The Web Application Hackers Handbook Chapters 6-7

Chapter 6: Attacking Authentication

Not as many notes on the first part of this chaper because its mostly common sense.

- The 2 main things you want to be on the hunt for when trying to break authentication is:
  1. Brute-Force Login ability, meaning no timeout when too many wrong username password combinations are entered.
  2. Verbose Failure Messages, which may say "Password is incorrect" or "User not recognized" (or both) to indicate that you entered in a proper username. This can be used to create a list of confirmed existing usernames.

Vulnerable Transmission of Credentials

- Many applications use HTTP for unauthenticated areas of the application and switch to HTTPS at the point of login.
  - If this is the case, then the correct place to switch to HTTPS is when the login page is loaded in the browser, enabling a user to verify that the page is authentic before entering credentials.
  - However, it is common to encounter applications that load the login page itself using HTTP and then switch to HTTPS at the point where credentials are submitted. **This is unsafe, because a user cannot verify the authenticity of the login page itself and therefore has no assurance that the credentials will be submitted securely.**
    - A suitably positioned attacker can intercept and modify the login page, chang- ing the target URL of the login form to use HTTP.

Password Change Functionality

- Although it is a necessary part of an effective authentication mechanism, password change functionality is often vulnerable by design. Vulnerabilities that are deliberately avoided in the main login function often reappear in the password change function.
- Many web applications' password change functions are accessible without authentication and do the following:
  - Provide a verbose error message indicating whether the requested user- name is valid.
  - Allow unrestricted guesses of the "existing password" field.
  - Check whether the "new password" and "confirm new password" fields have the same value only after validating the existing password, thereby allowing an attack to succeed in discovering the existing password noninvasively.

Password Reset Functionality

- A lot of times, password reset is insecure. If there is a security question that must be guessed when resetting a password this can often times be bruteforced without a timeout and generally has less possible combinations than the password itself.
- Additionally, a password reset that doesn't email a user with a time sensitive reset link is generally flawed.

Remember-Me Functionality

- Some “remember me” functions set a cookie that contains not the username but a kind of persistent session identifier, such as `RememberUser=1328`, or a username, such as `RememberUser=TonyG`. When the identifier is submitted to the login page, the application looks up the user associated with the id or username and creates an application session for that user.

Chapter 7: Attacking Session Management

- The HTTP protocol is essentially stateless. It is based on a simple request-response model, in which each pair of messages represents an independent transaction.
    - The protocol itself contains no mechanism for linking the series of requests made by a particular user and distinguishing these from all the other requests received by the web server.
    - Web applications use **Sessions** to remember your preferences and who is intereacting with the site, until you log out or the session expires.
        - Without a session, a user would have to reenter his password on every page of the application.
        - The simplest and still most common means of implementing sessions is to issue each user a unique session token or identifier.
            - On each subsequent request to the application, the user resubmits this token, enabling the application to determine which sequence of earlier requests the current request relates to.
- The vulnerabilities that exist in session management mechanisms largely fall into two categories:
    1. Weaknesses in the generation of session tokens
    2. Weaknesses in the handling of session tokens throughout their life cycle

---

- **HACK STEPS: Identifyng Session Tokens**

1. The application may often employ several different items of data collectively as a token, including cookies, URL parameters, and hidden form fields. Some of these items may be used to maintain session state on different back-end components. Do not assume that a particular parameter is the session token without proving it, or that sessions are being tracked using only one item.
2. Sometimes, items that appear to be the application’s session token may not be. In particular, the standard session cookie generated by the web server or application platform may be present but not actually used by the application.

3. Observe which new items are passed to the browser after authentication. Often, new session tokens are created after a user authenticates herself.
4. To verify which items are actually being employed as tokens, find a page that is definitely session-dependent (such as a user-specific "my details" page). Make several requests for it, systematically removing each item that you suspect is being used as a token. If removing an item causes the session-dependent page not to be returned, this may confirm that the item is a session token. Burp Repeater is a useful tool for performing these tests.

