Lesson 4 – Insecure Authentication

Introduction

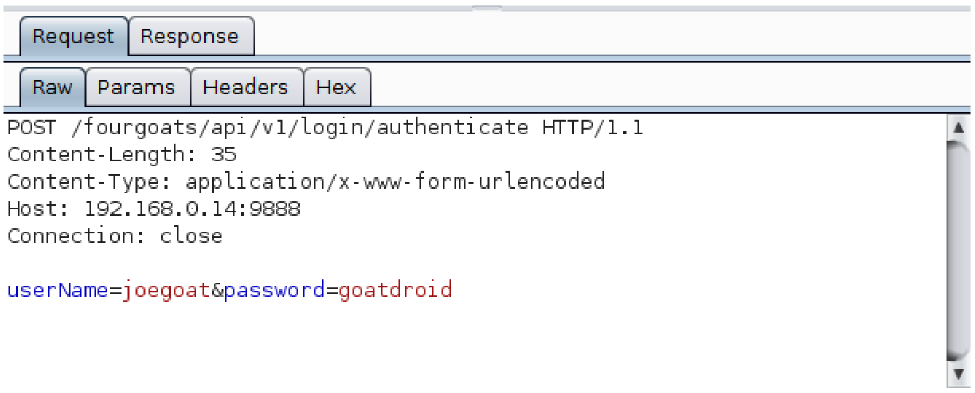
This lesson focuses on finding and exploiting security issues in Android applications that are caused by weaknesses in authenticating the user and managing the session.

Getting started

For this lesson we will use the intercepting proxy, so the set set up is the same as in lesson 3.

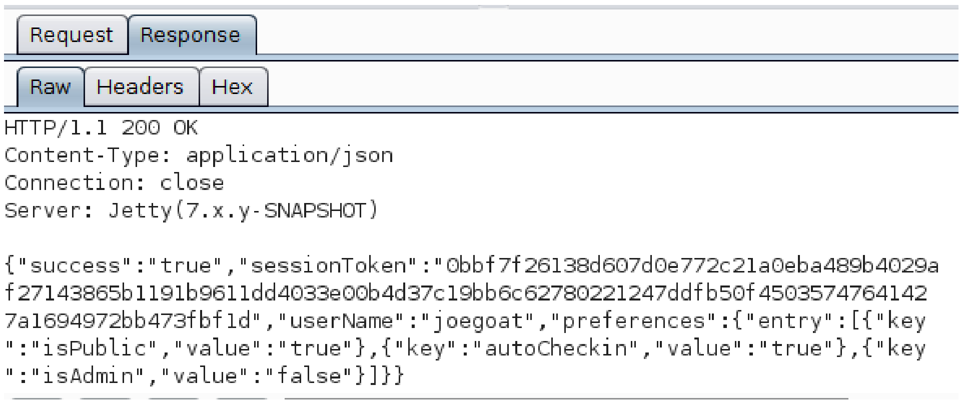
Finding vulnerabilities

Start with intercepting the application traffic and look how login request and response look like:



```
POST /fourgoats/api/v1/login/authenticate HTTP/1.1
Content-Length: 35
Content-Type: application/x-www-form-urlencoded
Host: 192.168.0.14:9888
Connection: close

userName=joegoat&password=goatdroid
```



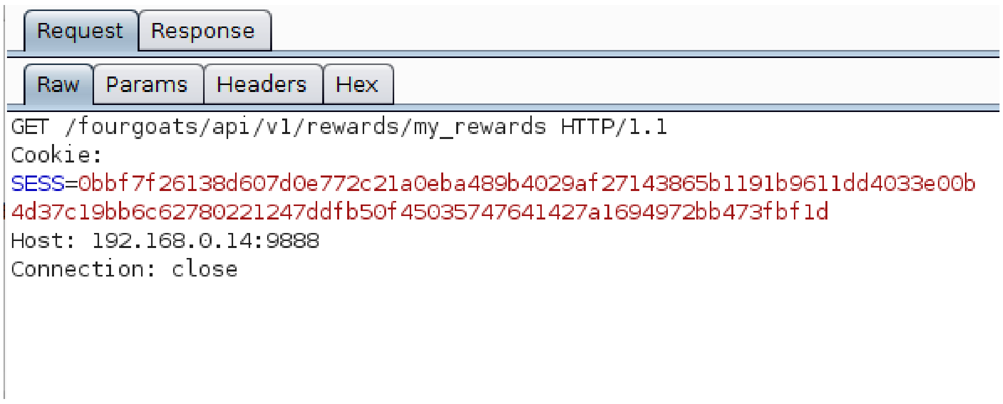
```
HTTP/1.1 200 OK
Content-Type: application/json
Connection: close
Server: Jetty(7.x.y-SNAPSHOT)

{"success":"true","sessionToken":"0bbf7f26138d607d0e772c21a0eba489b4029a
f27143865b1191b9611dd4033e00b4d37c19bb6c62780221247ddfb50f4503574764142
7a1694972bb473fbf1d","userName":"joegoat","preferences":{"entry":[{"key
":"isPublic","value":"true"},{"key":"autoCheckin","value":"true"},{"key
":"isAdmin","value":"false"}]}}
```

