Final Project: Hacking WebGoat

Members: Tanner Balluff, Rafael George, Meruyert Karzhaubaeva, Eric Kumara, Yu-An Tsai
Date: February 28, 2025

Part 1: Exploiting Web Vulnerabilities

Vulnerability 1a, Broken Access Control – Hijack a Session: The WebGoat Hijack a Session module demonstrates a broken access control vulnerability resulting from predictable session management. The application issues a session cookie named hijack_cookie that follows a predictable format—a long sequential number followed by a Unix epoch timestamp (separated by a dash). Because both components are either sequential or time-based, an attacker can predict or brute-force a valid cookie value, allowing unauthorized session hijacking.

Exploit: The exploitation process involves the following steps:

- Interception: An attacker intercepts the POST request to `/WebGoat/HijackSession/login` using a proxy tool (in our case, ZAP). The request includes the *hijack_cookie* in the Cookie header. See Fig. 1 in Appendix **(subsequent figure references, please, go to Appendix).**
- Data Collection & Analysis: The attacker sends the intercepted request multiple times and records the returned hijack_cookie values. Analysis of these values reveals that the first part increments predictably while the second part (the timestamp) falls within a narrow range. See Fig. 2 for the manual request editor adjusted for brute forcing.
- Using ZAP's Fuzzer, the attacker brute-forces the timestamp portion within the observed range. The fuzzer output eventually returns a response with a success message (e.g., "Congratulations" or similar confirmation) indicating that the correct cookie value was found.
  See Fig. 3 for the brute force output showing the correct session, and Fig. 4 for the final successful activity screen.

Risk:

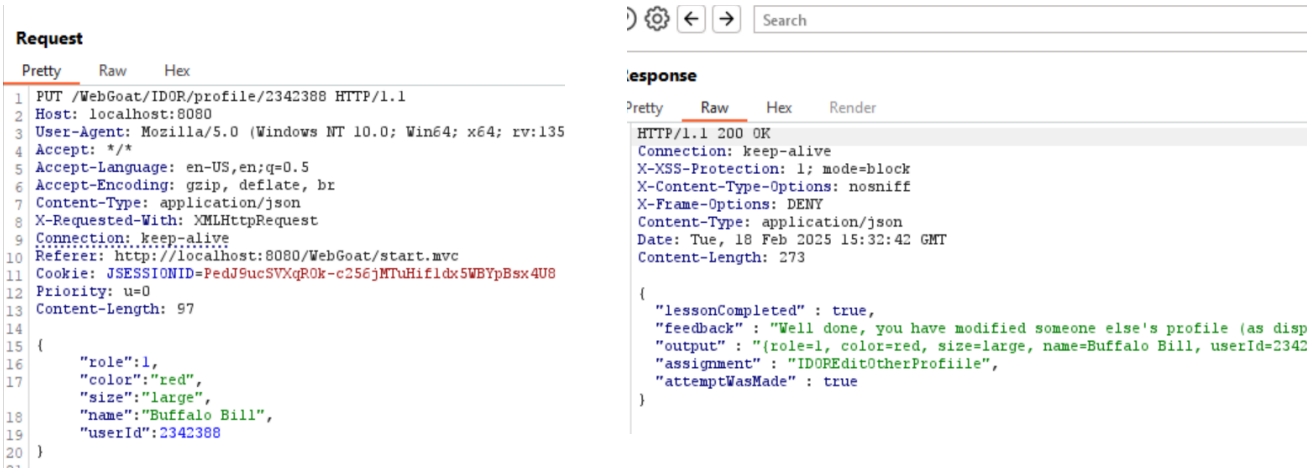
- *Technical Risk:* This vulnerability allows attackers to bypass authentication controls and access user sessions without proper credentials.
- *Business Risk:* Unauthorized session hijacking can lead to data breaches, loss of sensitive information, and unauthorized transactions, severely impacting customer trust and the organization's reputation. The predictable nature of the session tokens makes this a high-risk issue that must be mitigated through secure session management practices—such as using cryptographically secure random values for session identifiers and implementing additional session security controls.

Vulnerability 1b, Insecure Direct Object References (IDOR): The vulnerability occurs when internal services do not implement proper access control, which would allow an adversary to gain information that could escalate its privileges and have access to otherwise restricted services.

Exploit: Initially, when the user is logged in, it is assigned an ID of 2342384 which does not have certain privileges. However, the IDOR vulnerability allows an adversary to enumerate other internal IDs due to

the lack of access control and eventually find another ID that has higher privileges. For example, after enumerating the GET request captured in Burp, the adversary obtained another ID of 2342388.

With this new ID, altering the initial request to a PUT request and the *Content type* to *application/json* allows an attacker to alter information such as the *role, color, size* related to any ID because the ID 2342388 has higher privileges. In this case, we altered the 2342388 ID to have *role: 1, color: "red", size: "large"*



Risk:

- *Technical Risk:* An IDOR could be used as a foothold for privilege escalation and could alter the internal privileges of the existing roles used in the architecture. Users with altered privileges could be prevented from performing their core functions, which would affect business operations.

Vulnerability 2a (Crypto Basics (8)), Insecure Default Configurations & Exposed Secrets in Docker Container Images: Many systems suffer from insecure default configurations—such as leaving private keys unencrypted or using default passwords—which can be especially problematic in cloud environments. In this exercise, a secret was inadvertently left inside a Docker container image. This vulnerability demonstrates that if container images are not properly hardened (for example, by changing default settings), attackers can gain unauthorized access to sensitive information. In this case, the unprotected secret enables an attacker to decrypt a confidential message, thereby illustrating the risk posed by default and insecure configurations.

Exploit:

Launching the Vulnerable Container:
The attacker begins by starting the vulnerable Docker container using a command such as:
`docker run -d webgoat/assignments:findthesecret`

Accessing the Container Shell:
The attacker then logs into the container using:
`docker exec -it <containerID> /bin/bash`

1. Initially, attempts to access protected directories (like `/root`) are blocked due to default permissions.
2. **Extracting and Modifying Sensitive Files:**
   By exploring the container's file system (e.g., in `/etc`), the attacker locates critical files such as `/etc/passwd`. The attacker uses `docker cp` to copy this file to the host and then edits it—using a script (e.g., in PowerShell)—to change the WebGoat user's UID/GID from `1000:1000` to `0:0`, effectively granting root privileges.
   See Fig. 5 for a screenshot of the passwd file being edited.
3. **Retrieving the Secret and Decrypting the Message:**
   After modifying the credentials and re-injecting the file back into the container, the attacker gains elevated access. This allows retrieval of the secret left within the container. Using the secret along with an OpenSSL command inside the container, the attacker decrypts an encrypted message. The successful decryption, accompanied by a confirmation screen, verifies the exploit. *See Fig. 6* for the screenshot showing the retrieval of the secret/full process and *Fig. 7* for the final activity successful screen. The other technically complex activity has been included in *Fig. 8.*

Risk:

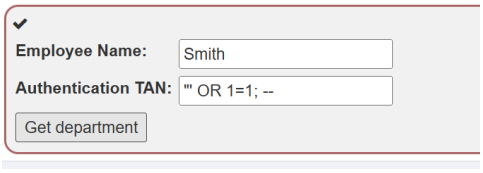
- *Technical Risk:* Insecure default configurations enable attackers to extract sensitive secrets from container images, allowing privilege escalation and unauthorized access.
- *Business Risk:* Such vulnerabilities can lead to data breaches, compromise of cloud-hosted services, and significant reputational and financial damage, as attackers may exploit these weaknesses to infiltrate critical systems.

Vulnerability 3a, SQL (Introduction): The vulnerability discusses the basic idea of how unsanitized user input can be interpreted as part of the instruction which could execute additional instructions to fetch data, change data or assign privileges.

Exploit: This module assumes that the backend SQL server has an equivalent parsing prompt referenced below.

```
"SELECT * FROM employees WHERE last_name = '" + name + "' AND auth_tan = '" + auth_tan + "';
```

In this prompt, neither the *name* or *auth_tan* is sanitized, so an attacker could input *auth_tan* to include SQL specific instructions such as the comment (--) and eol (;) to execute a query. For example, with the payload below, an attacker obtained the entire employees table by always evaluating to True with the input *OR 1=1;*



Additionally, with command chaining, additional commands can be executed after the eol (;). If an adversary wanted to alter the data, setting *auth_tan* to *1234';UPDATE employees SET Salary=100000 WHERE first_name='John' AND last_name='Smith' ;--* would execute an additional update statement to change the salary from 64350 to 100000.

37648	John	Smith	Marketing	64350

Well done! Now you are earning the most money. And

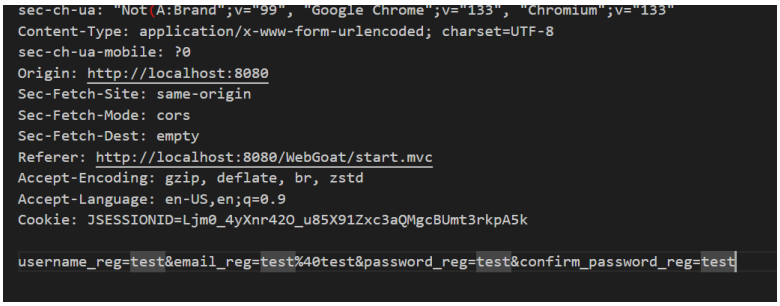
USERID	FIRST_NAME	LAST_NAME	DEPARTMENT	SALARY
37648	John	Smith	Marketing	100000

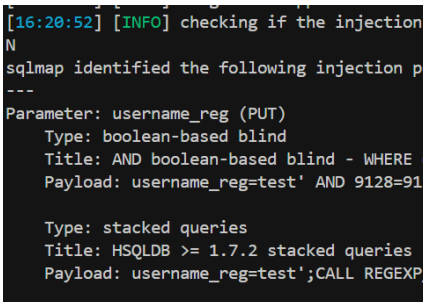
Risk:

- *Technical Risk:* Exploiting SQL Injections allows attackers to alter the underlying database that could be used in tangent with other services. Updating the database could cause other technological dependencies to crash.
- *Business Risk: However,* with the implication of exfiltrating data and updating values, this information can be used to disrupt business operations such as deleting a username database which prevents any incoming users from being authenticated which disrupts business operations.

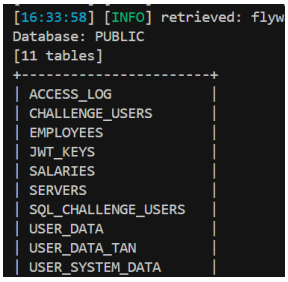
Vulnerability 3b, SQL (Advanced): The vulnerability explores a more realistic case of SQL-injections and using inferences of boolean statements to eventually brute force the credentials of a user.

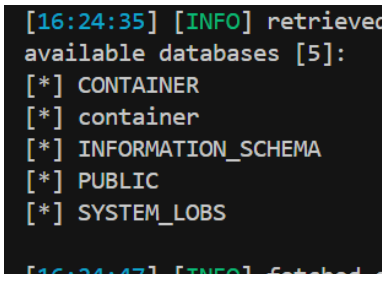
Exploit: The User Registration form of an application is captured with Wireshark, and, after running an open-source tool called SQLMap, it determined that the username_reg input is vulnerable to a boolean based blind injection. If the input created a True statement, then the form would return “User already exists”, but, if the input created a False statement, nothing was returned.