Alternatives to Sessions

1. HTTP Authentication
    - Applications using the various HTTP-based authentication technologies (basic, digest, NTLM) sometimes avoid the need to use sessions.
    - With HTTP authentication, the client component interacts with the authentication mechanism directly via the browser, using HTTP headers, and not via application-specific code contained within any individual page.
    - After the user enters his credentials into a browser dialog, the browser effectively resubmits these credentials (or reperforms any required handshake) with every subsequent request to the same server.
        - This is equivalent to an application that uses HTML forms-based authentication and places a login form on every application page, requiring users to reauthenticate themselves with every action they perform.
2. Sessionless State Mechanisms
    - Involves transmitting all data required to manage that state via the client, usually in a cookie or a hidden form field.
    - In effect, this mechanism uses sessionless state much like the ASP.NET ViewState does.
    - For this type of mechanism to be secure, the data transmitted via the client must be properly protected.
        - This usually involves constructing a binary blob containing all the state information and encrypting or signing this using a recognized algorithm.
        - Sufficient context must be included within the data to prevent an attacker from collecting a state object at one location within the application and submitting it to another location to cause some undesirable behavior.
        - The application may also include an expiration time within the object's data to perform the equivalent of session timeouts.

- There are numerous locations where an application's security depends on the unpredictability of tokens it generates. Here are some examples:
    - Password recovery tokens sent to the user's registered e-mail address

- Tokens placed in hidden form fields to prevent cross-site request forgery attacks (see Chapter 13)
- Tokens used to give one-time access to protected resources
- Persistent tokens used in "remember me" functions
- Tokens allowing customers of a shopping application that does not use authentication to retrieve the current status of an existing order

Predictable Tokens

Concealed Sequences

- It is common to encounter session tokens that cannot be easily predicted when analyzed in their raw form but that contain sequences that reveal themselves when the tokens are suitably decoded or unpacked.
- Consider the following series of values, which form one component of a structured session token:

```
lwjVJA
Ls3Ajg
xpKr+A
XleXYg
9hyCzA
jeFuNg
JaZZoA
```

  - No immediate pattern is discernible; however, a cursory inspection indicates that the tokens may contain Base64-encoded data.
  - Running the tokens through a Base64 decoder reveals the following:

```
--Õ$
.ÍÀŽ
Æ'«ø
^W-b
ö,Ì
?án6
%¦Y
```

  - These strings appear to be gibberish and also contain nonprinting characters.
    - This normally indicates that you are dealing with binary data rather than ASCII text.
  - Rendering the decoded data as hexadecimal numbers gives you the following:

```
9708D524
2ECDC08E
C692ABF8
```

```
5E579762
F61C82CC
8DE16E36
25A659A0
```

- There appears to be no visible pattern... However, if you subtract each number from the previous one you arrive at the following, which reveals the concealed patter:

```
FF97C4EB6A
97C4EB6A
FF97C4EB6A
97C4EB6A
FF97C4EB6A
FF97C4EB6A
```

- The algorithm used to generate tokens adds `0x97C4EB6A` to the previous value, truncates the result to a 32-bit number, and Base64-encodes this binary data to allow it to be transported using the text-based protocol HTTP.
- *Using this knowledge, you can easily write a script to produce the series of tokens that the server will next produce, and the series that it produced prior to the captured sample.*

Time Dependency

- Some web servers and applications employ algorithms to generate session tokens that use the time of generation as an input to the token's value.
  - If insufficient other entropy is incorporated into the algorithm, you may be able to predict other users' tokens.
- Consider the following example:

```
3124538-1172764258718
3124539-1172764259062
3124540-1172764259281
3124541-1172764259734
3124542-1172764260046
3124543-1172764260156
3124544-1172764260296
3124545-1172764260421
3124546-1172764260812
3124547-1172764260890
```

  - Each token is clearly composed of two separate numeric components.
    - The first number follows a simple incrementing sequence and is easy to predict.
    - The second number increases by a varying amount each time.