The password is sent in plaintext. It would not be so dangerous, since the communication is protected with SSL/TLS. But as it was proven in lesson 3. there is a possibility of traffic interception. Even with properly implemented SSL/TLS without Certificate Pinning a malicious third party can install its certificate on the device and inspect the traffic.

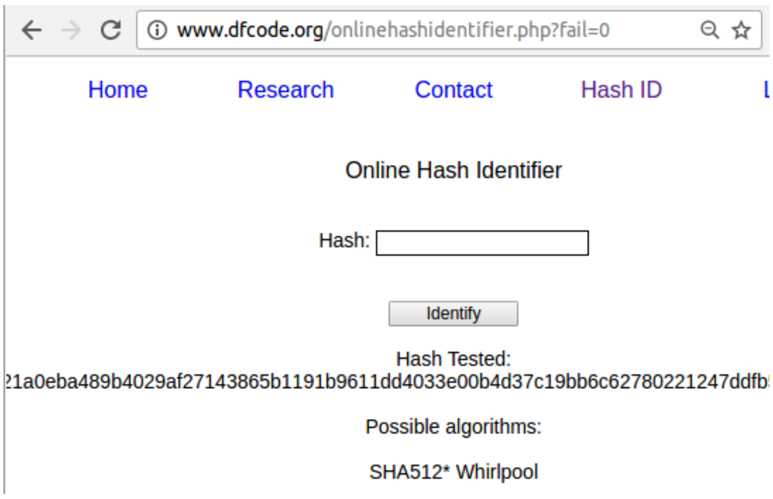
Sending password in plaintext bears the risk of an attacker capturing it and trying to use it in different services, since most users use the same password in many different places. Also an attacker could have access to the traffic once, capture the hash and use it later to log in to the service many times.

In the response to the authentication request there was a session token sent. It is appended as a cookie to next requests:



```
GET /fourgoats/api/v1/rewards/my_rewards HTTP/1.1
Cookie:
SESS=0bbf7f26138d607d0e772c21a0eba489b4029af27143865b1191b9611dd4033e00b
4d37c19bb6c62780221247ddfb50f45035747641427a1694972bb473fbf1d
Host: 192.168.0.14:9888
Connection: close
```

The requirements for session token include it being long enough not to contain any additional information and to have a good entropy to make it difficult for an attacker to guess one. At first glance the token looks pretty good, it is long and does not look like it would carry any additional information. You can suspect it is a hash, since it looks like one and hashes are often used as session tokens. To confirm these suspections you can use hash identifying software. One of the tools available online, www.dfcode.org/onlinehashidentifier.php suggests that it might be SHA512.



Online Hash Identifier

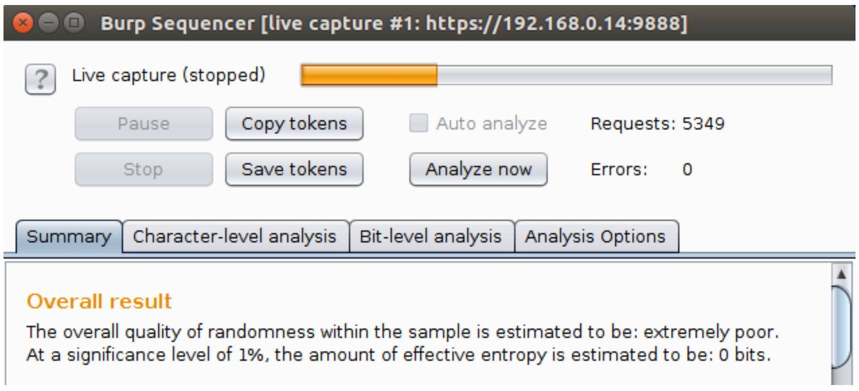
Hash:

Hash Tested:
21a0eba489b4029af27143865b1191b9611dd4033e00b4d37c19bb6c62780221247ddfb

Possible algorithms:

SHA512* Whirlpool

To check the entropy of the session token we will use Sequencer, a tool that is a part of Burp Suite Free Edition. Select the authentication request, right click on it and select Send to Sequencer. Sequencer is a tool that repeats the authentication packet many times and collects different session tokens extracted from the response in order to calculate their entropy. Under the Token Location Within Response section of Sequencer pick Configure and select the session token string. Click Start Live Capture. Wait for at least 5000 tokens to get captured for the results of analysis to be trustworthy.

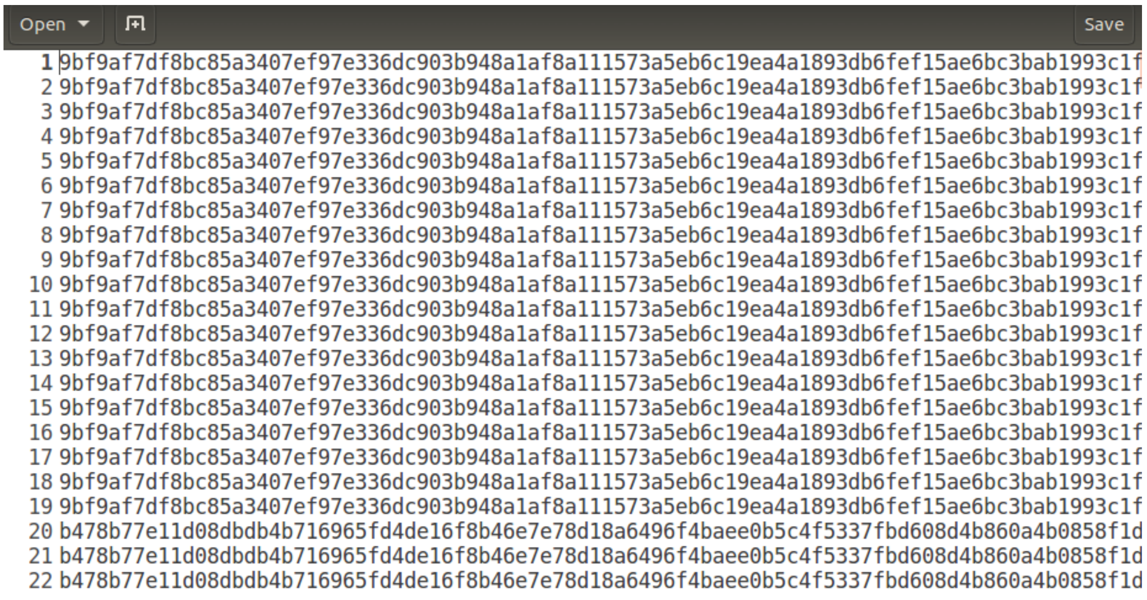


Requests: 5349
Errors: 0

Overall result

The overall quality of randomness within the sample is estimated to be: extremely poor.
At a significance level of 1%, the amount of effective entropy is estimated to be: 0 bits.

The Sequencer showed that the entropy is 0 bits. No entropy at all. What happened? Save tokens to the text file to look at them.

