interactions: Visualizing the calls to linked modules, such as, Users and Contacts, to retrieve participant information. This helps identify potential privilege escalation paths if the

system implicitly trusts data retrieved from these modules without proper revalidation.

- Calendar Integration: If the meeting scheduling integrates with an internal calendar or external service, the trace would expose the functions responsible for these interactions. This enables security analysis of data exchanged with external systems.
- Notification Mechanisms: Tracing the functions responsible for sending meeting invitations or reminders. This can reveal vulnerabilities related to email spoofing, content injections in notifications, or denial-of-service if the notification system can be abused.
- Time and Date Handling: How time and date inputs are processed and stored. Incorrect handling of time zones or date formats can lead to logical flaws or even a denial-of-service attack if parsing errors are not handled gracefully.

3. Scenario 3: User Creates an Invoice

- User Action: An accountant generates a new invoice through the "Invoices" module, associating it with a specific client (Account), adding various products or services, specifying quantities and prices, and saving the invoice.
- Trace Insight and Security Relevance: This scenario is particularly sensitive due to its financial implications. The XDebug trace would capture the complex interactions involved in creating invoice entries, calculating totals, and establishing relationships between Accounts, Products, and Invoices modules.
- The sequence diagram would reveal:
- Financial Calculation Logic: The precise functions involved in calculating line item totals, taxes, and the grand total of the invoice. This is paramount for identifying potential manipulation vulnerabilities (e.g., rounding errors, incorrect tax calculations, or unauthorized price modifications) that could lead to financial discrepancies.
- Relationship Management: How the invoice is linked to an Account (client) and Products. This helps in understanding access control mechanisms for financial data and preventing unauthorized association of invoices with incorrect clients or products.
- Data Integrity: Tracing the flow of product quantities, prices, and client details into the invoice. Any points where these values are not adequately validated or where they could be tampered with before persistence would be highlighted.
- State Transitions: If an invoice goes through different states (e.g., Draft, Pending, Paid), the trace shows the functions responsible for these state changes, allowing for analysis of potential unauthorized state transitions.
- Reporting and Export: If invoice creation triggers the generation of a PDF or an export to an accounting system, the trace would expose the functions handling this, allowing for security review of data serialization and external communication.

In each of these scenarios, the automatically generated UML Sequence Diagrams provide a visual roadmap of the application's runtime behavior. This "living documentation" is far more accurate and up-to-date than static, manually created diagrams, which often become obsolete as the codebase evolves. For brownfield applications like SuiteCRM, these diagrams transform opaque execution paths into transparent, analyzable flows, significantly reducing the time and effort required for security auditing, vulnerability discovery, and targeted remediation. They empower security professionals and developers to ask precise questions about data handling, access control, and business logic, ultimately leading to a more secure and resilient system.

IV. PARSER IMPLEMENTATION

The core of our methodology lies in the ability to accurately parse and interpret the detailed trace logs generated by XDebug. This section describes the implementation of our parser developed in Java, designed to transform the raw, verbose XDebug output into a structured, actionable format suitable for subsequent UML Sequence Diagram generation.

XDebug trace files, typically in the .xt format, contain a chronological record of every function call, method invocation, file inclusion, and variable assignment during a PHP script's execution. While incredibly rich in detail, their raw format is not directly consumable by UML diagramming tools. Our Java parser addresses this by extracting salient information and organizing it into a programmatic representation that captures the essential elements of a sequence diagram: lifelines (objects/functions), messages (method calls), and their temporal order.

1. XDebug Trace File Format Overview

Before detailing the parser's design, it's essential to understand the structure of XDebug trace files. XDebug offers several trace formats, but the most common and detailed is the "computer readable" format (format 1). Each line in this format represents an event (e.g., function entry, function exit, include, require, eval) and contains a series of tab-separated fields. Key fields include:

- Level: The nesting level of the function call.
- Function Number: A unique identifier for the function call instance.
- Type: Indicates the event type (e.g., 0 for function call, 1 for function return, 2 for include, 3 for require, 4 for eval).
- Function Name: The name of the function or method being called.
- File Name: The PHP file where the function call originated.
- Line Number: The line number within the file.