- Calculating the differences between its value in each successive token reveals the following:

```
344
219
453
312
110
140
125
391
78
```

- The sequence does not appear to contain a reliably predictable pattern. However, it would clearly be possible to brute-force the relevant number range in an automated attack to discover valid values in the sequence.
- Before attempting this attack, however, we wait a few minutes and gather a further sequence of tokens:

```
3124553-1172764800468
3124554-1172764800609
3124555-1172764801109
3124556-1172764801406
3124557-1172764801703
3124558-1172764802125
3124559-1172764802500
3124560-1172764802656
3124561-1172764803125
3124562-1172764803562
```

- Comparing this second sequence of tokens with the first, two points are imme- diately obvious:
  1. The first numeric sequence continues to progress incrementally; however, five values have been skipped since the end of the first sequence. This is presumably because the missing values have been issued to other users who logged in to the application in the window between the two tests.
  2. The second numeric sequence continues to progress by similar intervals as before; however, the first value we obtain is a massive 539,578 greater than the previous value (3124547-1172764260890).
     - This observation alerts us that time is a factor in session token generation.
     - The second number is clearly time dependent and is probably a simple count of milliseconds.
- It becomes apparent that the token generation algorithm employed is something like this:

```
String sessId = Integer.toString(s_SessionIndex++) + "-" +
System.currentTimeMillis();
```

- After our analysis of how tokens are created in the example, we can devise a plan to create a scripted attack to harvest the session tokens that the application issues to other users:
  - We continue polling the server to obtain new session tokens in quick succession.
  - We monitor the increments in the first number. When this increases by more than 1, we know that a token has been issued to another user.
  - **When a token has been issued to another user, we know the upper and lower bounds of the second number that was issued to that person, because we possess the tokens that were issued immediately before and after his.**
    - Because we are obtaining new session tokens frequently, the range between these bounds will typically consist of only a few hundred values.
  - Each time a token is issued to another user, we launch a brute-force attack to iterate through each number in the range, appending this to the missing incremental number that we know was issued to the other user.
    - We attempt to access a protected page using each token we construct, until the attempt succeeds and we have compromised the user's session
  - Running this scripted attack continuously will enable us to capture the session token of every other application user. When an administrative user logs in, we will fully compromise the entire application.

Weak Random Number Generation

- Very little that occurs inside a computer is random.
- In general, if an algorithm is not explicitly described as being cryptographically secure, it should be assumed to be predictable.
- A highly effective formula for incorporating this entropy is to construct a string that concatenates a pseudorandom number, a variety of request-specific data as listed, and a secret string known only to the server and generated afresh on each reboot.
  - A suitable hash is then taken of this string (using, for example, SHA-256) to produce a manageable fixed-length string that can be used as a token
- Some of the algorithms used produce sequences that appear to be stochastic and manifest an even spread across the range of possible values, meaning, they can be extrapolated forwards or backwards with perfect accuracy by anyone who obtains a small sample of values.
- For Example: Jetty, a poplular 100% Java webserver, was discovered to use the Java API's `java.util.Random` to generate session tokens....
- `java.util.Random` is a "linear congruential generator" which generates the next number in a sequence as follows:

```
synchronized protected int next(int bits) {
   seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
   return (int)(seed >>> (48 - bits));
}
```

    - This algorithm takes the last number generated, multiplies it by a constant, and adds another constant to obtain the next number. The number is truncated to 48 bits, and the algorithm shifts the result to return the specific number of bits requested by the caller.

- The best tool that is currently available for testing the randomness of web application tokens is Burp Sequencer
    - To use Burp Sequencer, you need to find a response from the application that issues the token you want to test, such as a response to a login request that issues a new cookie containing a session token. Select the "send to sequencer" option from Burp's context menu, and in the Sequencer configuration, set the location of the token within the response
    - You must have a minimum of 100 tokens, or a maximum of 20,000 tokens. The more tokens the better the results.
        - To achieve compliance with the formal FIPS tests for randomness, you need to obtain a sample of 20,000 tokens.
    - **Note:** Just because tokens pass the test for randomness does not necessarily mean that an application is or is not exploitable. 2 caveats to consider:
        1. Tokens that are generated in a completely deterministic way may pass the statistical tests for randomness. Yet an attacker who knows the algorithm and the internal state of the generator can extrapolate its output with complete reliability in both forward and reverse directions.
        2. Tokens that fail the statistical tests for randomness may not actually be predictable in any practical situation. If a given bit of a token fails the tests, this means only that the sequence of bits observed at that position contains characteristics that are unlikely to occur in a genuinely random token.

