Lab 7.pdf

Sonal Chandrakant Hasbi

I. Abstract

This lab performs static reverse engineering of an iOS application package (IPA) to evaluate insecure client-side storage practices. By unpacking the IPA, inspecting the bundle structure, analyzing the Mach-O binary, and examining property lists, I identified a secret marker embedded in a binary plist within the .app bundle (PL.plist). The work demonstrates how easily "secrets" can be recovered from app resources without executing the app, underscoring why plist files are not appropriate for sensitive data. Negative findings (e.g., no obvious hardcoded URLs) are documented. The workflow is reproducible with standard Unix utilities and Python's built-in plistlib.

II. Introduction

This lab focused on exploring and exploiting authentication vulnerabilities in web applications using WebGoat 7.1 and WebGoat v2023.8. The goal was to understand common authentication weaknesses, such as poor password management, weak password reset mechanisms, misconfigured multi-level logins, and insecure JSON Web Token (JWT) handling. By performing these exercises, we gained practical insights into how attackers can exploit these flaws and how developers can implement more secure authentication controls.

III. Tools & Environment

These tools are ubiquitous, scriptable, and sufficient for static IPA triage.

1) WebGoat 7.1 and WebGoat v2023.8 for vulnerable environments.
2) OWASP ZAP/Burp Suite for intercepting and modifying requests.
3) Online password strength tool (e.g., https://howsecureismypassword.net) for entropy analysis.

IV. Methodology

The lab was divided into two parts:

PART I : WebGoat 7.1 – Authentication Flaws For Part I we used WebGoat 7.1 as the vulnerable target and Burp Suite as the intercepting and manipulation tool. Burp's Proxy captured traffic, Repeater let us iterate on requests, Decoder/Comparer assisted with payload inspection, and Intruder was used when we needed automated variations. All interactions were conducted in a controlled VM environment against the WebGoat exercises; evidence for each step was captured as screenshots from Burp and WebGoat.

1. Password Strength – Used an online password-strength checker to analyze password entropy and estimate time-to-crack values.

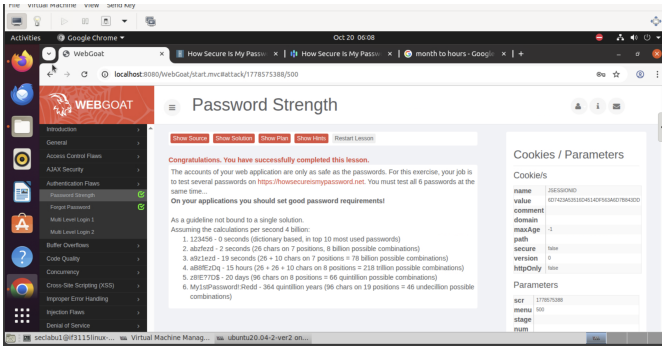

Fig. 1: Password strength

2. Forgot Password –Explored insecure password reset logic to understand how predictable recovery flows can be exploited.

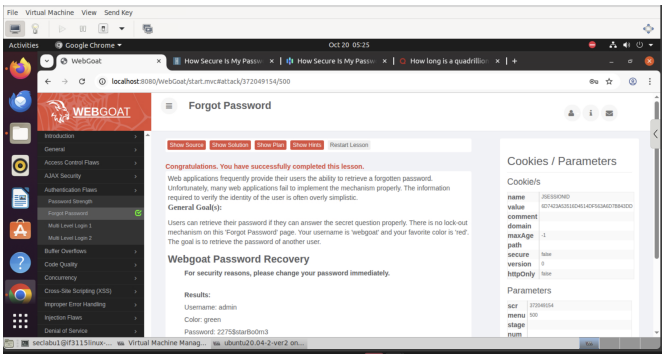

Fig. 2: Forgot Password

3. Multiple Level Login 1 & 2 – Tested multi-level authentication configurations to observe logic flaws that allow privilege escalation or authentication bypass. For both multi-level login exercises we captured the multi-step authentication flow in Burp. Using Breakpoints and Repeater we inspected hidden form fields and step indicators (e.g., step, level, authStage) and tested tampering (changing step=2 → step=3, forcing role=admin, etc.). We also copied session cookies from low-privilege