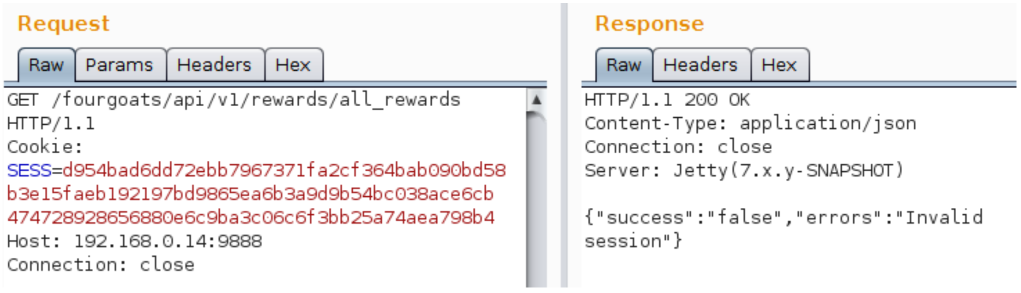


```
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
9bf9af7df8bc85a3407ef97e336dc903b948a1af8a111573a5eb6c19ea4a1893db6fef15ae6bc3bab1993c1f
b478b77e11d08dbdb4b716965fd4de16f8b46e7e78d18a6496f4baee0b5c4f5337fbd608d4b860a4b0858f1d
b478b77e11d08dbdb4b716965fd4de16f8b46e7e78d18a6496f4baee0b5c4f5337fbd608d4b860a4b0858f1d
b478b77e11d08dbdb4b716965fd4de16f8b46e7e78d18a6496f4baee0b5c4f5337fbd608d4b860a4b0858f1d
```

It looks like tokens are generated based on time and do not change so often.
Let’s look at the server implementation of token generation to confirm our suspicions:

```
String userNameAndTime = userName + LoginUtils.getCurrentDateTime();
long sessionStartTime = LoginUtils.getTimeMilliseconds();
String sessionToken = LoginUtils.generateSaltedSHA512Hash(userNameAndTime,
                        Salts.SESSION_TOKEN_SALT);
dao.updateSessionInformation(userName, sessionToken,sessionStartTime);
```

The session token is generated by hashing a concatenated username and current time (with accuracy to 1 second) and generating a SHA512 hash from it with constant salt. This is not a good idea. Attacker can predict the time of user’s login in order to steal his session. The session ID should be time independent and difficult to predict for the attacker.
Now let’s check if the session invalidation is implemented correctly. Log out of the application and send one of the packets capture before to the Repeater (right mouse click) in order to check if they would still get a response:



```
GET /fourgoats/api/v1/rewards/all_rewards
HTTP/1.1
Cookie:
SESS=d954bad6dd72ebb7967371fa2cf364bab090bd58
b3e15faeb192197bd9865ea6b3a9d9b54bc038ace6cb
474728928656880e6c9ba3c06c6f3bb25a74aea798b4
Host: 192.168.0.14:9888
Connection: close
```

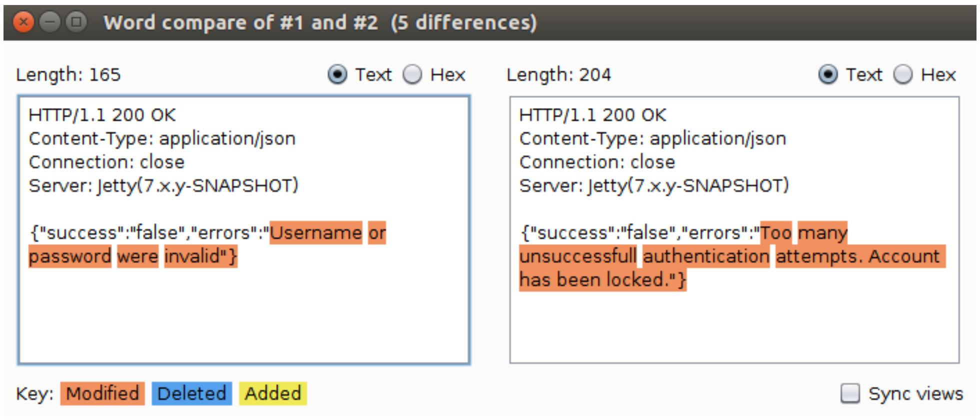
```
HTTP/1.1 200 OK
Content-Type: application/json
Connection: close
Server: Jetty(7.x.y-SNAPSHOT)

{"success":"false","errors":"Invalid
session"}
```

This worked correctly and did not allow to use the session token after its invalidation.

The next part of authentication process we are going to cover is user enumeration and account password brute force prevention mechanism. Finding a way to recognize which user accounts exist can help the attacker to pick existing accounts and target them in a brute force or dictionary attack to guess the correct credentials.

