Comprehensive Analysis of Blind SQL Injection Vulnerabilities in PortSwigger Labs

Executive Summary

This report details the methodologies and findings from a series of penetration tests conducted on PortSwigger's Web Security Academy labs, specifically focusing on various forms of Blind SQL Injection (SQLi) vulnerabilities. The objective was to systematically identify, exploit, and document these vulnerabilities, providing insights into their impact and the techniques employed for their exploitation. This analysis is presented from the perspective of a seasoned penetration tester, emphasizing practical approaches and the thought process involved in navigating complex SQLi scenarios.

Introduction to Blind SQL Injection

Blind SQL Injection is a type of SQLi attack where the attacker does not receive direct feedback from the database regarding the results of their injected queries. Instead, the attacker infers information about the database structure or data by observing the application's behavior or response times. This often involves sending carefully crafted queries that cause a subtle, observable difference in the application's response, such as a change in a boolean condition or a time delay. Exploiting blind SQLi requires a methodical approach, often leveraging automated tools or custom scripts due to the iterative nature of data extraction.

Lab 1: Blind SQL Injection with Conditional Responses

Objective

This lab presented a scenario where a web application exhibited a welcome-back message based on user interaction, but direct SQLi attempts in visible input fields yielded no discernible changes. The primary objective was to identify a blind SQLi vulnerability and subsequently extract sensitive information, specifically an administrator's username and password, by observing conditional responses.

Methodology

Initial reconnaissance involved probing various input parameters for SQLi vulnerabilities. Standard SQLi payloads, such as a single quote (`'`), were injected into common user-controlled fields. While the initial attempts on the visible category input were unsuccessful, further investigation using an intercepting proxy (Burp Suite) revealed a `TrackingId` cookie parameter within the HTTP requests. This parameter became the focal point of the subsequent testing.

Upon injecting a single quote (`'`) into the `TrackingId` cookie, a notable change in the application's behavior was observed: the

welcome-back message, which was previously present, disappeared. This immediate and observable change strongly indicated a potential SQLi vulnerability. To confirm this, a SQL comment (`--` or `#`) was appended to the payload, which restored the welcome-back message, thus confirming the presence of a blind SQLi vulnerability within the `TrackingId` cookie parameter. This behavior is characteristic of a boolean-based blind SQLi, where the application's response changes based on the truthiness of the injected condition.

With the vulnerability confirmed, the next phase involved enumerating database information. The primary target was to ascertain the existence of an `administrator` user. This was achieved by crafting a series of conditional SQL queries designed to return a true or false response, manifested by the presence or absence of the welcome-back message. For instance, a payload similar to `TrackingId=' OR (SELECT 'a' FROM users WHERE username='administrator')='a'--` would be used. If the `administrator` user existed, the query would evaluate to true, and the welcome-back

message would appear. This iterative process allowed for the confirmation of the `administrator` user's presence.

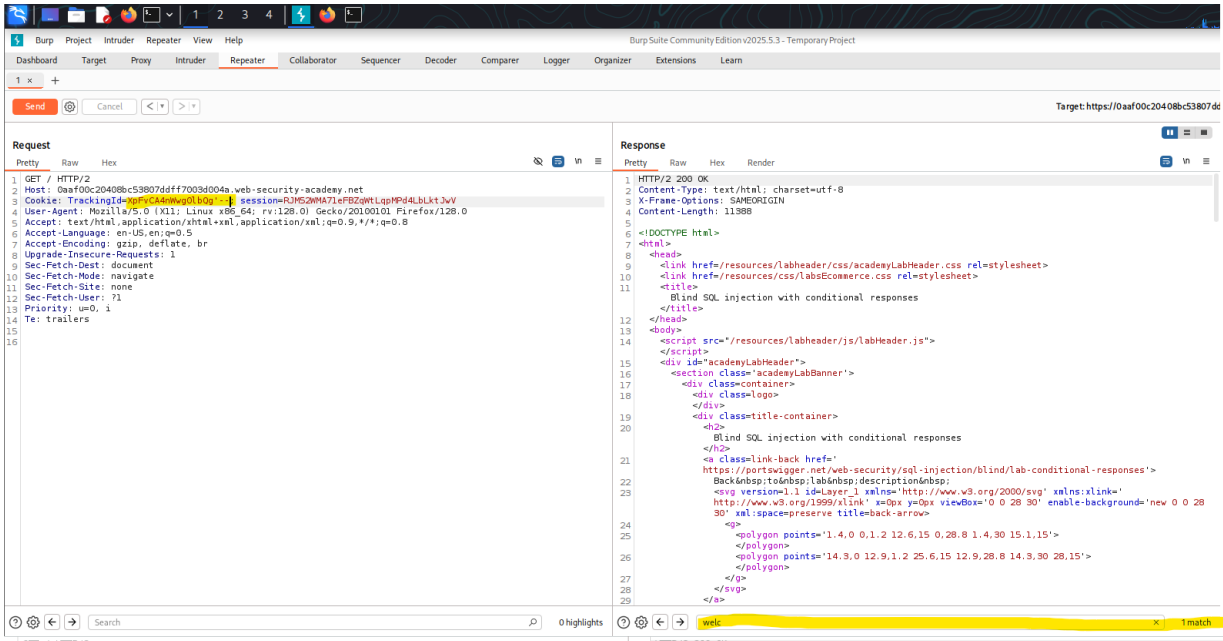
The subsequent step focused on determining the length of the `administrator`'s password. This was accomplished by leveraging the `LENGTH()` SQL function in conjunction with conditional responses. Payloads such as `TrackingId=' OR (SELECT LENGTH(password) FROM users WHERE username='administrator') > 10--` were employed, incrementing the length value until the welcome-back message consistently appeared, thereby revealing the exact password length. Through this systematic approach, the password length was determined to be 20 characters.

Finally, the password itself was extracted character by character. This required a brute-force approach, where each character of the password was tested against a predefined set of alphanumeric characters. For each character position, a payload similar to `TrackingId=' OR (SELECT SUBSTRING(password, 1, 1) FROM users WHERE username='administrator') = 'a'--` would be constructed. The character (`'a'`) would be iterated through the character set until the welcome-back message appeared, indicating a correct character. This process was repeated for all 20 characters of the password. Due to the highly iterative nature of this process, automated tools or custom scripts are typically employed to expedite the extraction. In this specific lab, the password was successfully identified as `bunh3op0tgbwa8g5mcju`.