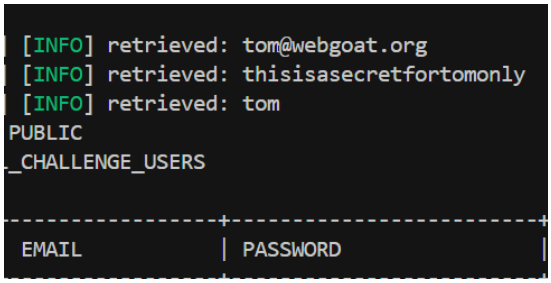




The difference in the response values can be used to infer the password because only the valid letter at a substring position would return “User already exists”, and any other letter would return nothing. SQLMap uses this behaviour to determine the database, the tables in the database, and eventually the entry of *tom* through brute forcing.







Risk:

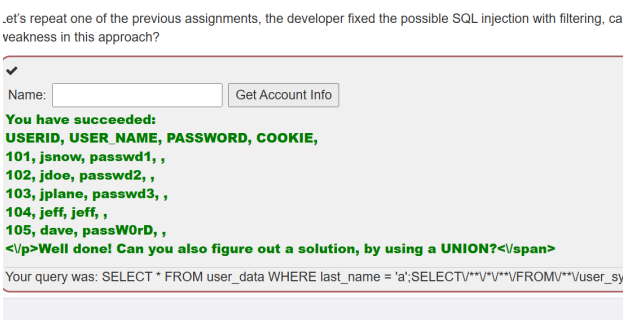
- *Technical Risk:* The underlying focal point of a boolean based SQL injection is that the response value provides too much information, and this is used by the adversary to learn and eventually access the internal database.
- *Business Risk:* As with Vulnerability 3a, the behaviour exhibited could disrupt business operations if, in this case, boolean based injections could compromise credentials and allow an adversary to login and perform unwanted actions. This would likely stop users from using this service in the future.

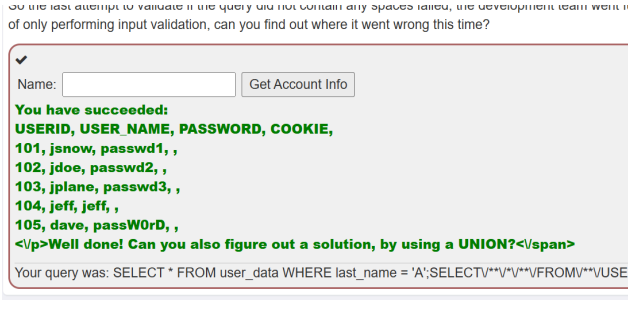
Vulnerability 3c, SQL (Mitigation): This module was divided into 2 sections: A boolean based attack similar to Vulnerability 3b and proper input handling. Since boolean based attack was discussed in 3b, only the exploits related to improper handling will be discussed.

Exploit: The ideal way to handle SQL statements would be to have a prepared statement where inputs are directly translated to strings. Thereby, no input is interpreted as part of a command. However, there are different ways of input handling that attackers could circumvent.

```
try {
    Connection conn = DriverManager.getConnection(DBURL, DBUSER, DBPW);
    PreparedStatement ps = conn.prepareStatement("SELECT * FROM users WHERE name = ?");
    ps.setString(1, "Admin");
    ps.executeUpdate();
} catch (Exception e) {
    System.out.println("Oops. Something went wrong!");
}
```

For example, if an input validation prohibited the use of spaces in the input, an attacker could insert a payload *a';SELECT/**/*/**/FROM/**/user_system_data;--* where */**/* effectively behaves as a space. Similarly, if an input validation deleted command words such as *SELECT* or *FROM*, wrapping those words such as *SESELECTLECT* would circumvent the rule. Therefore, the payload would be *a';SELECSELECTT/**/*/**/FROFROMM/**/user_system_data;--* for that input validation. The module highlighted that input validation is not enough to properly handle SQL statements.





Risk:

- *Technical Risk:* Improper input handling and validation would allow an attacker to inject a malicious payload. A faulty input validation implementation is likely to assure developers that the handler is secure even when it is not.

- *Business Risk:* The business risk of a fault input validation implementation means that resources are needed to develop a robust implementation which includes the overhead of SDLC and development.

Vulnerability 7a, Software & Data Integrity – Insecure Deserialization: Insecure deserialization occurs when an application blindly deserializes data without proper validation, allowing attackers to inject malicious objects that can lead to unintended behavior or remote code execution. In WebGoat's Insecure Deserialization lesson, the application expects a serialized object of type `VulnerableTaskHolder`—which encapsulates a task (specifically, a "sleep 5" operation). The lesson validates that the deserialized object meets strict criteria (e.g., a 5-second timeout). By crafting a malicious serialized object, an attacker can manipulate this behavior and complete the lesson.

Exploit:

1. **Source Code Analysis:**
   We began by examining the source code from the WebGoat GitHub repository, specifically the `InsecureDeserializationTask.java` and its associated `VulnerableTaskHolder.java`. The latter defines the object structure expected by the server.
   See Fig. 9 for a screenshot of our modified `VulnerableTaskHolder.java` code.
2. **Payload Crafting with Custom Java Code:**
   Using our custom `Attack.java` code (compiled with OpenJDK 11), we created a `VulnerableTaskHolder` object preconfigured with a "sleep 5" task. We compiled our exploit code and generated a serialized file. This file was then converted to a Base64 token so it could be submitted via the WebGoat lesson interface.
   See Fig. 10 for a screenshot of our `Attack.java` source code.
3. **Submission and Verification:**
   The Base64-encoded token was submitted as input in the WebGoat lesson. Upon successful deserialization, the application recognized our payload as a valid `VulnerableTaskHolder` object with the expected 5-second timeout, thus marking the lesson as complete.
   See Fig. 11 for a screenshot showing our final activity success screen.

Risk:

- *Technical Risk:* Insecure deserialization can allow attackers to execute arbitrary code or perform unauthorized actions by manipulating serialized objects.
- *Business Risk:* Exploitation of this vulnerability could lead to full system compromise, data theft, and significant financial and reputational damage.

Vulnerability 8a, Security Logging Failures – Log Spoofing and Log Bleeding: Insecure logging can allow attackers to manipulate or inject false data into application logs, thereby obfuscating malicious activity or exposing sensitive information. In WebGoat's Logging Security lesson, the vulnerability is twofold: first, the logging mechanism does not adequately sanitize input—allowing an attacker to inject fabricated log entries (e.g., a successful "admin" login)—and second, sensitive credentials (such as the administrator's password) are inadvertently logged and can be retrieved from the system logs. This not only undermines audit integrity but also provides an attacker with critical access information.

Exploit:

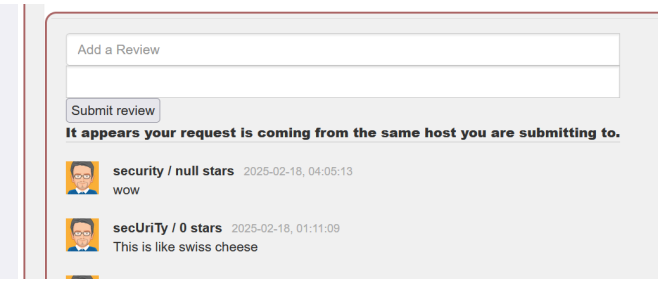
1. **Log Spoofing:**
    - **Objective:** Make it appear as though the username “admin” has successfully logged in.
    - **Method:** The attacker crafts and submits a POST request (via a proxy tool) with a manipulated username field designed to inject a false log entry. This spoofed log entry is then recorded by the WebGoat server, as shown by the developer tools.
    - *See Fig. 12,* which shows the spoofed login entry in the server log, confirming that the log file can be manipulated.
2. **Log Bleeding and Credential Retrieval:**
    - **Objective:** Extract sensitive administrator credentials that are inadvertently logged during server boot-up.
    - **Method:** After spoofing the log entry, the attacker inspects the log files (using the browser’s developer tools and the Docker container’s terminal). Within the session labeled "log-bleeding," the admin credentials (e.g., a password) are revealed.
    - *See Fig. 13,* which displays the admin credential in the log file, and *Fig. 14,* which shows the final successful activity screen proving that the admin login has been achieved using the retrieved credentials.

Risk:

- *Technical Risk:* Unsanitized log inputs and exposed sensitive information in logs enable attackers to both mislead administrators (by tampering with audit logs) and directly obtain credentials needed for unauthorized access.
- *Business Risk:* Compromised logs can lead to undetected breaches and full system compromise, resulting in data theft, regulatory non-compliance, and significant damage to organizational reputation and customer trust.

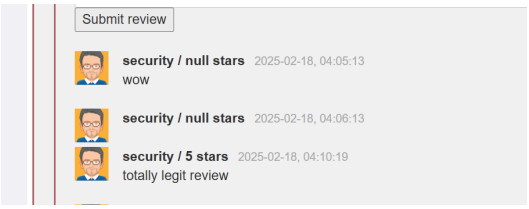
Vulnerability 9a, Cross Site Request Forgeries (CSRF): The vulnerability is an exploit where an adversary tricks a logged in user to perform unwanted actions by exploiting the trust web services have for an authenticated user session.

Exploit: The exploit assumes that a user is currently logged into a web service. In this exploit, this review page is vulnerable to a CSRF where a threat actor could arbitrarily insert comments from a different site. After capturing an initial request in Burp, we can see that there is a hard-coded value stored in *validateReq* for any request.



Then, a payload html would automatically send a POST request to the same URL endpoint, which would trigger the current authenticated session to send a request. The server would accept this request since it

assumes that the authenticated session meant to send this request. This would result in a new review with an arbitrary value set by the adversary which in this case would be a 5 star review and a text payload *totally legit review*



Risk:

- *Business Risk:* A CSRF is likely to affect the user of a web service by exploiting the trust between a web server and an authenticated user session. Affected users are unlikely to continue using a service that allows others to act on their behalf, which would affect the business bottom line.

Vulnerability 9b, Server-Side Request Forgery (SSRF): Server-Side Request Forgery (SSRF) occurs when an application accepts user-supplied URLs and causes the server to issue HTTP requests to arbitrary destinations. In WebGoat's SSRF lessons, the application embeds hidden input fields that determine which resource the server will request. An attacker can modify these hidden inputs to redirect the server's request—first, by changing the resource identifier (e.g., from "tom" to "jerry") and then by specifying an external URL (e.g., "http://ipconfig.pro"), thereby demonstrating how SSRF can be exploited.