PART II : WebGoat v2023.8 – (A7) Identity & Authentication Failure For Part II we used OWASP ZAP as the intercepting proxy and manipulation tool and WebWolf's decoder/encoder utilities to inspect and modify token contents. ZAP captured HTTP requests/responses (Breakpoints, HTTP History, Manual Request Editor) so we could pause traffic, alter Authorization headers/cookies, and resend crafted messages.

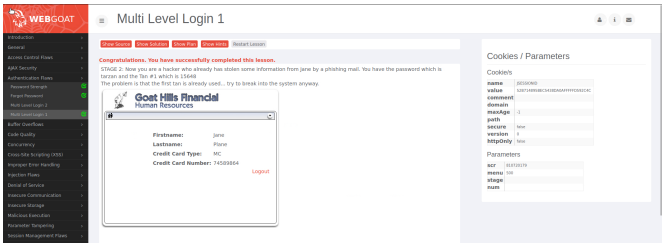

Fig. 3: Multi level login 1

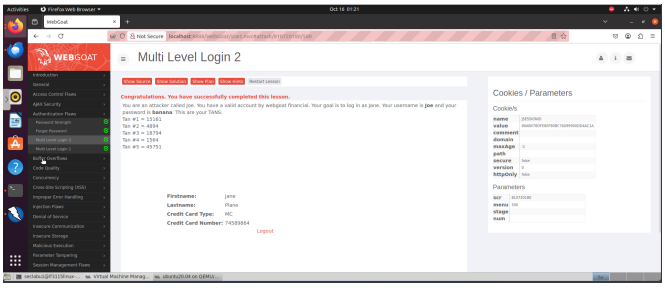

Fig. 4: Multi level login 2

WebWolf's decoder/encoder was used to base64url-decode JWT components and to re-encode modified header/payload pairs when forging tokens. All experiments were performed against WebGoat v2023.8 in a controlled lab environment; evidence (requests, responses, decoded tokens) was collected as screenshots. We explored vulnerabilities aligned with the OWASP A7 category:

1. Authentication Bypass – Attempted to log in using crafted input to bypass authentication. We proxied a normal login by using webwolf to decode the encoded text and find the username

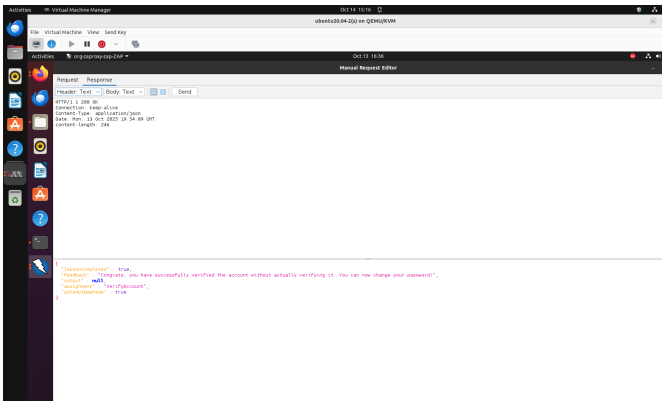

Fig. 5: Authentication Bypass

2. JWT Tokens – Investigated how JWTs work, manipulated header and payload components, and observed how missing signature validation can lead to unauthorized access. After logging in we located the Authorization: Bearer <token> header in ZAP's History, copied the token, and used WebWolf's decoder to inspect the header and payload (alg, claims, exp, role). We then edited the payload (for example changing role:user → role:admin) and used WebWolf to re-encode the header+payload (and experimented with unsigned alg:none variants). The forged token replaced the original Authorization header in ZAP's Request Editor and was resent to privileged endpoints; successful elevation or different server behavior was recorded.
Token signing tests: When testing HS256 re-signing attempts we used a local helper (or jwt libraries) where required, but primary decoding/encoding steps were performed with WebWolf to ensure correct base64url formatting. For the alg=none check we removed the signature component after re-encoding and observed whether the server accepted unsigned tokens. ZAP responses showed whether the server validated signatures and claim values.