Impact

This blind SQLi vulnerability allowed for the complete enumeration of user credentials, specifically the administrator's username and password. Such an exploit can lead to unauthorized access, data exfiltration, privilege escalation, and potentially full control over the affected application and underlying database.

Screen shots as a poc


Burp Project Intruder Repeater View Help
Burp Suite Community Edition v2025.5.3 - Temporary Project
Dashboard Target Proxy Intruder Repeater Collaborator Sequencer Decoder Comparer Logger Organizer Extensions Learn
Send Cancel
Target: https://0aaf00c20408bc53807dd
Request
Pretty Raw Hex
GET / HTTP/2
Host: 0aaf00c20408bc53807ddff7003d004a.web-security-academy.net
Cookie: TrackingId=XpFvCA4nWwgOlbQg'--; session=RJM52WMA71eFBZqWtLqpMPd4LbLktJvV
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Priority: u=0, i
Te: trailers
Response
Pretty Raw Hex Render
HTTP/2 200 OK
Content-Type: text/html; charset=utf-8
X-Frame-Options: SAMEORIGIN
Content-Length: 11388
<!DOCTYPE html>
<html>
<head>
<link href=/resources/labheader/css/academyLabHeader.css rel=stylesheet>
<link href=/resources/css/labsEcommerce.css rel=stylesheet>
<title>
Blind SQL injection with conditional responses
</title>
</head>
<body>
<script src="/resources/labheader/js/labHeader.js">
</script>

<section class='academyLabBanner'>

<h2>
Blind SQL injection with conditional responses
</h2>
<a class=link-back href='
https://portswigger.net/web-security/sql-injection/blind/lab-conditional-responses'>
Back to lab description
<svg version=1.1 id=Layer_1 xmlns='http://www.w3.org/2000/svg' xmlns:xlink='
http://www.w3.org/1999/xlink' x=0px y=0px viewBox='0 0 28 30' enable-background='new 0 0 28
30' xml:space=preserve title=back-arrow>
<g>
<polygon points='1.4,0 0,1.2 12.6,15 0,28.8 1.4,30 15.1,15'>
</polygon>
<polygon points='14.3,0 12.9,1.2 25.6,15 12.9,28.8 14.3,30 28,15'>
</polygon>
</g>
</svg>

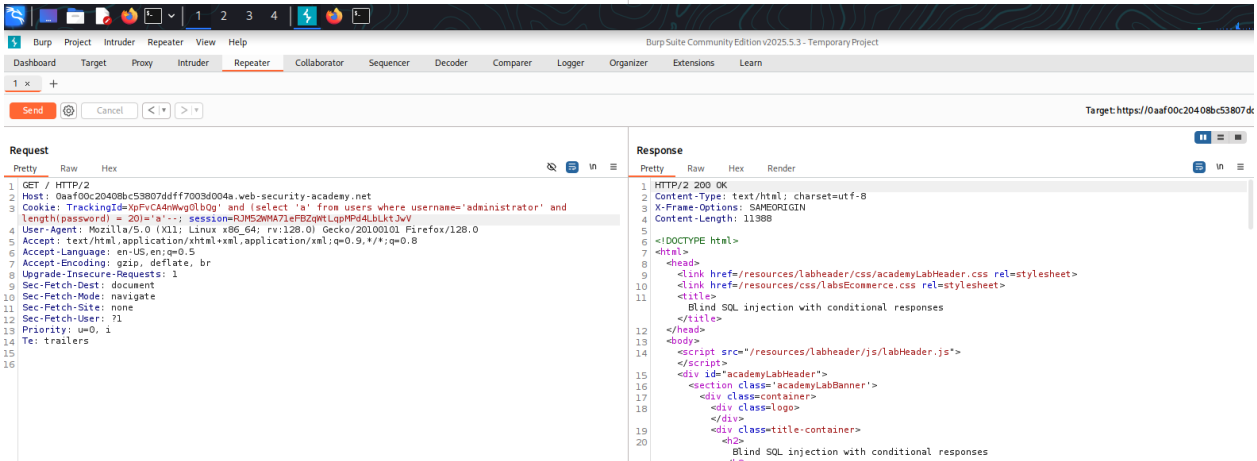
Search
0 highlights
welc
1 match


GET / HTTP/2
Host: 0aaf00c20408bc53807ddff7003d004a.web-security-academy.net
Cookie: TrackingId=XpFvCA4nWwgOlbQg' and (select 'a' from users where username='administrator')='a'--; session=RJM52WMA71eFBZqWtLqpMPd4LbLktJvV
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Priority: u=0, i
Te: trailers
HTTP/2 200 OK
Content-Type: text/html; charset=utf-8
X-Frame-Options: SAMEORIGIN
Content-Length: 11388
<!DOCTYPE html>
<html>
<head>
<link href=/resources/labheader/css/academyLabHeader.css rel=stylesheet>
<link href=/resources/css/labsEcommerce.css rel=stylesheet>
<title>
Blind SQL injection with conditional responses
</title>
</head>
<body>
<script src="/resources/labheader/js/labHeader.js">
</script>

<section class='academyLabBanner'>

<h2>
Blind SQL injection with conditional responses
</h2>
<a class=link-back href='
https://portswigger.net/web-security/sql-injection/blind/lab-conditional-responses'>
Back to lab description
<svg version=1.1 id=Layer_1 xmlns='http://www.w3.org/2000/svg' xmlns:xlink='
http://www.w3.org/1999/xlink' x=0px y=0px viewBox='0 0 28 30' enable-background='new 0 0 28
30' xml:space=preserve title=back-arrow>
<g>
<polygon points='1.4,0 0,1.2 12.6,15 0,28.8 1.4,30 15.1,15'>
</polygon>
<polygon points='14.3,0 12.9,1.2 25.6,15 12.9,28.8 14.3,30 28,15'>
</polygon>
</g>
</svg>