Exploit:

1. **Changing the Resource Identifier:**
    - *Observation:* Initially, clicking the button in the SSRF module returns data labeled "Tom."
    - *Exploitation:* Using browser developer tools, we inspected the page and identified a hidden input field that holds the default value "tom." By modifying this value to "jerry" and then submitting the form, the server responded with data labeled "Jerry."
    - *Outcome:* This confirms that the server is dynamically fetching a resource based on the hidden input value. *See Fig. 16/17/18 for a screenshot of the hidden input field modification and the resulting "Jerry" output.*
2. **Redirecting to an External URL:**
    - *Observation:* A new button is introduced in the next exercise. Initially, clicking the button does not yield the expected outcome.
    - *Exploitation:* Upon inspecting the page, we found another hidden input field controlling the URL to be fetched by the server. Changing the value of this hidden field to "http://ipconfig.pro" forces the server to make a request to that external site. The response confirms that the server retrieved data from ipconfig.pro.
    - *Outcome:* The successful redirection demonstrates that the application does not properly validate the URL input, leaving it vulnerable to SSRF attacks. *See Fig. 19/20 for a screenshot showing the modified hidden input with "http://ipconfig.pro" and the corresponding server output as well as activity proof.*

Risk:

- *Technical Risk:* SSRF vulnerabilities allow attackers to manipulate server-side requests, potentially accessing internal resources, bypassing firewalls, or performing further attacks from the server's perspective.
- *Business Risk:* Exploiting SSRF can lead to data leakage, unauthorized internal access, or compromise of critical systems, thereby impacting organizational reputation and incurring financial losses.

Part II: Mitigation Strategies

Mitigate 1a, Broken Access Control – Hijack a Session: As described above, the weak session cookie IDs are subject to session-based brute force attacks. An unencrypted client-server communication via HTTP allows intercepting payloads in plain text.

Mitigation Strategy

- Cookies generated by the server should be complex and random to prevent brute force, guessing, and reverse engineering. Truly random cookies should make it impractical to brute force them.
- Session cookies transferred between a server and a browser should be encrypted with TLS/SSL. Hence the application-layer protocol should be HTTPS.
- Cookies should be sent with appropriate secure flags, such as `Secure` and `HttpOnly`, to protect confidentiality in a MITM attack [1].
- To prevent cookie hijacking, the cookie should be updated after user authentication [2]. The number of simultaneous sessions should be limited to one per user, and any new session for the same user should require re-authentication.
- There should be a limit on the number of requests coming from the same host to prevent capturing session cookies for analysis and prediction, and potentially causing a DoS attack.

Security Touchpoints

- **Code Review** would have assessed proper session cookie handling, such as security flags. It would also identify issues in session management, such as enforcing limiting session number and re-authentication.
- **Penetration Testing** would have simulated MITM and session hijacking attacks to understand the system behavior and robustness, allowing to identify any unaddressed weaknesses.
- **Risk-Based Security Testing** would have covered edge cases, such as session expiration, cookie regeneration, and repeated session connection.
- **Abuse Cases** would have accounted for the multiple frequent repeated requests for the purpose of scraping the cookie IDs for further analysis, brute force, and guessing;
- **Security Requirements** would have set functional requirements of secure cookie handling by enforcing policies. By applying a non-functional requirement of availability, it could have restricted a number of attempts, thus impeding the attacker's ability to guess and brute force the cookie ID.

Mitigate 1b, Insecure Direct Object References (IDOR): As explained in Part I, Insecure Direct Object References represent the vulnerability of direct unauthorized object access, compromising confidentiality and allowing unauthorized operations.

Mitigation Strategy

- Set up role-based access controls based on business and design logic behind the URLs to prevent authenticating and abusing authorizations, such as escalating user privileges and modifying data;
- Ensure RESTful API calls (GET, PUT, POST, DELETE) are available to only authorized users [3]. There are services online that enable role-based API calls access, that provide additional security.
- Obscure references by hashing or randomization [4] to prevent brute force/guessing of references that may lead to potentially sensitive objects (such as documents and web-pages).

Security Touchpoints

- **Code Review** would have helped to detect the hard-coded and improper object references handling and to identify missing access controls to IDOR objects and permissions for API calls. It also could have checked for authorizations to modify the object's parameters, like *role.*
- **Architectural Risk Analysis** would have detected improper obfuscation of referenced objects, such as hashing or randomization, and exposure of database objects. Considering special online services, it could have ensured that appropriate API limitations are configured.
- **Penetration Testing** could have verified if it is possible to brute force IDOR references, abuse authentications, escalate privileges, modify other objects, and obtain sensitive information.

Mitigate 2a, (Crypto Basics (8)): Insecure Default Configurations & Exposed Secrets in Docker Container Images: While WebGoat contains several cryptography-related exercises, in this section, we focus on the vulnerability of storing private keys in the Docker container.

Mitigation Strategy

- To prevent inadvertent disclosure of private keys, they should be stored in appropriate key managers (such as Docker Secrets [5]), removed from Docker images, and be encrypted at rest.
- While attempts to access root directories initially failed, due to wide file permissions, it was possible to access the critical passwd file, copy it, and abuse authentication by changing user ID. To mitigate this, remove default Docker container configurations, enable SSH authorization, and narrow down file permissions (such as copying and writing) to prevent unauthorized access and escalation of privileges.

Security Touchpoints

- **Code review** could have helped to scan the Docker image to detect inappropriately stored and hard-coded keys. It could have identified excessive permissions, like absence of read-only flags;
- **Architectural Risk Analysis** could have ensured properly set access controls, permissions, and authorization check of the users inside the Docker container;
- **Penetration Testing** together with **Risk-Based Testing** could have helped in testing the system's robustness against escalation of privileges, obtaining a secret key, and using it to decrypt information;
- **Abuse Cases** could have anticipated directory traversal for critical files as well as possible file manipulation. It could have added to functional security requirements, as described next;
- **Security Requirements** could have mandated proper key management and regulated user role-based and least-privilege access to file permissions, files themselves, and directories. As a functional requirement, it would have accounted for the detection and alert of critical file access,

copy, and escalation of privileges. As a non-functional requirement, it should provide auditability of system actions.

Mitigate 3a, SQL Injection (intro): The SQL injection (intro) in the webgoat highlights several attack techniques, including basic injection, numeric injection, SQL chaining (updating/deleting), and privilege escalation. Below are mitigation techniques and software security touchpoints in mitigating the vulnerabilities and how.

Mitigation Strategies

- Prepared statement (with parameterized queries), using `placeholders (?)` ensures user input is treated as data, preventing execution of injected SQL commands. Functions like `setString()` securely bind values, mitigating injection risks.
- Implement stored procedures, predefined queries stored in the database prevent direct SQL manipulation. Unlike dynamic queries, stored procedures reduce the risk of malicious input execution.
- Enforce whitelist input validation, validates input against expected patterns, ensuring only safe values are processed. This is crucial when sorting dynamically (e.g., ASC/DESC), where bind variables aren't feasible.
- Escape All user-supplied input, escaping prevents SQL injection by ensuring user input is treated as data, not SQL commands. In Part I, basic injection, numeric injection, and SQL chaining exploited unescaped input. While useful, attackers can bypass it with encoding tricks (/**/, SELECSELECTT). Combining it with prepared statements and input validation strengthens security, especially in legacy systems without parameterized queries.
- Applying the Principle of least privilege, users should have access only to necessary tables, limiting SQL injection risks like SQL chaining and privilege escalation seen in Part I. This restricts attack impact by preventing unauthorized modifications or deletions. Using database views instead of full table access further enhances security.

Security Touchpoints

- **Code Review** identifies unsafe query concatenation and enforces prepared statements, reducing SQL injection risks early in development.
- **Penetration Testing** simulates real-world SQL injection attacks (e.g., SELECT * FROM employees WHERE last_name = '' OR 1=1; -- from Part I), exposing vulnerabilities before exploitation.
- **Security Requirements** require the prepared statements, input validation, and least privilege, preventing SQL injection at the development phase.
- **Abuse Cases** tests attacker techniques like password reset injection (' OR '1'='1' -- from Part I), ensuring input validation blocks unauthorized access. [6, 7]

Mitigate 3b, SQL Injection (advanced): Advanced SQL injection techniques, including SQL chaining and blind injection, allow attackers to extract sensitive data, manipulate authentication, and explore database structures. Part I demonstrated how attackers used SQLMap, Boolean-based injection, and error responses to infer database behavior and exfiltrate information.

Mitigation Strategy

- Parameterized Queries & Prepared Statements
  Prevents SQL chaining attacks like wrong' OR 1=1; SELECT * FROM user_system_data;-- by treating user input strictly as data.

- Input Validation & Whitelisting
  Blocks blind and Boolean-based SQL injection by restricting input to expected formats, preventing logical condition injections (AND 1=1).
- Principle of Least Privilege
  Ensures users have only necessary permissions, preventing privilege escalation even if an injection attempt succeeds
- Web Application Firewall (WAF)
  Detects and blocks SQLMap-driven and Boolean-based attacks using signature detection, behavioral analysis, and rate limiting.
- Error Handling & Captcha
  Prevents Boolean-based inference attacks by suppressing database errors and limiting automated queries through CAPTCHA and rate limiting.

Security touchpoints

- **Code Review** identifies unsafe query concatenation and ensures prepared statements prevent SQL chaining.
- **Penetration Testing** simulates SQLMap-based and Boolean injection attacks, uncovering vulnerabilities before exploitation.
- **Security Requirements** enforces prepared statements, input validation, and least privilege, blocking query manipulation.
- **Abuse Cases** simulates blind and Boolean-based SQL injection, ensuring defenses like WAF, CAPTCHA, and error handling are effective. [7, 8]

Mitigate 3c, SQL injection (mitigation): SQL Injection (mitigation) on WebGoat shows how improper input handling allows attackers to bypass weak validation mechanisms using comment obfuscation (/**/), keyword manipulation (SELECSELECTT), and dynamic query modification. Part I demonstrated how attackers evade filtering techniques, proving that input validation alone is insufficient.

Mitigation strategy

- Parameterized Queries & Prepared Statements, Ensures that user input is treated strictly as data, preventing command interpretation regardless of obfuscation techniques (a';SELECT/**/*/**/FROM/**/user_system_data;--).
- Strict Input Validation & Whitelisting, Validates user input beyond simple keyword filtering, preventing obfuscation attacks using comment-based (/**/) and keyword-based (SELECSELECTT) evasion tactics.
- Principle of Least Privilege, Restricts database permissions to prevent attackers from executing unauthorized queries, reducing the impact of successful injections.
- Web Application Firewall (WAF), Blocks SQL injection payloads by detecting manipulated keywords, obfuscated commands, and high-frequency malicious queries before reaching the database.

Security Touchpoints

- **Code Review** identifies improper input handling and ensures prepared statements replace dynamic SQL execution. Detects comment obfuscation (/**/) and keyword manipulation (SELECSELECTT) bypasses.
- **Penetration Testing** simulates SQL injection attempts using payloads that manipulate input validation to evaluate filtering robustness and uncover bypass techniques.

- **Security Requirements** defines prepared statements, proper input validation, and WAF integration as mandatory SDLC practices, ensuring SQL security measures are enforced at the development stage.
- **Security Operations** integrates real-time monitoring to detect input validation bypass attempts, preventing SQL injection attacks that leverage obfuscation techniques.
- **Abuse Cases** tests filter evasion strategies, such as inserting modified SQL keywords (SELECSELECTT) and bypassing space restrictions (/**/), to ensure validation mechanisms are resistant to obfuscated injection attempts [8, 9]

Mitigate 7a, Software & Data Integrity – Insecure Deserialization: Insecure deserialization happens when rebuilding an object using a certain data structure format, the data is not properly validated and therefore allows attacker-controlled data to be deserialized.