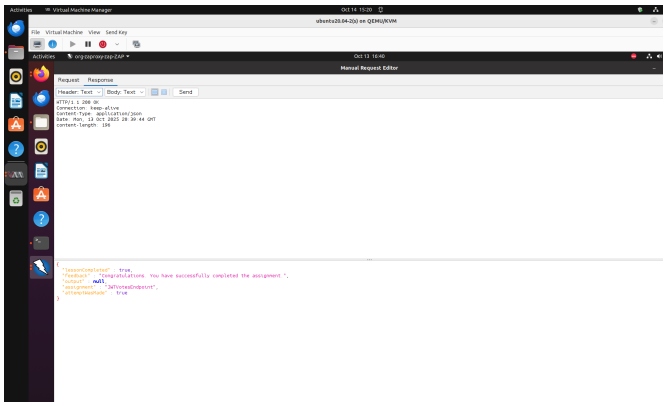

Fig. 6: JWT tokens - task 4

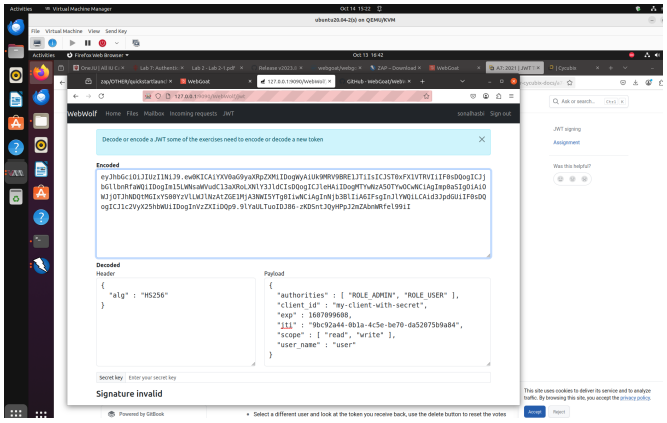

Fig. 7: JWT tokens - task 6

2. Password reset - We triggered the "Forgot password" flow in the browser while ZAP proxied the request, captured the reset response (which contained the simulated reset link/token), and inspected the token format using WebWolf. We tested token reuse, token modification (if decodeable), and direct POSTs to the reset endpoint with altered tokens via ZAP's Request Editor. Source-code inspection in WebGoat confirmed whether tokens were generated with secure RNG, stored/hashed server-side, had expiry, and were invalidated after use.