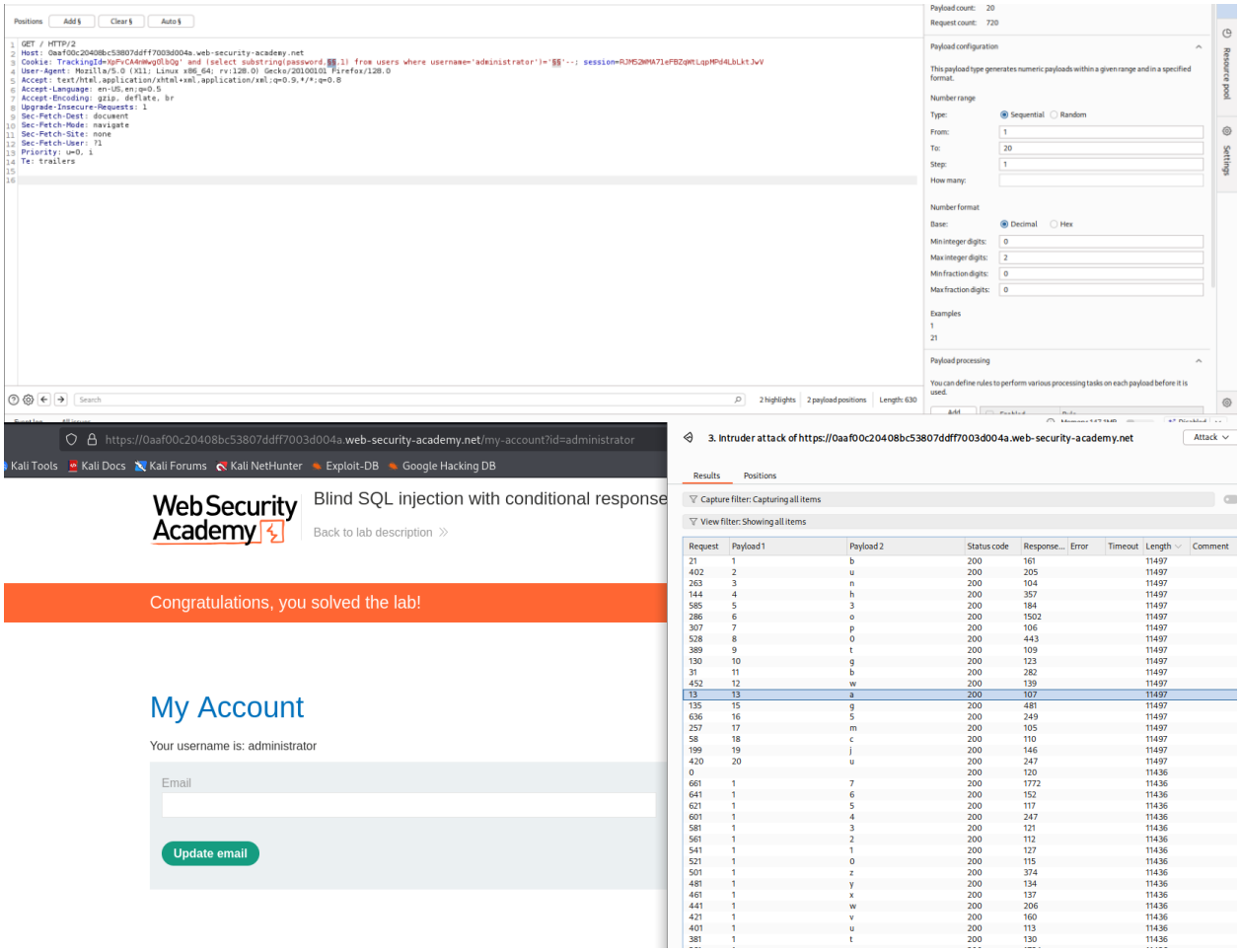
Search
0 highlights
welc
1 match


Burp Project Intruder Repeater View Help
Burp Suite Community Edition v2025.5.3 - Temporary Project
Dashboard Target Proxy Intruder Repeater Collaborator Sequencer Decoder Comparer Logger Organizer Extensions Learn
Send Cancel
Target: https://0aaf00c20408bc53807dc
Request
Pretty Raw Hex
GET / HTTP/2
Host: 0aaf00c20408bc53807ddff7003d004a.web-security-academy.net
Cookie: TrackingId=XpFvCA4nWwgOlbQg' and (select 'a' from users where username='administrator' and length(password) > 20)='a'--; session=RJM52WMA71eFBZqWtLqpMPd4LbLktJvV
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
Priority: u=0, i
Te: trailers
Response
Pretty Raw Hex Render
HTTP/2 200 OK
Content-Type: text/html; charset=utf-8
X-Frame-Options: SAMEORIGIN
Content-Length: 11388
<!DOCTYPE html>
<html>
<head>
<link href=/resources/labheader/css/academyLabHeader.css rel=stylesheet>
<link href=/resources/css/labsEcommerce.css rel=stylesheet>
<title>
Blind SQL injection with conditional responses
</title>
</head>
<body>
<script src="/resources/labheader/js/labHeader.js">
</script>

<section class='academyLabBanner'>

<h2>
Blind SQL injection with conditional responses
</h2>
<a class=link-back href='
https://portswigger.net/web-security/sql-injection/blind/lab-conditional-responses'>
Back to lab description
<svg version=1.1 id=Layer_1 xmlns='http://www.w3.org/2000/svg' xmlns:xlink='
http://www.w3.org/1999/xlink' x=0px y=0px viewBox='0 0 28 30' enable-background='new 0 0 28
30' xml:space=preserve title=back-arrow>
<g>
<polygon points='1.4,0 0,1.2 12.6,15 0,28.8 1.4,30 15.1,15'>
</polygon>
<polygon points='14.3,0 12.9,1.2 25.6,15 12.9,28.8 14.3,30 28,15'>
</polygon>
</g>
</svg>

Search
0 highlights
welc
1 match



Lab 2: Blind SQL Injection with Conditional Errors

Objective

This lab presented a similar blind SQLi scenario, but with a crucial difference: the application's front-end did not exhibit any visible changes based on query results. Instead, the success or failure of injected queries was reflected solely in the HTTP status codes returned by the server. The objective remained to extr`200 OK` administrator's password by observing these conditional error response. injection point. However, the absence of front-end changes necessitated a different approach to infer query results. By meticulously observing the HTTP status codes returned by the server, it was possible to differentiate between successful and unsuccessful injected queries. For instance, a successful query might return a status, while an unsuccessful or erroneous query might result in a `500 Internal Server Error` or a `400 Bad Request` .