Mitigation Strategy

- Avoid deserializing objects from untrusted sources. If deserialization is required, enforce strict data validation such as using cryptographic signatures and allowing only explicitly permitted types of frameworks [10].
- Run deserialization operations in a restricted environment with minimal privileges to limit the impact of exploits.
- Use libraries that support safe deserialization techniques. Each programming language has libraries and functions that are suggested safe, such as Jackson with @JsonCreator annotations to enforce whitelisted classes [11].
- Use security logging and monitoring to detect abnormal deserialization attempts.

Security Touchpoints

- **Code Review**
    - Identifies areas where deserialization occurs, and ensures that only necessary data types are allowed and safe methods, classes, and frameworks are used.
    - Helps detect unsafe usage of ObjectInputStream, Gson, Jackson, or Yaml.load(), which can be exploited for remote code execution.
- **Penetration Testing**
    - Simulates deserialization attacks to test if the application is vulnerable.
    - Validates the effectiveness of implemented mitigations by testing how the system handles unexpected serialized input.
- **Risk-Based Security Testing**
    - Focuses on security-critical components where deserialization occurs, ensuring that no untrusted input is processed insecurely.
    - Uses attack patterns (ex. fuzzing) to determine if unsafe deserialization can lead to code execution.

Mitigate 8a, Security Logging Failures – Log Spoofing and Log Bleeding: In this particular WebGoat vulnerability exploit, logging is enabled, but the logs were not recorded securely. Specifically, the admin password can be located in the application logs. After locating the password and decoding the hash or encrypted value, the password can be acquired, allowing unauthorized access.

Mitigation Strategy [12]

- Avoid logging sensitive information. If sensitive information is necessary to be logged, avoid

including sensitive information in logs without encryption.
- To prevent injection attacks, sanitize, encode, and validate log data, use cryptographic hash functions to verify log integrity, and follow a consistent log format to not only facilitate analysis and monitoring but also prevent unexpected, potentially dangerous format.
- Limit log access to authorized personnel through access controls

Security Touchpoints

- **Code Review**
    - Detects instances where sensitive information is improperly logged and are properly encoded to prevent log injection attacks.
    - Verifies the use of logging frameworks that support secure log handling.
- **Penetration Testing & Abuse Cases**
    - Identifies whether an attacker can retrieve sensitive information from logs.
    - Tests for log injection vulnerabilities where attacker-controlled data could be used to manipulate logs.
- **Security Operations**
    - Ensures logs are encrypted and access-controlled to prevent unauthorized modifications or access.
    - Establishes incident response plans based on log analysis to detect and respond to security threats

Mitigate 9A, Cross–site request forgeries (CSRF): CSRF exploits the trust between a web application and an authenticated user, allowing attackers to execute unintended actions on behalf of victims. In Part I, we examined how a POST request with a hardcoded validation parameter was exploited to submit unauthorized reviews using an authenticated session.

Mitigation Strategy

- Anti-CSRF Tokens, requires each request to include a unique, server-generated token, ensuring that only legitimate requests are processed. In Part I, the lack of dynamic validation parameters made the application vulnerable—tokens would prevent forged submissions.
- SameSite Cookies, restricts cookies from being sent with cross-origin requests, preventing attackers from executing unauthorized actions through external sites. Part I highlighted how the absence of SameSite attributes allowed external request submissions, which this mitigation would block.
- Double Submit Cookies, ensures that a CSRF token is stored in both a cookie and a request parameter, requiring them to match for validation. Since attackers cannot access cookies, this method prevents forged requests from being processed.
- Validating Origin & Referer Headers, ensures that requests originate from trusted sources, blocking unauthorized actions initiated from malicious sites. Part I showed that CSRF attacks exploited unverified request sources, which origin validation would prevent.
- Enforcing Proper HTTP Method ensures state-changing actions (e.g., updates, transactions) require POST, PUT, or DELETE requests rather than GET, which can be exploited for CSRF. In Part I, attackers leveraged an unprotected POST request, which could have been blocked with strict method enforcement.

Security Touchpoints

- **Code Review** identifies missing CSRF tokens, improper cookie settings, and unvalidated request sources, ensuring that vulnerable requests are properly secured.

- **Penetration Testing** simulates CSRF attacks to test whether tokens, SameSite cookies, and referer validation effectively prevent unauthorized requests.
- **Security Requirements** establishes CSRF protection mechanisms as a mandatory security measure in development, preventing unauthorized state-changing requests.
- **Architectural Risk Analysis** identifies design flaws that expose applications to CSRF, ensuring anti-CSRF tokens and authentication checks are required for sensitive actions.
- **Risk-Based Security Testing** evaluates CSRF defenses under different attack conditions, ensuring session expiration, token validation, and HTTP method restrictions are enforced.
- **Abuse Cases** tests attacker behavior by simulating forged request attempts, validating whether CSRF protections effectively block unauthorized actions.
- **Security operations** monitors request patterns and logs to detect suspicious CSRF attempts, ensuring early mitigation of fraudulent activity. [13]

Mitigate 9b, Server-Side Request Forgery (SSRF): SSRF occurs when an application processes user-supplied URLs, allowing attackers to manipulate requests [14]. In this particular WebGoat vulnerability exploit, hidden input fields control the requested resource. By modifying these fields, attackers can redirect requests to unauthorized destinations.

Mitigation Strategy [15, 16]

- Restrict access to internal networks and metadata services from external web applications, and only allow requests from pre-approved domains and IP addresses.
- Sanitize and validate user-supplied URLs before making requests.
- Restrict HTTP methods (ex. PUT, DELETE) unless required.
- Limit and track outbound requests to detect suspicious patterns, ex. configure a WAF to detect and block SSRF payloads.

Security Touchpoints

- **Code Review**
    - Identifies areas where external requests are made and ensures proper validation on user-controlled URLs is enforced.
    - Reviews API request-handling logic to verify the application does not allow unrestricted access to internal services.
- **Architectural Risk Analysis**
    - Helps map out potential SSRF attack surfaces and determine how an attacker could misuse legitimate application functionality.
    - Assesses the security of third-party services that the application interacts with.
- **Penetration Testing & Abuse Cases**
    - Actively tests the application for SSRF vulnerabilities using tools such as Burp Suite to attempt unauthorized requests.
    - Identifies whether the application can be manipulated into requesting internal resources, such as cloud metadata endpoints.
    - Confirms the effectiveness of firewall rules and whitelisting restrictions.

Part III: Appendix

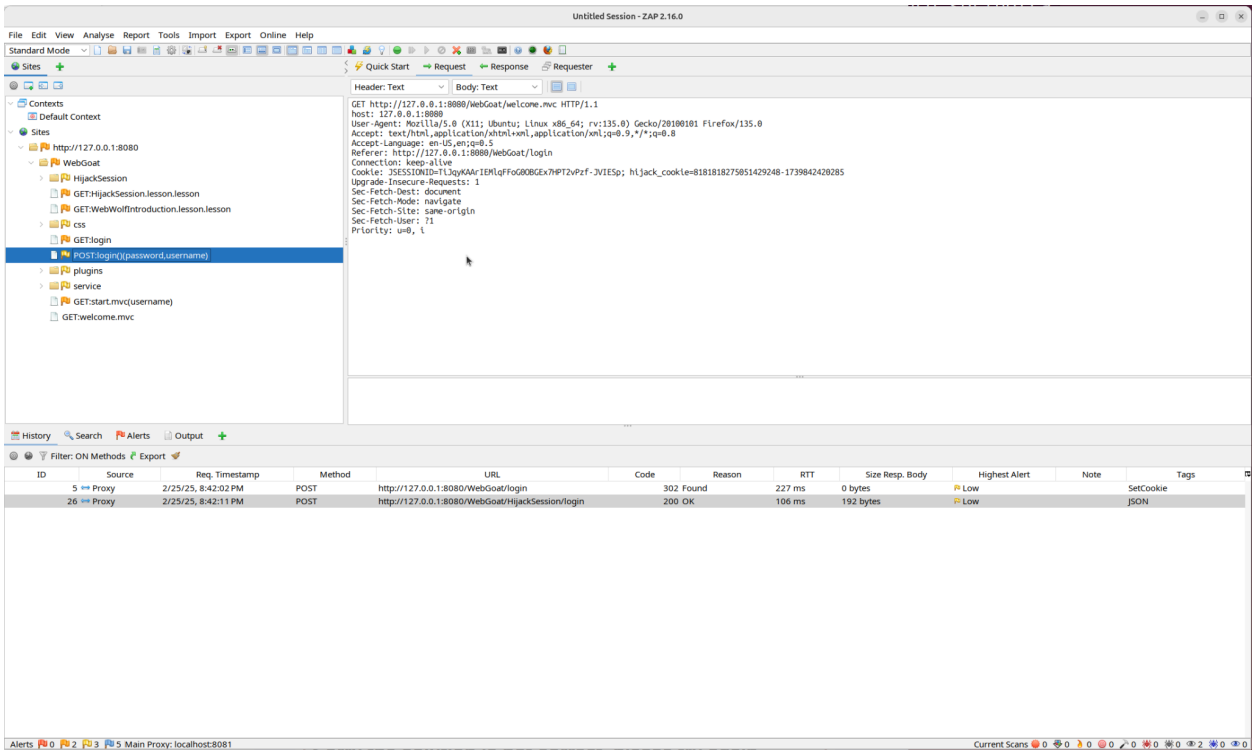


Figure 1: Intercepted initial POST request showing the *hijack_cookie* in the Cookie header.