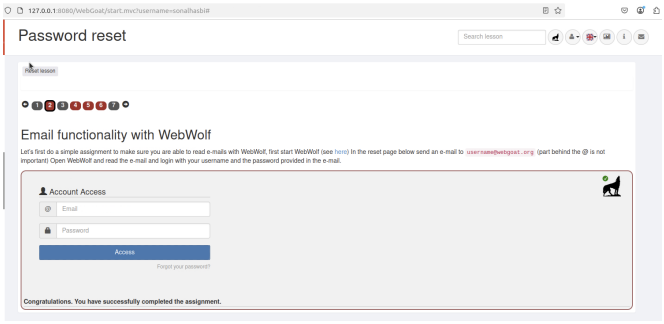

Fig. 8: Password reset - task 2

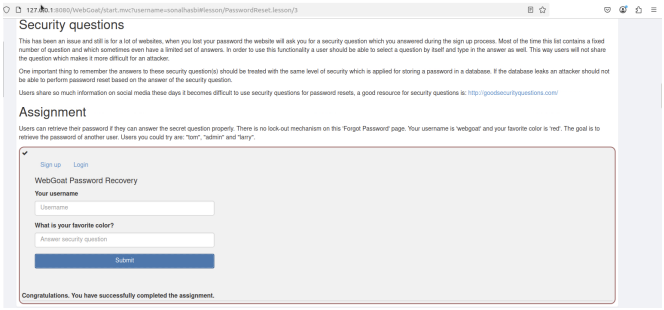

Fig. 9: Password reset- task 4

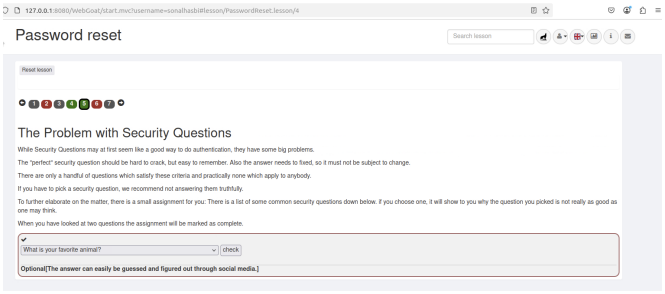

Fig. 10: Password reset - task 5

V. Results

Password Strength

Different passwords exhibited large variations in cracking time due to entropy and complexity. For example, "Password123" could be cracked in seconds, while a random combination like "T&5zR8Lp!v3Q" took centuries to brute-force. Takeaway: Longer passwords with diverse character sets (uppercase, lowercase, digits, symbols) increase entropy and reduce the success rate of brute-force attacks.

Forgot Password

The exercise revealed that many applications fail to validate user identity securely before sending reset links. We observed how predictable recovery tokens or insecure validation parameters can allow unauthorized password resets.

Multiple Level Login 1 & 2

By analyzing login logic, we discovered that authentication layers can be misconfigured, allowing users to access restricted accounts by directly manipulating parameters or skipping intermediate validation steps. This highlights the importance of proper session management and consistent credential checks at each level.

Authentication Bypass

In the modern WebGoat version, we successfully bypassed login checks using specific payload manipulations. This reinforced the concept that flawed backend validation can make even complex authentication UIs insecure.

JWT Tokens (Tasks 1–8)

We explored how JWTs encode user identity and authorization data. Through tampering, we demonstrated how omitting or replacing the signature verification can result in successful unauthorized access. In Step 8:

1. First code snippet result: Throws an exception at line 12 - inspecting the task's source-code path shows the snippet enters an error path (an unhandled/exception-raising branch) at the line referenced, so the observable result is an exception being thrown.

2. Second code snippet result: Invokes the removeAllUsers method at line 7 - reasoning about the control flow in the provided code shows the condition that leads to the removeAllUsers call is satisfied in the second snippet, so execution reaches and invokes that method.

This illustrates how logical flaws in JWT handling can lead to critical privilege escalation.

Password Reset (Tasks 1–5)

By inspecting the source code, we learned that weak validation in password reset flows could expose users to account takeover. Secure implementations must use time-limited, cryptographically secure reset tokens and require reauthentication for critical changes.

VI. Conclusion

This lab provided hands-on experience in identifying, exploiting, and understanding authentication vulnerabilities. Through both legacy and modern WebGoat exercises, we learned how weak password practices, misconfigured login systems, and insecure JWT implementations can compromise entire applications. These vulnerabilities highlight the importance of robust input validation, secure session handling, and defense-in-depth authentication mechanisms in real-world systems.

References

[1] OWASP Foundation. OWASP Top 10: 2021 – A07: Identification and Authentication Failures.
[2] How Secure Is My Password? – Password strength estimation tool.
[3] WebGoat Project, OWASP. WebGoat 7.1 and WebGoat v2023.8 Documentation.