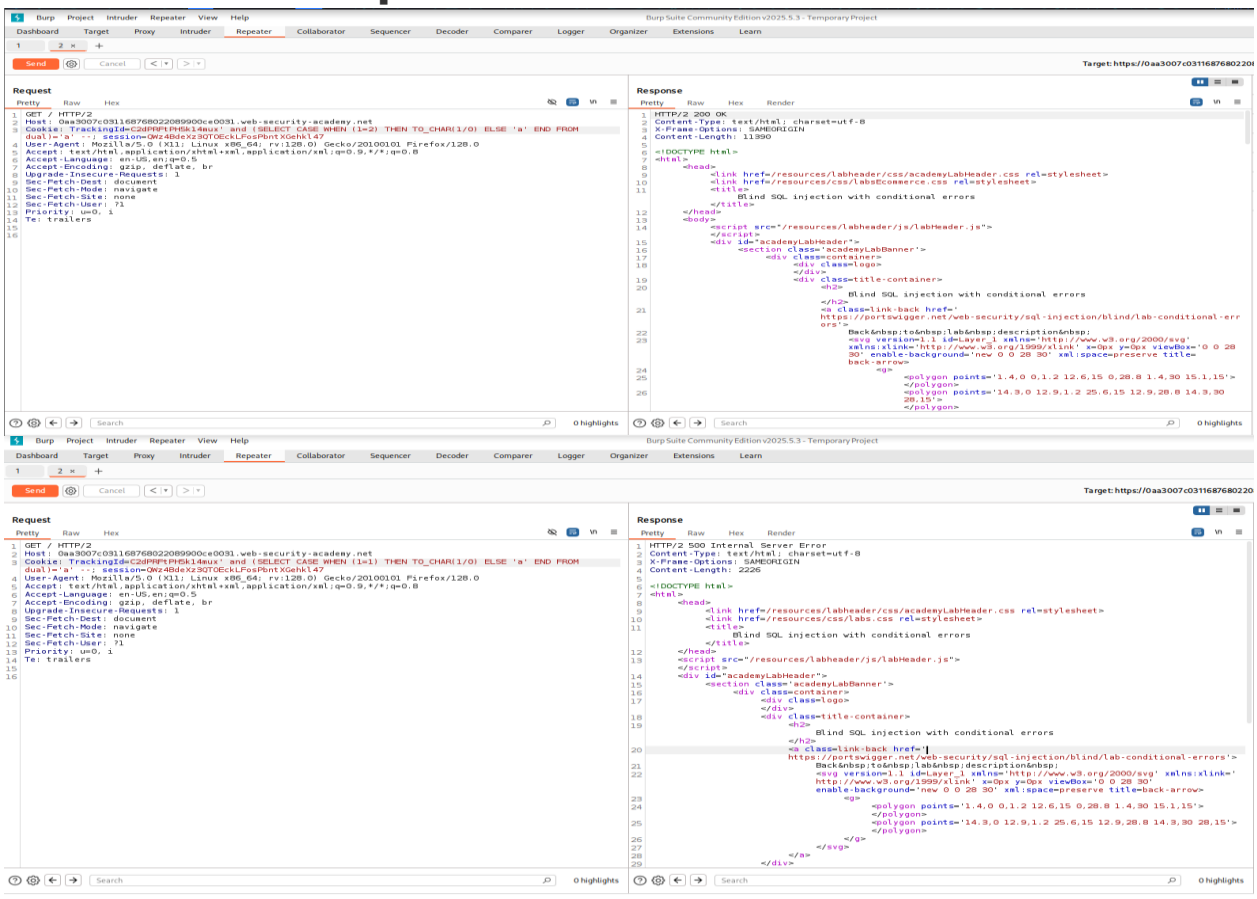
The initial phase involved determining the password length, analogous to the previous lab. Payloads leveraging `LENGTH()` and conditional logic were employed, but instead of observing a welcome message, the HTTP status code was monitored. Through this process, the password length was confirmed to be 20 characters.

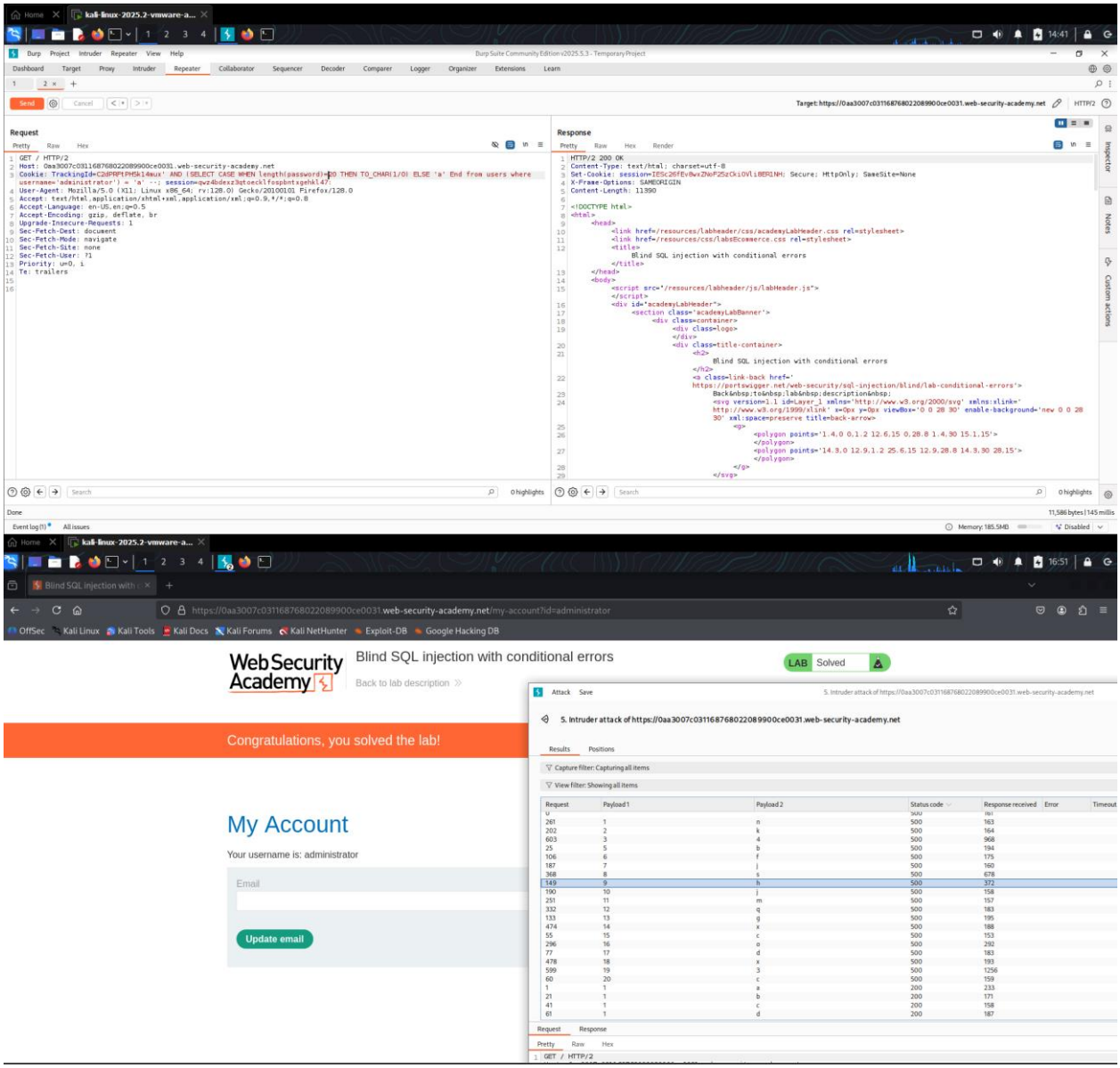
Subsequently, the password was extracted character by character using a similar brute-force methodology. Each character test involved crafting a query that would result in a distinct HTTP status code based on the correctness of the guessed character. For example, a query like `TrackingId=' OR (SELECT SUBSTRING(password, 1, 1) FROM users WHERE username='administrator') = 'a'--` would be sent, and the resulting HTTP status code would indicate whether 'a' was the correct first character. This iterative process, though time-consuming without automation, successfully revealed the password as `nk47bfjshjmqgxcodx3c` .