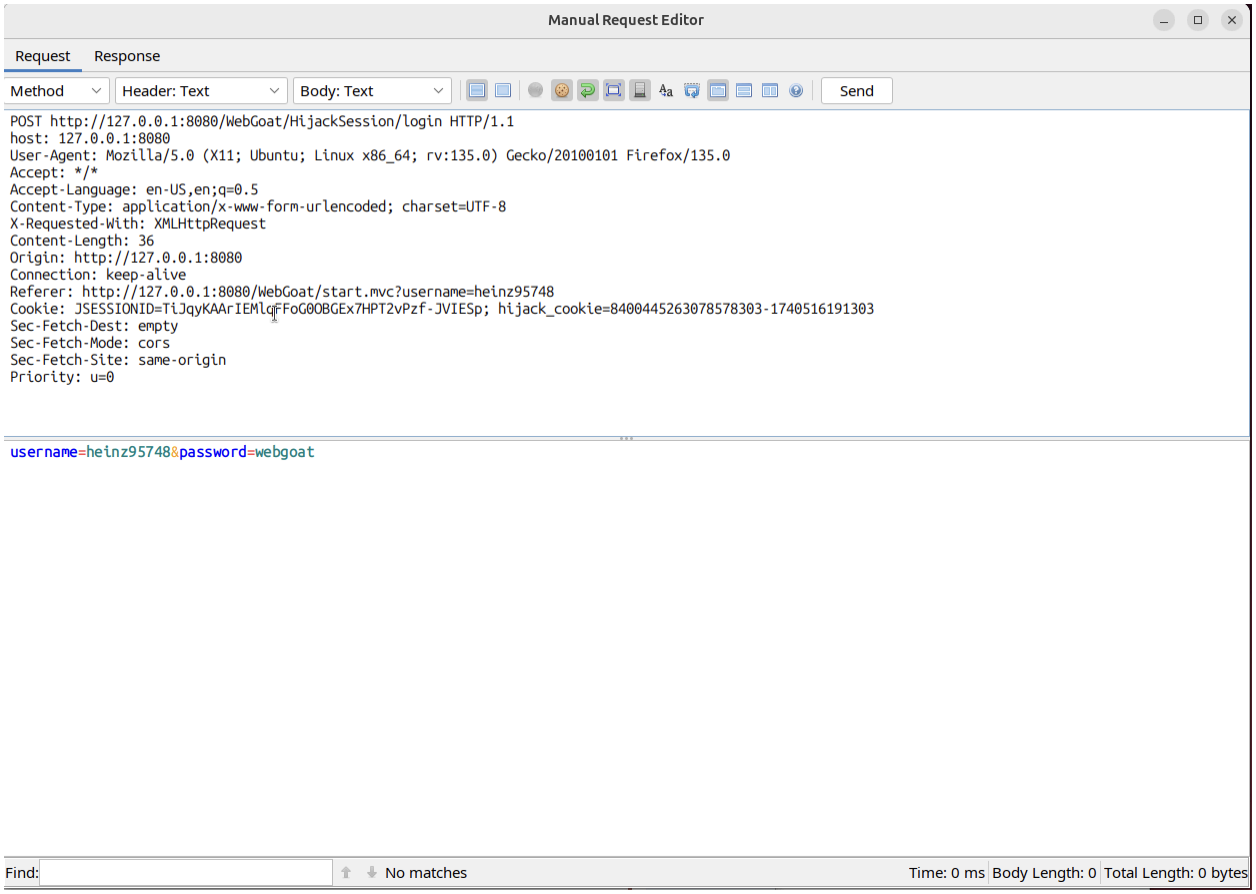


Figure 2: Manual Request Editor in ZAP with the payload adjusted for brute forcing.

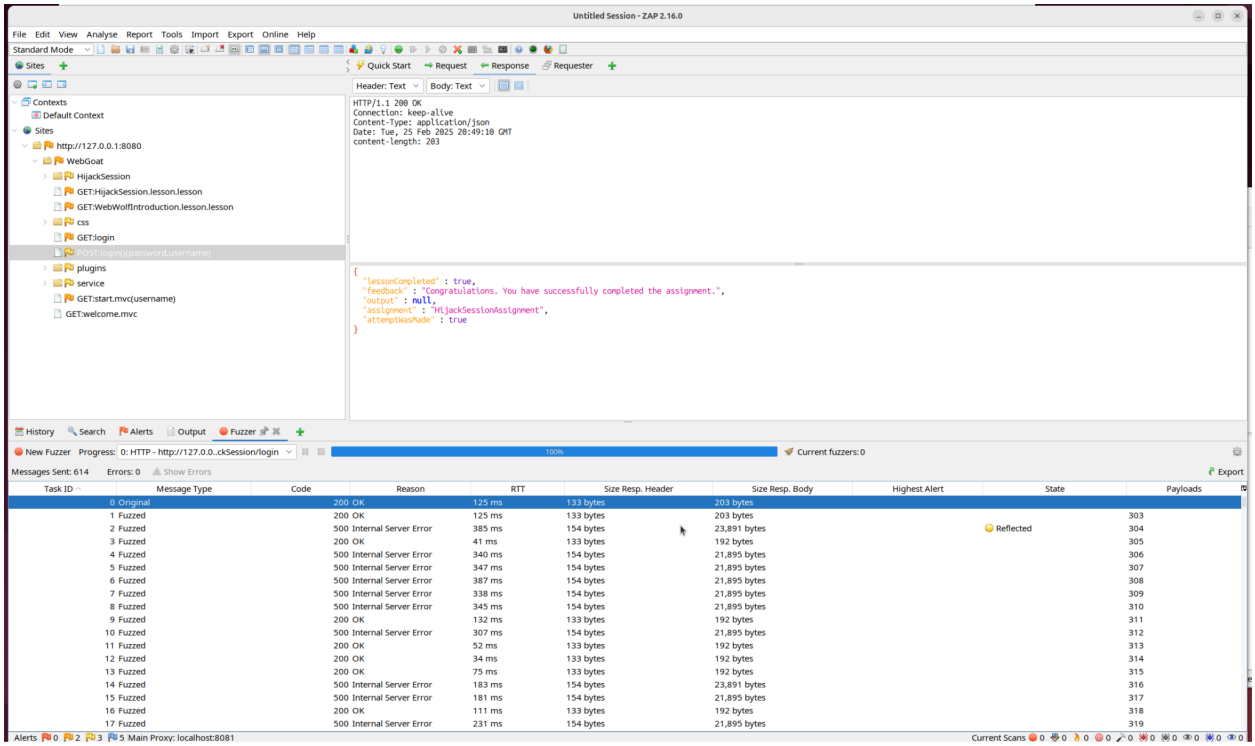


Figure 3: Fuzzer output displaying the successful response containing the correct session token (with the success message).

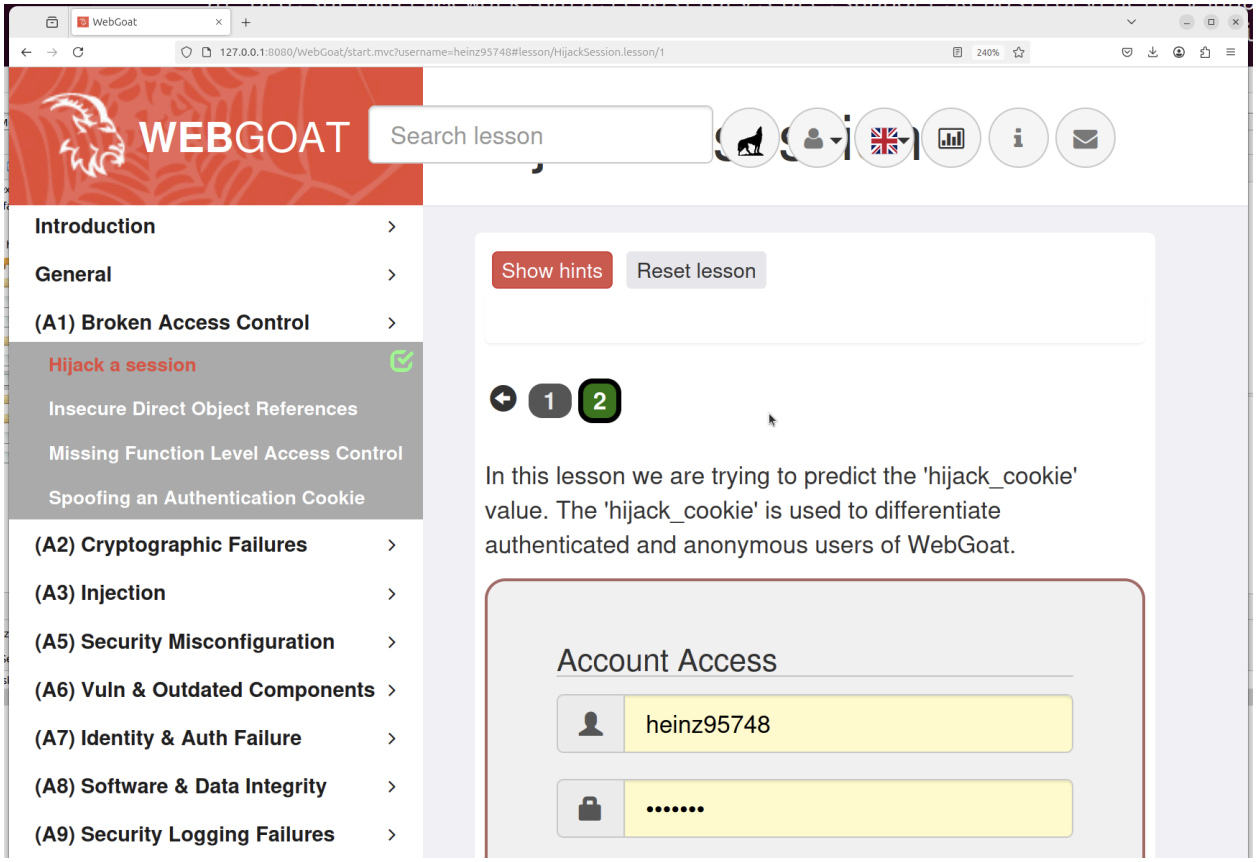


Figure 4: Final screen indicating that the activity was completed successfully.

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/bin/false
webgoat:x:1000:1000::/home/webgoat:
~
~
~
~
~
~
```

Figure 5: Editing the `/etc/passwd` file