- Time: Timestamp of the event.
- Memory: Memory usage at the time of the event.
- Arguments: A representation of the arguments passed to the function (if configured to be included).

2. Parser Design and Architecture

Our Java parser is designed as a modular component, following a typical parsing pipeline: reading, lexical analysis, syntactic analysis, and data model construction.

3. File Reading and Line-by-Line Processing:

The parser begins by reading the XDebug trace file line by line. Given that trace files can be very large (hundreds of megabytes for complex operations), an efficient line-by-line reading mechanism is crucial to avoid excessive memory consumption. Java's BufferedReader is employed for this purpose.

4. Lexical Analysis (Tokenization):

Each line read from the trace file undergoes lexical analysis. Since the fields are tab-separated, a simple String.split("\t") operation is sufficient to break down each line into its constituent tokens. Robust error handling is incorporated to manage malformed lines or unexpected field counts, preventing parser crashes due to corrupted trace data.

5. Syntactic Analysis and Event Interpretation:

After tokenization, the parser performs syntactic analysis by interpreting the meaning of each token based on its position and the event Type field. A switch statement or a strategy pattern can be used to handle different event types (0 for call, 1 for return, etc.).

Function/Method Calls (Type 0): When a function call event is encountered, the parser extracts the function name, the originating file and line number, and the call level. This information is used to identify the "caller" and "callee" in the sequence. The Function Number is critical for matching function calls with their corresponding returns.

Function/Method Returns (Type 1): Upon encountering a function return event, the parser uses the Function Number to locate the corresponding outstanding function call. This pairing is essential for determining the duration of a call and for correctly nesting messages in the sequence diagram.

Includes/Requires (Type 2, 3): These events indicate file inclusions. While not direct messages in a UML Sequence Diagram, they are important for understanding the context and dependencies within the PHP application. The parser can record these events to provide additional context or to help in identifying the "lifeline" associated with the executed code.

6. Data Model Construction:

The most critical phase is the construction of an in-memory data model that represents the sequence of interactions. We define several Java classes to represent the elements of a UML Sequence Diagram:

7. SequenceDiagram: The top-level class representing the entire diagram, containing a list of lifelines and messages.

Lifeline: Represents an object or function participating in the sequence. For PHP, this typically maps to a class name, an object instance, or a global function. The parser dynamically creates lifelines as new, unique function or method owners are encountered.

Message: Represents a communication between two lifelines. Key attributes include:

- sender: The Lifeline initiating the message.
- receiver: The Lifeline receiving the message.
- methodName: The name of the function/method being called.
- callLevel: The nesting depth of the call.
- startTime: Timestamp of the call.
- endTime: Timestamp of the return.
- arguments: (Optional) Parsed arguments.
- returnValue: (Optional) Parsed return value.
- messageType: (e.g., synchronous call, return).

The parser maintains a stack-like structure (e.g., a Deque or Stack in Java) to keep track of current active function calls. When a Type 0 event (call) occurs, a new Message object is created and pushed onto the stack. When a Type 1 event (return) occurs, the corresponding Message is popped, its endTime is set, and it is added to the Sequence Diagram's list of messages. This stack-based approach correctly handles nested function calls and ensures the proper temporal ordering of messages.

V. UML SEQUENCE DIAGRAM GENERATION

Once the XDebug trace data has been successfully parsed into our structured Java data model (comprising Sequence Diagram, Lifeline, and Message objects), the next step is to translate this model into a visual UML Sequence Diagram. For this purpose, we leverage PlantUML, a powerful open-source tool that allows users to create UML diagrams using a simple, human-readable text description.