Impact

Despite the lack of direct visual feedback, the ability to infer query results through HTTP status codes still allowed for the complete compromise of user credentials. This highlights the importance of comprehensive error handling and logging, as subtle differences in server responses can still be leveraged by attackers.

Screen shots as a poc





Lab 3: Visible Error-Based SQL Injection

Objective

This lab presented a more straightforward SQLi scenario where the application directly leaked database error messages to the front-end. The objective was to exploit this verbose error reporting to extract sensitive information, including the database query itself, usernames, and passwords.

Methodology

Upon initial interaction with the application, it was immediately apparent that database errors were being displayed directly on the web page. This provided a significant advantage, as the error messages often contained fragments of the underlying SQL query, revealing valuable insights into the database structure and query logic. One particularly informative error message included the fragment:

```
SELECT * FROM tracking WHERE id = '30qklZyOS2a6QEAv' and cast((select
```

`username from users) as`.

This leaked query fragment indicated that the application was attempting to cast the result of a subquery (`select username from users`) into a specific data type. The presence of `id = '30qklZyOS2a6QEAv'` suggested that the `TrackingId` cookie (or a similar parameter) was still the primary injection point. However, the error message also indicated that the query was being truncated, suggesting a length limitation. To overcome this, the `TrackingId` cookie value was shortened or removed to allow for longer injected payloads.

Further manipulation of the injected query, combined with careful observation of the error messages, allowed for the extraction of specific data. When an error indicated that

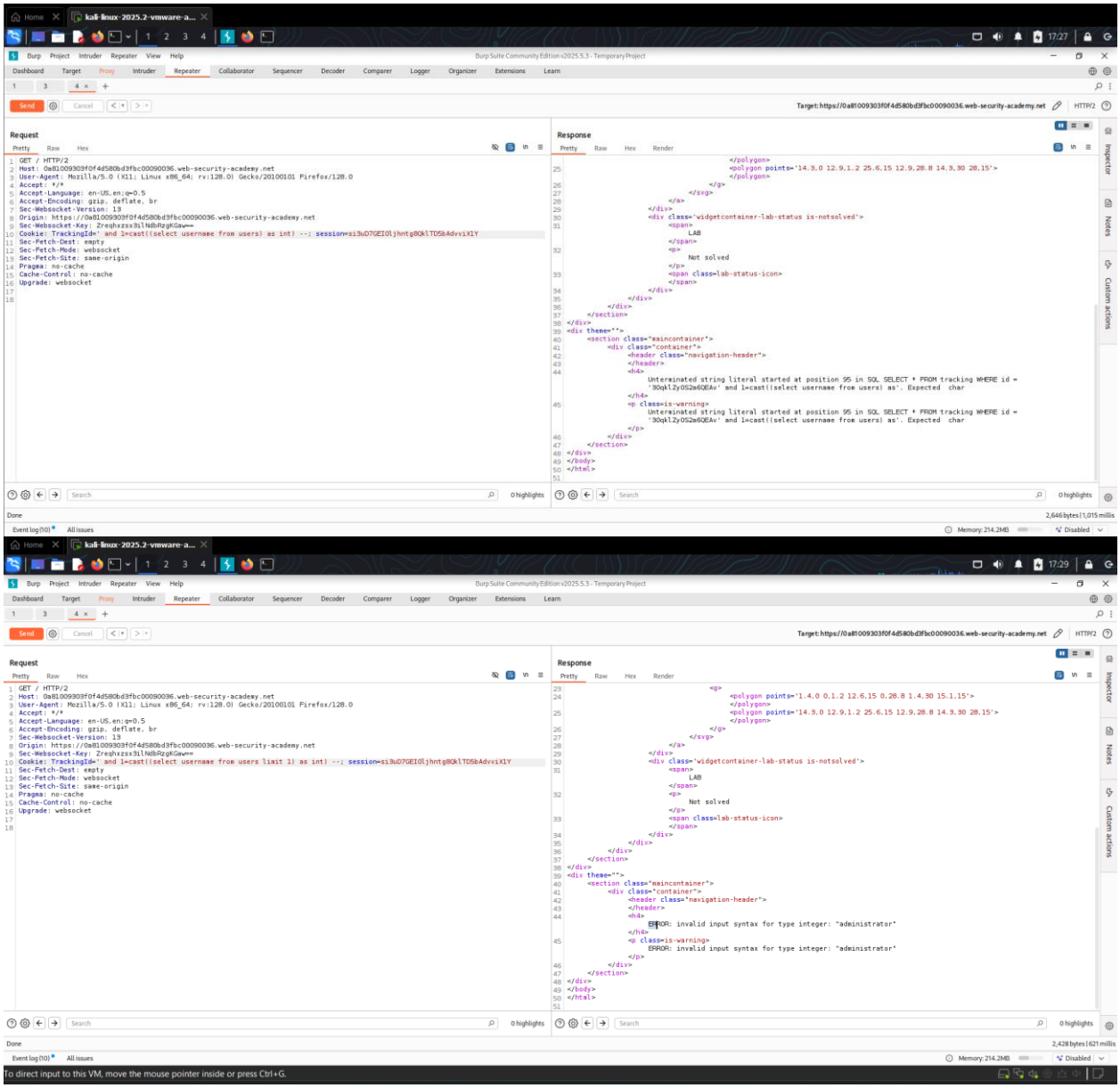
more than one row was returned, the `LIMIT 1` clause was introduced into the subquery to ensure only a single result was processed, thereby preventing the error and allowing the query to proceed. This iterative refinement of the injected payload, guided by the detailed error messages, ultimately led to the successful extraction of the `administrator` username.