```
Last login: Wed Feb 12 20:35:08 on console
tanner@tanners-mbp ~ % docker run -d webgoat/assignments:findthesecret
zsh: command not found: docker
tanner@tanners-mbp ~ % docker run -d webgoat/assignments:findthesecret
Unable to find image 'webgoat/assignments:findthesecret' locally
findthesecret: Pulling from webgoat/assignments
5d9d21aca480: Download complete
190ddf324545: Download complete
e6d9d96381c8: Download complete
ff2e10214d79: Download complete
4cf180de4a1f: Download complete
d6419a981ec6: Download complete
0a126fb8ec28: Download complete
5e6ec7f28fb7: Download complete
1cf4e4a3f534: Download complete
Digest: sha256:3fba41f35dbfac1daf7465ce0869c076d3cdef017e710dbec6d273cc9334d4a6
Status: Downloaded newer image for webgoat/assignments:findthesecret
WARNING: The requested image's platform (linux/amd64) does not match the detected host platform (linux/arm64/v8) and no specific platform was requested
a52fb3757d10ebb13110a1ef746c63e175883496abcf9919dff5977e7597e7ea
tanner@tanners-mbp ~ % docker cp <a52fb3757d10ebb13110a1ef746c63e175883496abcf9919dff5977e7597e7ea>:/etc/passwd pass.txt
zsh: no such file or directory: a52fb3757d10ebb13110a1ef746c63e175883496abcf9919dff5977e7597e7ea
tanner@tanners-mbp ~ % docker cp a52fb3757d10ebb13110a1ef746c63e175883496abcf9919dff5977e7597e7ea:/etc/passwd pass.txt
Successfully copied 2.56kB to /Users/tanner/pass.txt
tanner@tanners-mbp ~ % vim pass.txt
tanner@tanners-mbp ~ % docker cp pass.txt 098f135c6b08:/etc/passwd
Successfully copied 2.56kB to 098f135c6b08:/etc/passwd
Error response from daemon: No such container: 098f135c6b08
tanner@tanners-mbp ~ % docker cp pass.txt a52fb3757d10ebb13110a1ef746c63e175883496abcf9919dff5977e7597e7ea:/etc/passwd
Successfully copied 2.56kB to a52fb3757d10ebb13110a1ef746c63e175883496abcf9919dff5977e7597e7ea:/etc/passwd
tanner@tanners-mbp ~ % docker exec -it a52fb3757d10ebb13110a1ef746c63e175883496abcf9919dff5977e7597e7ea /bin/bash
root@a52fb3757d10:/# cd /root/
root@a52fb3757d10:~# ls
default_secret
root@a52fb3757d10:~# cat default_secret
ThisIsMySecretPassw0rdF0rY0u
root@a52fb3757d10:~# echo "U2FsdGVkX199jgh5oANElFdtCxIEvdEvciLi+v+5loE+VCuy6Ii0b+5byb5DXp32RPmT02Ek1pf55ctQN+DHbwCPiVRfFQamDmbHBUpD7as=" | openssl enc -aes-256-cbc -d -a -kfile default_secret
Leaving passwords in docker images is not so secureroot@a52fb3757d10:~#
```

Figure 6: Retrieving secret from container process

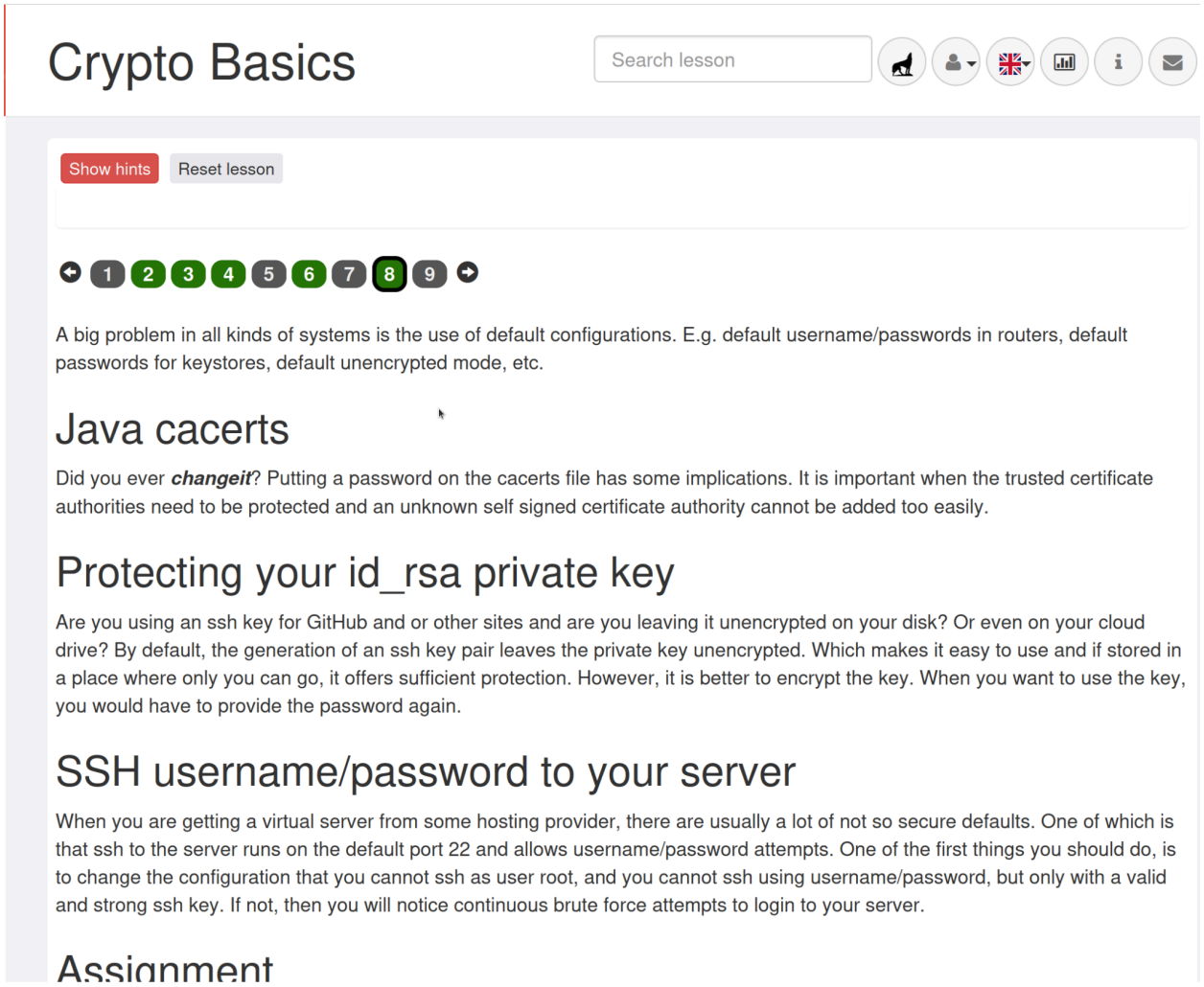


Figure 7: Full activity completion proof

```
linux-vm-1@linux-vm-1:~$ openssl rsa -in test.pub -pubin -modulus -noout
Modulus=D4ECC7E3DDC84F67F42EF608A8B752D04ED0ED28AB2513668302154436DEA7E529410F232271F5F421036F8728DF8F622B92C0FB8828
A862F7F93033F704BBC35B3B5D9A655F738EA0B94F37E0B4CA47E23D02A1BFDA900A5F335B0119AD4D7798A968C086A9ABDE3464A180F35CC668
D48FF2D6989368D65FDDB9787DD97618A2EA817DC2F0D87FD0C47E663C17E90C7073C815156399A650EB833DBCBAF3F6A5188C5263BF6B40291F
89074940564ADA2BD30B82B02236872B3AB6918956157905EE53D71545C2B5ECF6F0D88CEAB1E511693F3280DC16FA5B249B7A4CBA6F9B112E23
3D20432B89BD22B63D342E8E2A66B2FB62F2FD63B029223330701E87
linux-vm-1@linux-vm-1:~$ echo -n "D4ECC7E3DDC84F67F42EF608A8B752D04ED0ED28AB2513668302154436DEA7E529410F232271F5F421
036F8728DF8F622B92C0FB8828A862F7F93033F704BBC35B3B5D9A655F738EA0B94F37E0B4CA47E23D02A1BFDA900A5F335B0119AD4D7798A968
C086A9ABDE3464A180F35CC668D48FF2D6989368D65FDDB9787DD97618A2EA817DC2F0D87FD0C47E663C17E90C7073C815156399A650EB833DBC
BAF3F6A5188C5263BF6B40291F89074940564ADA2BD30B82B02236872B3AB6918956157905EE53D71545C2B5ECF6F0D88CEAB1E511693F3280DC
16FA5B249B7A4CBA6F9B112E233D20432B89BD22B63D342E8E2A66B2FB62F2FD63B029223330701E87" \
| openssl dgst -sign test.key -sha256 | base64
XstizyOeSOak3iq6vxMaeGB3qf9Uh38heg+vOdlOPqMl7c70oUaO77FLNkFIRfn2NNgTeQKME/w7
t2NqMIAW+ccuNJ42MjhclZ31L5A+2N9r8T8wvQwmmRUYm87qaTC3zQtCD5cDReG/ZIOjBgXAFmDP
X4m2tUebNchB2w+4zzFoPLPpCHa37YSA8tCbr/OsJMxp/tXKSenRqka9EsBdUiV7ohImKOVdDZyT
txImmlgRvKDUq/V1HTLuWzY5GQJUruUuacxL4fpRjm+iDG8HGGnXHVRzjoTr6i4Y7irpWbmKV7vL
bW2bElZSD0wxJaeY/P7NrnjV/5RZ5SSv6OKJfw==
linux-vm-1@linux-vm-1:~$
```

Figure 8: Crypt Basics 6 activity completion

Terminal

```java
import java.io.*;
import java.util.Base64;
import org.dummy.insecure.framework.VulnerableTaskHolder;

public class GeneratePayload {
    public static void main(String[] args) throws Exception {
        // Create the exploit object
        VulnerableTaskHolder payload = new VulnerableTaskHolder("exploit", "sleep 5");

        // Serialize to payload.ser
        try (FileOutputStream fos = new FileOutputStream("payload.ser");
             ObjectOutputStream oos = new ObjectOutputStream(fos)) {
            oos.writeObject(payload);
        }

        // Convert to Base64
        byte[] serializedData = readFile("payload.ser");
        String base64Payload = Base64.getEncoder().encodeToString(serializedData);

        // Print Base64 string (copy this and paste into WebGoat form)
        System.out.println("Base64 Payload:");
        System.out.println(base64Payload);
    }