To check the user enumeration compare the response packets from requests with nonexistent account and an existing account with incorrect password. The differences sometimes can be difficult to find out manually, so you can send both packets to Burp Comparer to do the task.



Word compare of #1 and #2 (5 differences)

Length: 165 ● Text ○ Hex

```
HTTP/1.1 200 OK
Content-Type: application/json
Connection: close
Server: Jetty(7.x.y-SNAPSHOT)

{"success":"false","errors":"Username or
password were invalid"}
```

Length: 204 ● Text ○ Hex

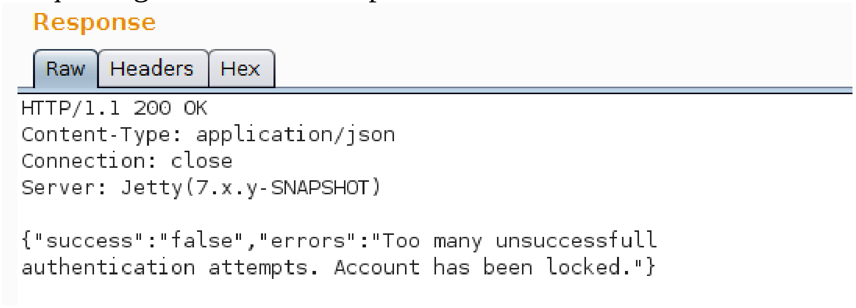
```
HTTP/1.1 200 OK
Content-Type: application/json
Connection: close
Server: Jetty(7.x.y-SNAPSHOT)

{"success":"false","errors":"Too many
unsuccessfull authentication attempts. Account
has been locked."}
```

Key: Modified Deleted Added ☐ Sync views

On the left you can see the server response to an existing account, while on the right there is a response to a nonexistent account.

Now, send the login packet to the Repeater and repeat it to see if the account will get locked. After 10 times, the next requests get the lockout response:



```
HTTP/1.1 200 OK
Content-Type: application/json
Connection: close
Server: Jetty(7.x.y-SNAPSHOT)

{"success":"false","errors":"Too many unsuccessfull
authentication attempts. Account has been locked."}
```

By analysis the server source code you can find out that the account gets unlocked after 24 hours from the last unsuccessful login attempt. It opens up a threat of Denial of Service attack – an attacker could first enumerate existing user accounts and then send requests with incorrect password to block the accounts, resulting in blocking access to the server for all the users.

Remember to check these functionalities thoroughly, often the user enumeration vulnerabilities exists because of incorrect architecture of account lockout functionality. For example existing accounts get locked after sending certain number of requests with incorrect password, while nonexistent accounts do not get locked at all.

The time after the session is automatically invalidated if no user action is performed and a time when the session is unconditionally invalidated should also be checked (If the attacker would steal the session, he could repeat performing some kind of action in order to keep access to the system).

Another important authentication process can be lack of or insufficient password policy that could allow an attacker to easily guess the password. You can check that in Fourgoats application the account password can have only on character.

Fixing the vulnerabilities

A simple way to avoid sending passwords in plaintext for authentication would be to hash the password and send the hash to the server. This resolves this issue, but does not resolve a second one – a possibility to capture the hash and use it for authentication later. An ideal solution would not allow that. This issue can be resolved with the server sending one-time random salt that the client would use to hash the password and send the hash back to server.

For generation of session ID that fulfills all the security requirements it is best to use the session mechanism implemented in the application server. Other simple solution would be to use Java UDID class can be used to generate a random, safe unique string. Another possibility which requires some more effort to be as secure as the previous options is to use SecureRandom Java class to get the random seed and hash it to generate session token.

The response sent by server after authentication attempt with existing account and wrong password should be the same as the response from attempt with nonexistent account to prevent user enumeration possibility.

To make the system resistant from Denial of Service attacks with the lockout functionality and to protect from guessing attack on the passwords a mechanism that would not allow automated guesses should be implemented. A Captcha image difficult to read for OCR software would work in these conditions.

Session invalidation after some time without user interaction and time-based session invalidation regardless user interaction should be implemented.

The password complexity policy should be implemented to enforce proper password length, usage of both alphabet and numeric characters and in some cases also a minimum number of special characters.