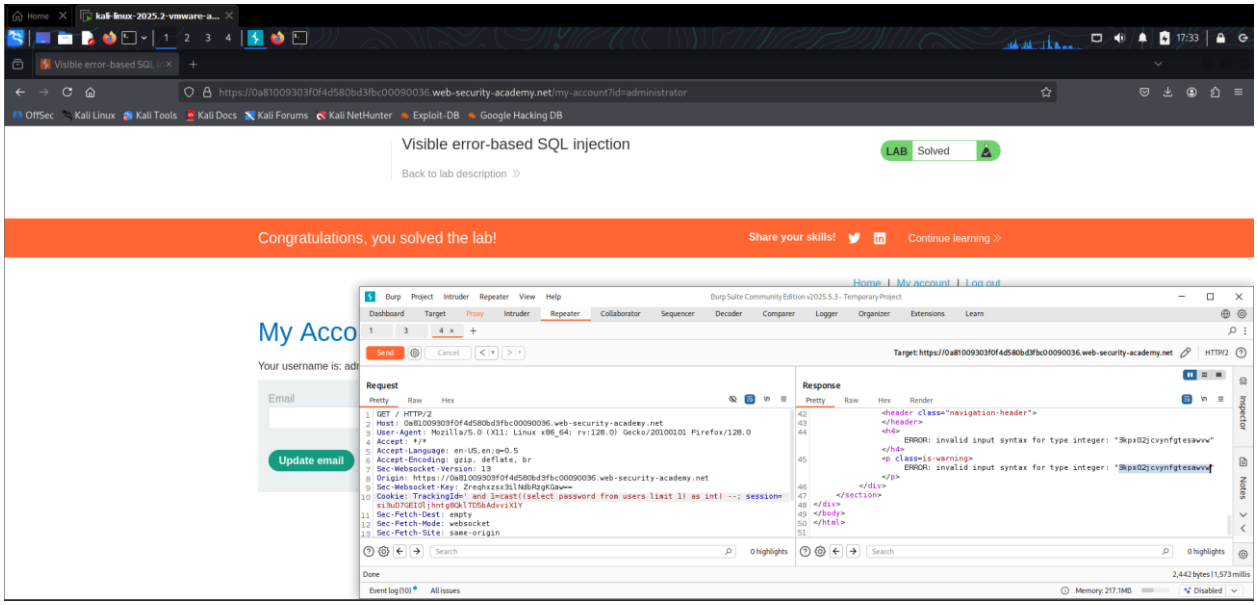
Once the username was confirmed, the same error-based technique was applied to extract the password. By modifying the subquery to `(select password from users where username='administrator')`, the application, in its verbose error reporting, inadvertently leaked the administrator's password: `3kpx02jcvynfgtesawvw`.

Impact

Visible error-based SQLi vulnerabilities are highly critical as they directly expose sensitive database information, including schema details, table names, column names, and even data. This type of vulnerability significantly accelerates the exploitation process, allowing attackers to quickly enumerate and exfiltrate critical data without the need for complex blind SQLi techniques.

Screen shots as a poc





Lab 4: Blind SQL Injection with Time Delays

Objective

This lab presented the most challenging blind SQLi scenario, where neither conditional responses nor error messages were available to infer query results. The only observable difference was a time delay in the application's response, making it a time-based blind SQLi. The objective was to identify this vulnerability and then leverage it to extract information.

Methodology

In the absence of visual or error-based feedback, the focus shifted to observing the application's response time. The core principle of time-based blind SQLi relies on injecting payloads that, when true, cause a measurable delay in the server's response. This delay acts as a binary indicator for the truthiness of the injected condition.

Initial testing involved injecting various time-delay functions specific to different database management systems (DBMS). After several attempts, the `pg_sleep()` function was found to be effective, indicating that the underlying database was likely PostgreSQL. A payload such as `TrackingId=\' OR pg_sleep(10)--` would cause a 10-second delay in the application's response if the condition was true, thereby confirming the vulnerability.

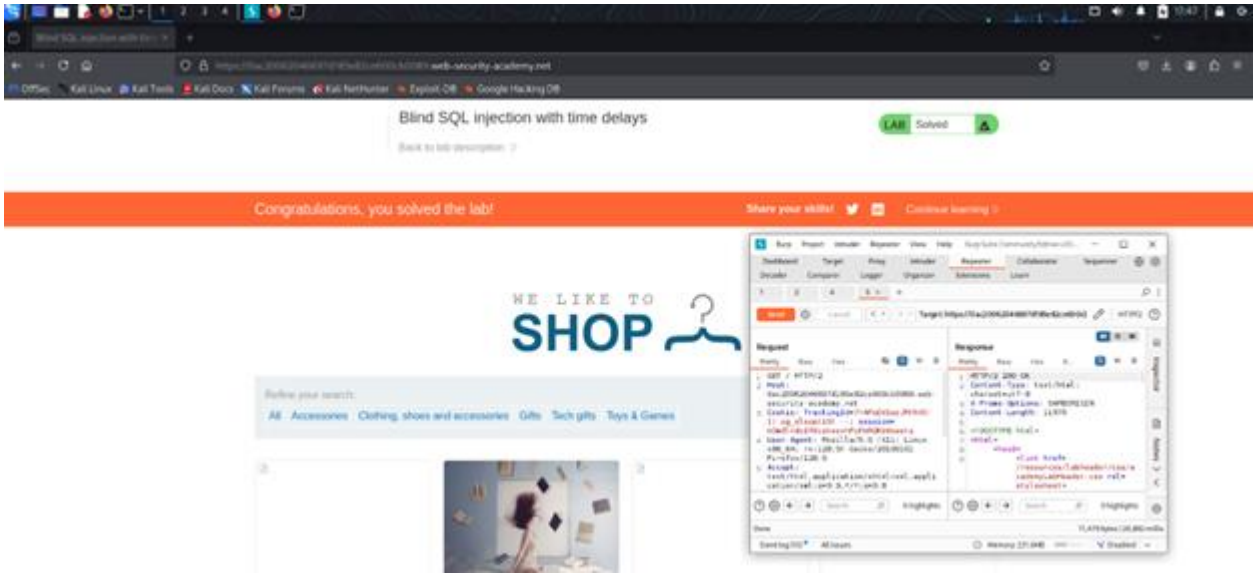
Once the time-based vulnerability was confirmed, the process of data extraction began. Similar to the conditional response lab, this involved an iterative, character-by-character brute-force approach. For instance, to determine the length of the administrator's password, a payload like `TrackingId=\' OR (SELECT CASE WHEN (LENGTH(password) > 10) THEN pg_sleep(10) ELSE pg_sleep(0) END FROM users WHERE username=\'administrator\')--` would be used. By observing whether a 10-second delay occurred, the password length could be precisely determined.