    private static byte[] readFile(String filename) throws IOException {
        File file = new File(filename);
        byte[] data = new byte[(int) file.length()];
        try (FileInputStream fis = new FileInputStream(file)) {
            fis.read(data);
        }
        return data;
    }
}
```

Figure 9: VulnerableTaskHolder.java Code

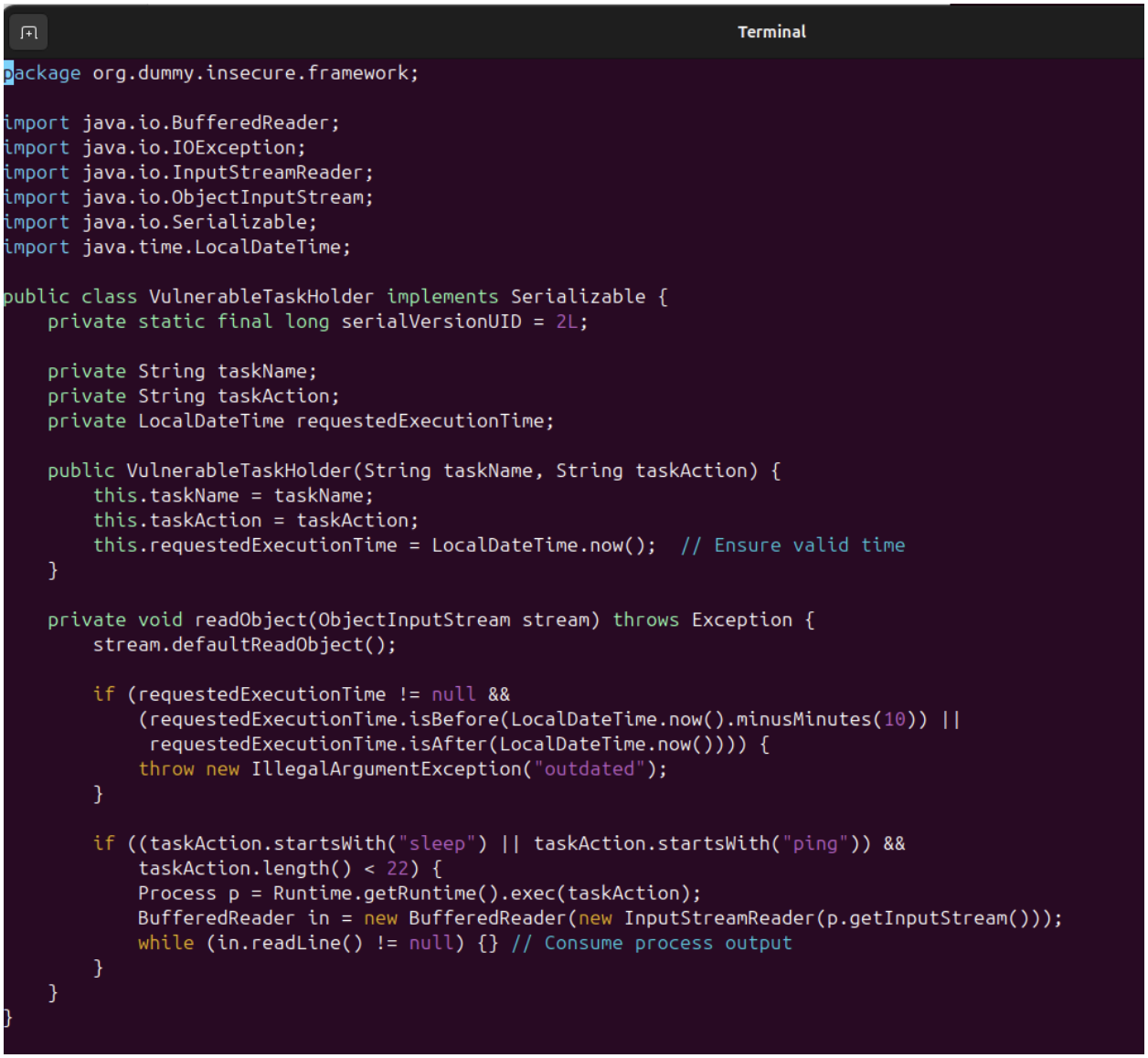


Figure 10: Attack.java Code

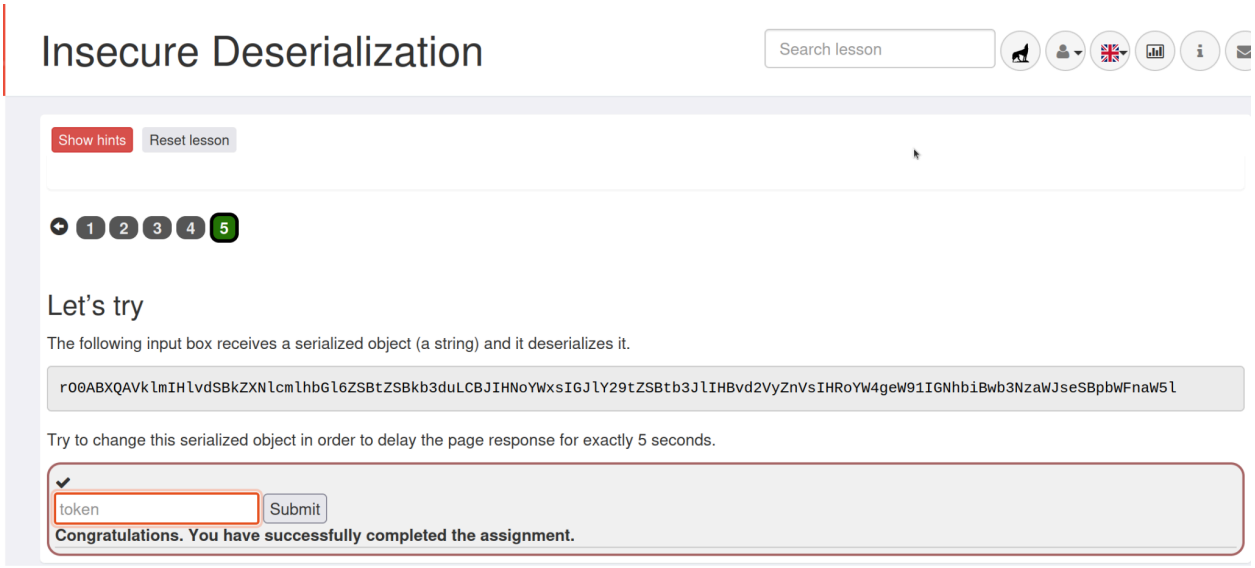


Figure 11: Activity completion proof

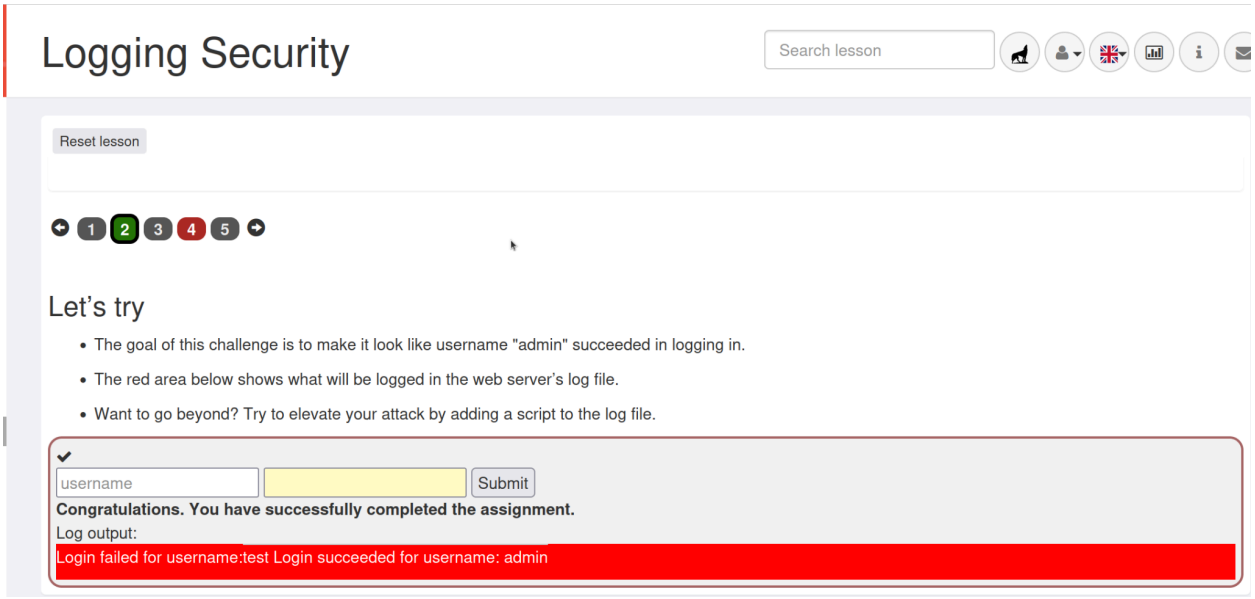


Figure 12: Manipulated log entry where the system records a successful "admin" login