The choice of PlantUML offers several significant advantages:

- Text-Based Definition: Diagrams are defined in plain text, making them easy to generate programmatically, version-controlled, and collaboratively worked on. This aligns well with automated generation from trace files, as our Java parser can directly output the PlantUML syntax.
- Ease of Integration: PlantUML can be integrated into various environments and workflows. The generated text file can be rendered into images (PNG, SVG) or other formats using the PlantUML command-line tool, a dedicated server, or IDE plugins.
- Flexibility and Expressiveness: PlantUML supports a wide range of UML diagram types, including Sequence Diagrams, with rich features for actors, participants, messages, activation bars, loops, conditionals, and notes, allowing for detailed and expressive visualizations.

Our Java parser, after constructing the Sequence Diagram object, includes a component responsible for generating the PlantUML syntax. This component iterates through the Lifeline and Message objects in the data model and translates them into PlantUML's specific syntax.

- Participants/Lifelines: Each unique Lifeline object identified during parsing (e.g., a class name like

ContactService, DatabaseHandler, or a generic Application for global functions) is declared as a participant in PlantUML using keywords like participant, actor, or boundary.

- Messages: Each Message object is translated into a PlantUML message arrow. The sender and receiver lifelines determine the source and target of the arrow, and the methodName becomes the message label. Activation bars are automatically handled by PlantUML when -> (call) and <- (return) messages are used.
- Nesting and Call Levels: The callLevel attribute of our Message objects is crucial for correctly representing nested calls and activation bars. PlantUML inherently handles nesting through the sequence of -> and <- messages, but explicit activate and deactivate keywords can be used for finer control.
- Conditional Logic and Loops: While XDebug traces capture the executed path, they don't directly provide information about if conditions or for loops that weren't taken. However, for executed loops or branches, the repeated messages or specific sequences can be grouped using PlantUML's loop or alt/else constructs, which can be inferred from patterns in the trace or added manually for clarity.

The output of this component is a plain text file (e.g., diagram.puml) containing the PlantUML definition. This file can then be fed into a PlantUML renderer to produce the final visual sequence diagram, providing an intuitive and accurate representation of the brownfield application's runtime behavior. This automated generation significantly reduces the manual effort traditionally associated with creating and maintaining such diagrams, making them a practical tool for security analysis and code comprehension.

VI. CONCLUSION AND FUTURE WORK

Future work should address multiple challenges starting with the issue of large file sizes according to our research findings. XDebug trace files tend to expand their size when complex operations or scripts run for extended periods. The parser needs both memory efficiency and the ability to handle files which exceed RAM capacity.

The trace data produced by XDebug includes all function arguments in its output. The process of interpreting complex PHP data structures (arrays, objects) in Java requires advanced logic to convert them into meaningful representations. The initial development should begin with basic data types before moving on to raw string logging of arguments.

The performance speed becomes vital when handling massive trace files. String manipulation using StringBuilder alongside data structure efficiency and object creation minimization will substantially boost performance.

Real-world trace files may contain corrupted lines or unexpected formats because of system crashes or misconfigurations. The parser needs to maintain robustness by either skipping malformed entries or logging warnings so it can prevent crashes.

Converting PHP dynamic elements like global functions and anonymous functions and closures into standard UML lifelines and messages requires thorough analysis. The system should treat global tasks as part of a basic "Application" lifeline and each object should receive its own lifeline based on its class name.

Our solution transforms XDebug trace data through Java parsing followed by PlantUML diagram generation to create structured programmatic models which become visual sequence diagrams. The generated model functions as the direct input source for any UML diagramming library or tool which produces valuable visual sequence diagrams to analyze and secure brownfield PHP applications. The parser architecture allows future extensions for new XDebug features while providing flexibility for processing various trace formats.

REFERENCES

[1] A. Olmsted, Security-Driven Software Development: Learn to analyze and mitigate risks in your software projects, Birmingham, UK: Packt Publishing, 2024.

[2] A. Olmsted, "Secure software development through non-functional requirements modeling," in *Proceedings of the 2010 International Conference on Software Engineering Research and Practice (SERP)*, 2010.

[3] A. Olmsted, "Secure Software Development–Models, Tools, Architectures and Algorithms," *Journal of Software Engineering and Applications,* pp. 743-750, 2012.