Extracting the password characters followed the same logic. A payload similar to `TrackingId=\' OR (SELECT CASE WHEN (SUBSTRING(password, 1, 1) = \'a\') THEN pg_sleep(10) ELSE pg_sleep(0) END FROM users WHERE username=\'administrator\')--` would be employed. If the guessed character (`\'a\'`) was correct, a delay would be observed. This process is inherently slow and computationally intensive, making automation absolutely essential. While manual attempts were made to confirm the first character, the full password extraction necessitated a custom Python script that could automate the iterative requests and analyze response times. Through this automated process, the password was successfully retrieved as `0wfxp5fo4rq93d6mnfqw`.

Impact

Time-based blind SQLi, while the most challenging to exploit, can be just as devastating as other forms of SQLi. It allows attackers to extract any data from the database, albeit at a slower pace. The stealthy nature of this attack, often leaving minimal traces in logs, makes it particularly dangerous. This vulnerability underscores the importance of robust input validation and the principle of least privilege in database interactions.

Screen shots as a poc



Lab 5: Blind SQL Injection with Time Delays and Information Retrieval

Objective

This lab combined the challenges of time-based blind SQLi with the need for systematic information retrieval. The objective was to leverage time delays to confirm the database type and then extract sensitive data, specifically the administrator's password, using a combination of techniques, including automation.

Methodology

Building upon the understanding gained from the previous time-based lab, the initial step involved confirming the database management system (DBMS). The `pg_sleep()` Payload, which proved effective in the prior lab, was re-tested and confirmed to induce a time delay, thereby reinforcing the conclusion that the underlying database was PostgreSQL.

The core challenge of this lab was to efficiently extract information, specifically the password, from a time-based blind SQLi vulnerability. This required a more sophisticated approach than manual observation. The strategy involved mixing the time-based technique with conditional logic to determine the password length and then each character of the password.

To determine the password length, payloads similar to those used in Lab 4 were employed, where a time delay would occur if the guessed length was correct. This iterative process, though conceptually simple, is practically infeasible without automation due to the numerous requests and the need for precise time

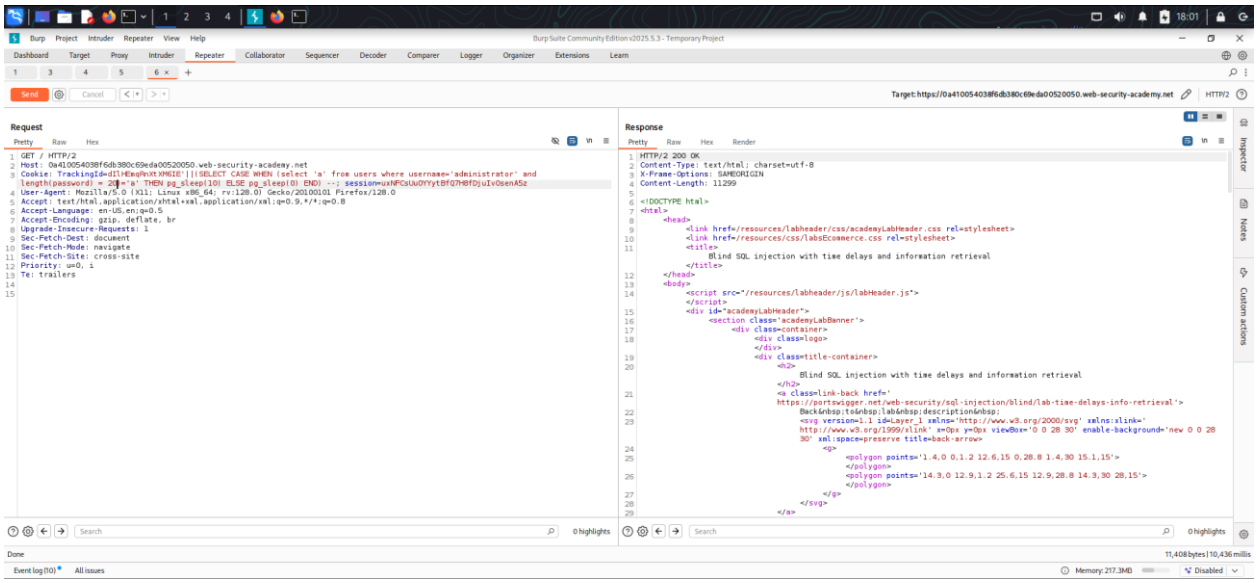
measurement.

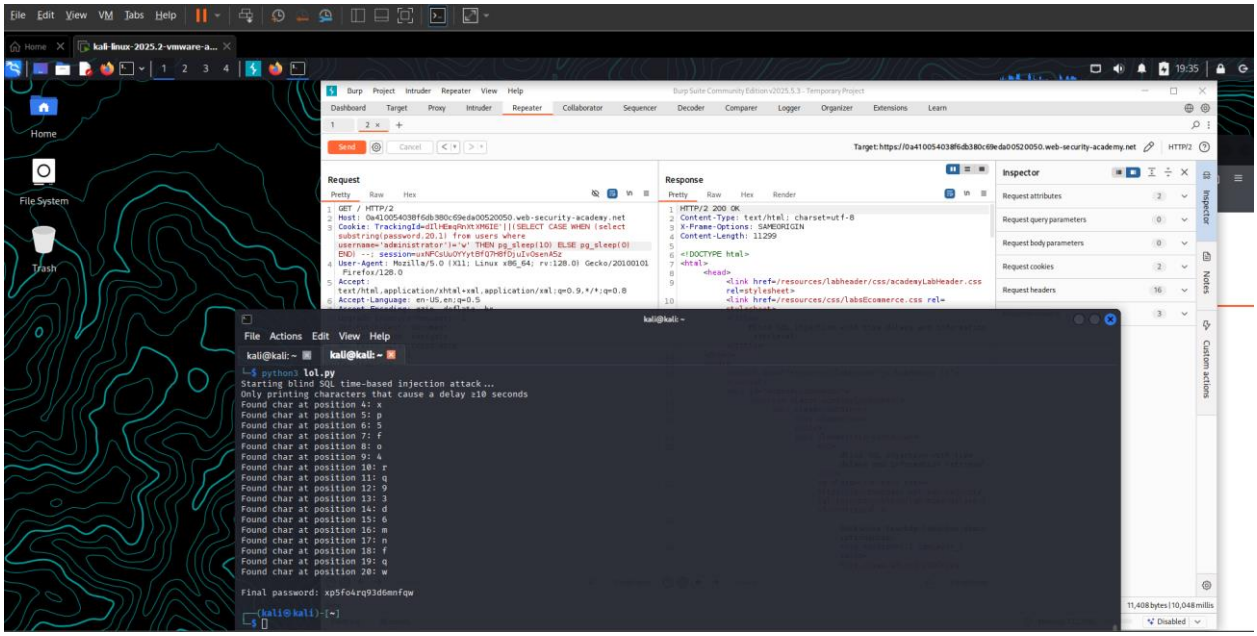
The most critical aspect of this lab was the character-by-character brute-force attack to reconstruct the password. While manual attempts were made to understand the response behavior for a single character, it quickly became apparent that a manual approach for the entire password would be prohibitively time-consuming. Therefore, a custom Python script was developed. This script emulated the functionality of a Burp Suite Repeater, sending crafted HTTP requests with time-based payloads and analyzing the response times to infer the correctness of each guessed character. The script systematically iterated through possible characters for each position of the password, observing the induced time delays to confirm the correct character. Through this automated and persistent effort, the administrator's password was successfully determined to be: `0wfxp5fo4rq93d6mnfqw` .