```
et 'hibernate.transaction.jta.platform' to enable JTA platform integration)
2025-02-13T19:04:14.828Z  INFO 4336 --- [           main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory fo
r persistence unit 'default'
2025-02-13T19:04:14.901Z  INFO 4336 --- [           main] o.o.w.lessons.logging.LogBleedingTask    : Password for admin: ZDZmNTg0OGQtZWE3MS0
0ODZmLWJkNjktNjUwZjEwMGEzOTBj
```

Figure 13: Application log displaying the inadvertently exposed admin credentials

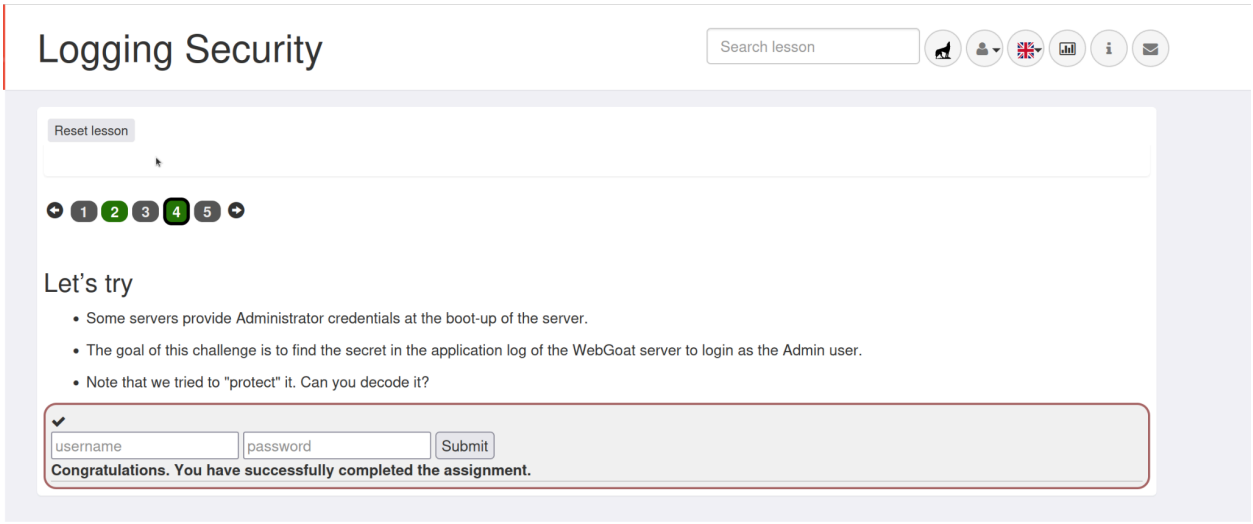


Figure 14: Proof of activity completion

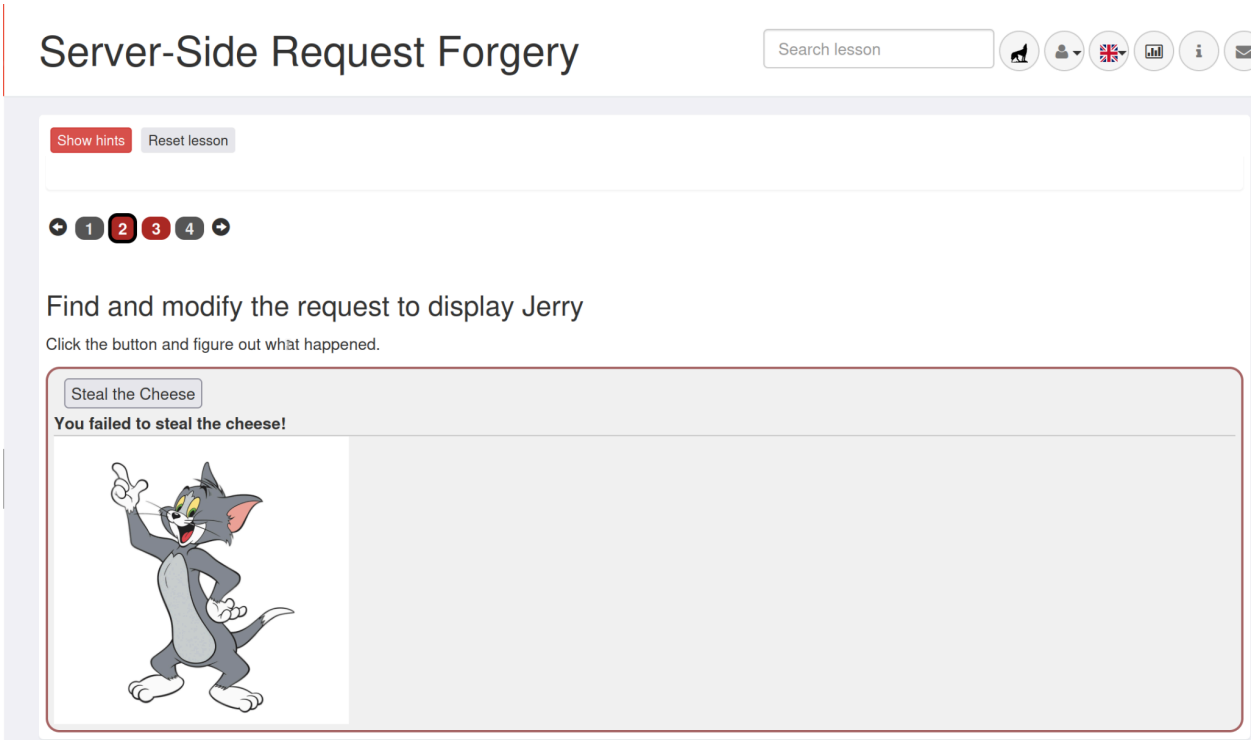


Figure 15: Original tommy.png

Figure 16: Hidden tommy.png field that was changed to jerry

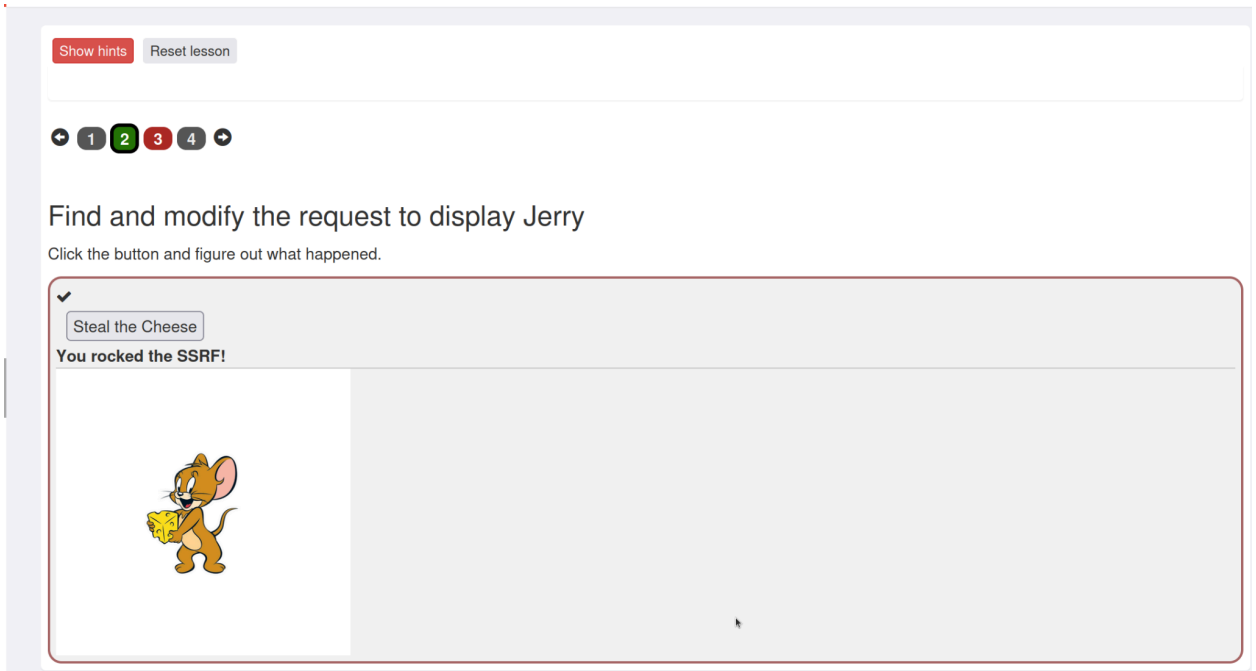


Figure 17: Changed Jerry image

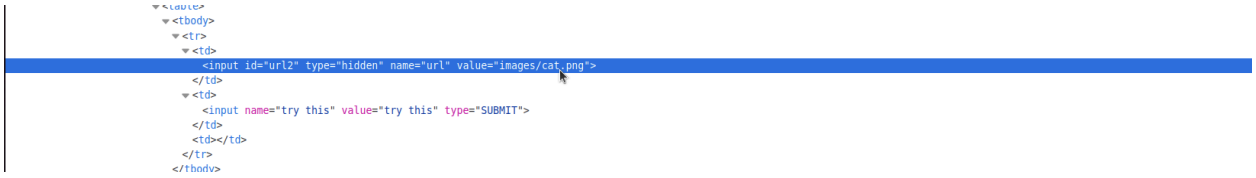


Figure 19: Hidden field

Figure 20: Inserted URL

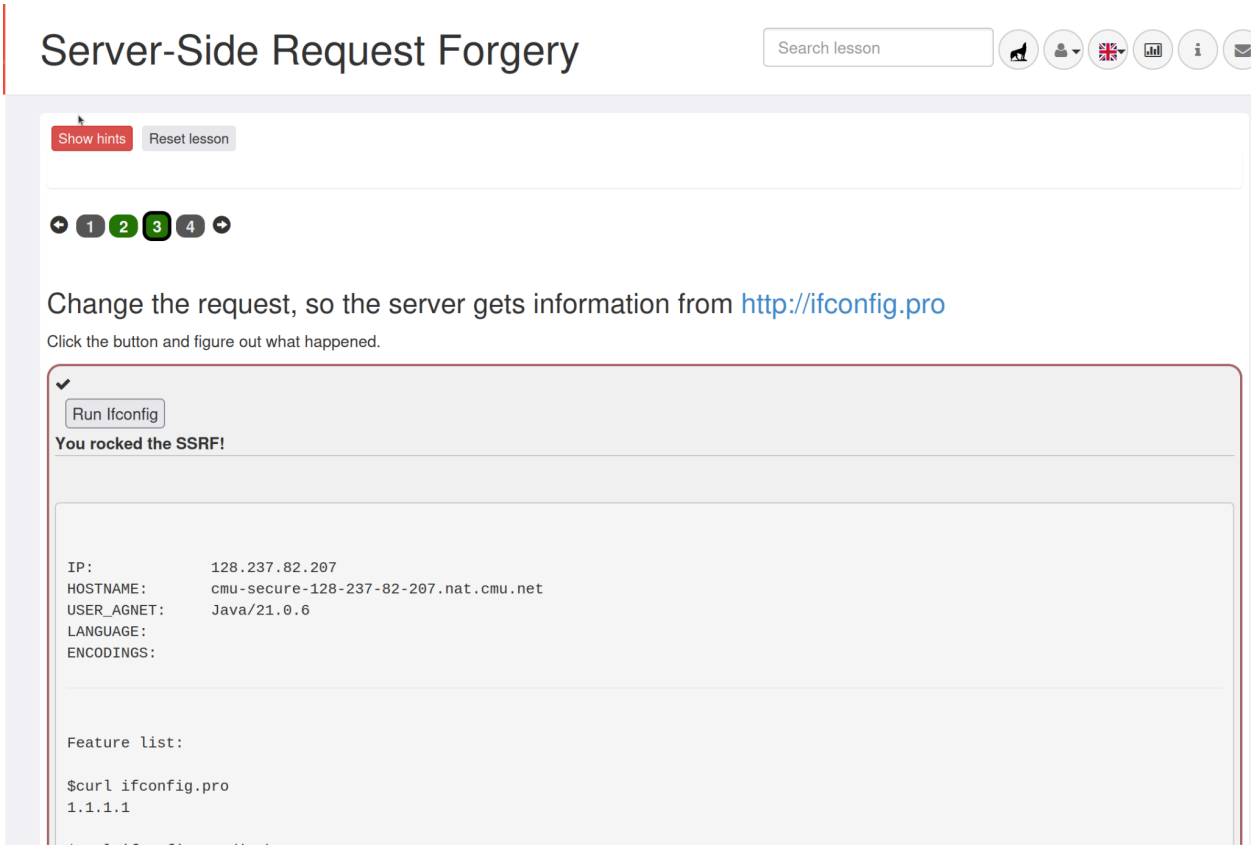


Figure 21: Revealed config and activity completion proof

References

[1] *Invicti. (2022, July 28). Cookie Security Flags | Learn AppSec. I*nvicti. https://www.invicti.com/learn/cookie-security-flags/

[2] *What is Cookies Hacking | Risk & Protection Techniques | Imperva.* (n.d.). Learning Center. https://www.imperva.com/learn/application-security/cookies-hacking/

[3] *Strategies to Limit API Access in Salesforce with User Roles. (2024).* Reco.ai. https://www.reco.ai/hub/limit-api-access-in-salesforce-with-user-roles

[4] *OWASP. (2013). Insecure Direct Object Reference Prevention · OWASP Cheat Sheet Series.* Owasp.org. https://cheatsheetseries.owasp.org/cheatsheets/Insecure_Direct_Object_Reference_Prevention_Cheat_Sheet.html

[5] *Manage sensitive data with Docker secrets.* (2022, August 12). Docker Documentation. https://docs.docker.com/engine/swarm/secrets/

[6] Kao, D.-Y., Lai, C.-J., & Su, C.-W. (2018). A framework for SQL injection investigations: Detection, investigation, and forensics. *IEEE Systems, Man, and Cybernetics (SMC'18)*. https://doi.org/10.1109/SMC.2018.00483

[7] Shar, L. K., & Tan, H. B. K. (2013). *Defeating SQL Injection.* IEEE Computer Society.

[8] Paul, A., Sharma, V., & Olukoya, O. (2024). SQL injection attack: Detection, prioritization & prevention. *Journal of Information Security and Applications, 85*, 103871. https://doi.org/10.1016/j.jisa.2024.103871

[9] Shar, L. K., & Tan, H. B. K. (2013). Defeating SQL injection. *IEEE Computer Society*.

[10] *Insecure deserialization | tutorials & examples*. (n.d.). Snyk Learn. Retrieved February 28, 2025, from https://learn.snyk.io/lesson/insecure-deserialization/

[11] *Lcnc-sec-10: Security logging and monitoring failures | owasp foundation*. (n.d.). Retrieved February 28, 2025, from https://owasp.org/www-project-top-10-low-code-no-code-security-risks/content/2022/en/LCNC-SEC-10-Security-Logging-and-Monitoring-Failures.html

[12] *Owasp developer guide | implement security logging and monitoring checklist | owasp foundation*. (n.d.). Retrieved February 28, 2025, from https://owasp.org/www-project-developer-guide/draft/design/web_app_checklist/security_logging_and_monitoring/

[13] Kour, P. (2020). *A Study on Cross-Site Request Forgery Attack and its Prevention Measures. Int. J. Advanced Networking and Applications,* 12(2), 4561-4566.

[14] *C10: Stop server side request forgery—Owasp top 10 proactive controls*. (n.d.). Retrieved February 28, 2025, from https://top10proactive.owasp.org/the-top-10/c10-stop-server-side-request-forgery/

[15] *A10 server side request forgery (Ssrf)—Owasp top 10:2021*. (n.d.). Retrieved February 28, 2025, from https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%28SSRF%29/

[16] *Server side request forgery prevention—Owasp cheat sheet series*. (n.d.). Retrieved February 28, 2025, from https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html