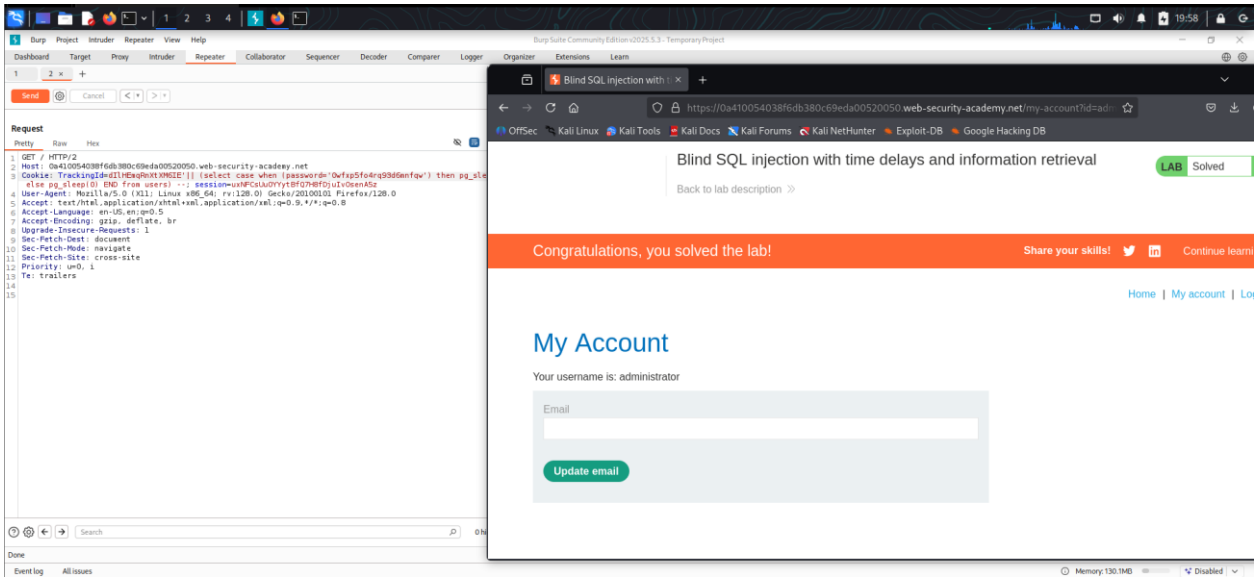
Impact

This lab underscores the power of automation in exploiting complex blind SQLi vulnerabilities. While time-based SQLi is often considered the most difficult to exploit due to its slow nature, it remains a critical threat. The ability to programmatically extract data, even with subtle time differences, means that any data within the database is potentially vulnerable. This highlights the necessity for robust security controls that prevent SQL injection at its source, as relying on the difficulty of exploitation is not a viable long-term security strategy.

Screen shots as a poc







Conclusion and Recommendations

The PortSwigger blind SQLi labs provide an excellent practical demonstration of the diverse forms and exploitation techniques associated with SQL Injection vulnerabilities. From conditional responses and error messages to time-based delays, each lab highlighted different facets of this critical web security flaw. As an experienced penetration tester, these labs reinforce several key takeaways:

Comprehensive Input Validation is Paramount: The fundamental defense against SQLi lies in rigorously validating and sanitizing all user-supplied input. This includes not only obvious input fields but also hidden parameters, HTTP headers, and cookies. Both client-side and server-side validation are crucial, with server-side validation being the ultimate safeguard.

1. **Parameterized Queries and Prepared Statements:** The most effective programmatic defense against SQLi is the use of parameterized queries or prepared statements. These mechanisms ensure that user input is treated as data and not as executable code, effectively neutralizing the injection vector.
2. **Minimize Error Verbosity:** Applications should never expose raw database error messages to the end-user. Verbose error messages provide invaluable information to attackers, significantly aiding in the exploitation process. Generic error messages should be displayed, and detailed error logging should be directed to secure, internal systems.
3. **Principle of Least Privilege:** Database users should operate with the absolute minimum privileges necessary to perform their functions. This limits the potential damage an attacker can inflict even if a SQLi vulnerability is successfully exploited.
4. **Regular Security Audits and Penetration Testing:** As demonstrated by these labs, even subtle application behaviors can indicate critical vulnerabilities. Regular security audits and penetration testing are essential to identify and remediate such flaws before they can be exploited by malicious actors.
5. **Awareness of Different SQLi Types:** Understanding the various types of SQLi (in-band, error-based, blind, time-based) is crucial for effective detection and exploitation. Each type requires a slightly different approach, and a skilled penetration tester must be adept at recognizing and adapting to these nuances.
6. In summary, SQL Injection remains a pervasive and dangerous vulnerability. The PortSwigger labs serve as an invaluable resource for understanding the intricacies of these attacks and for developing the skills necessary to defend against them. A proactive and multi-layered security approach, encompassing secure coding practices, robust input validation, and continuous security testing, is indispensable in mitigating the risks posed by sqli.