- **HACK STEPS: Harvesting Tokens**

1. Determine when and how session tokens are issued by walking through the application from the first application page through any login functions. Two behaviors are common:
    - The application creates a new session anytime a request is received that does not submit a token.
    - The application creates a new session following a successful login.

To harvest large numbers of tokens in an automated way, ideally identify a single request (typically either `GET /` or a login submission) that causes a new token to be issued.

2. In Burp Suite, send the request that creates a new session to Burp Sequencer, and configure the token's location. Then start a live capture to gather as many tokens as is feasible. If a custom session management mechanism is in use, and you only have remote access to the application, gather the tokens as quickly as possible to minimize the loss of tokens issued to other users and reduce the influence of any time dependency.
3. If a commercial session management mechanism is in use and/or you have local access to the application, you can obtain indefinitely large sequences of session tokens in controlled conditions.
4. While Burp Sequencer is capturing tokens, enable the "auto analyse" setting so that Burp automatically performs the statistical analysis periodically. Collect at least 500 tokens before reviewing the results in any detail. If a sufficient number of bits within the token have passed the tests, continue gathering tokens for as long as is feasible, reviewing the analysis results as further tokens are captured.
5. If the tokens fail the randomness tests and appear to contain patterns that could be exploited to predict future tokens, reperform the exercise from a different IP address and (if relevant) a different username. This will help you identify whether the same pattern is detected and whether tokens received in the first exercise could be extrapolated to identify tokens received in the second. Sometimes the sequence of tokens captured by one user manifests a pattern. But this will not allow straightforward extrapolation to the tokens issued to other users, because information such as source IP is used as a source of entropy (such as a seed to a random number generator).
6. If you believe you have enough insight into the token generation algorithm to mount an automated attack against other users' sessions, it is likely that the best means of achieving this is via a customized script. This can generate tokens using the specific patterns you have observed and apply any necessary encoding. See Chapter 14 for some generic techniques for applying automation to this type of problem.
7. If source code is available, closely review the code responsible for generating session tokens to understand the mechanism used and determine whether it is vulnerable to prediction. If entropy is drawn from data that can be determined within the application within a brute-forcible range, consider the practical number of requests that would be needed to brute force an application token.

Encrypted Tokens

- In some situations, depending on the encryption algorithm used and the manner in which the application processes the tokens, it may be possible for users to tamper with the tokens' meaningful contents without actually decrypting them.

ECB Ciphers

- Electronic Codebook (ECB) is a symmetric cipher.
    - This type of cipher divides plaintext into equal-sized blocks (such as 8 bytes each) and encrypts each block using the secret key.

    - During decryption, each block of ciphertext is decrypted using the same key to recover the original block of plaintext.
- One feature of this method is that **patterns within the plaintext can result in patterns within the ciphertext**, because identical blocks of plaintext will be encrypted into identical blocks of ciphertext.
- Consider an application whose tokens contain several different meaningful components, including a numeric user identifier:

```
rnd=2458992;app=iTradeEUR_1;uid=218;username=dafydd;time=634430423694715
000;
```

- When this token is encrypted, it is apparently meaningless and is likely to pass all standard statistical tests for randomness:

```
68BAC980742B9EF80A27CBBBC0618E3876FF3D6C6E6A7B9CB8FCA486F9E11922776F0307
329140AABD223F003A8309DDB6B970C47BA2E249A0670592D74BCD07D51A3E150EFC2E69
885A5C8131E4210F
```

- The ECB cipher being employed operates on 8-byte blocks of data, and the blocks of plaintext map to the corresponding blocks of ciphertext as follows:

```
rnd=2458        68BAC980742B9EF8
992;app=        0A27CBBBC0618E38
iTradeEU        76FF3D6C6E6A7B9C
R_1;uid=        B8FCA486F9E11922
218;user        776F0307329140AA
name=daf        BD223F003A8309DD
ydd;time        B6B970C47BA2E249
=6344304        A0670592D74BCD07
23694715        D51A3E150EFC2E69
000;            885A5C8131E4210F
```

- Because each block of ciphertext will always decrypt into the same block of plaintext, it is possible for an attacker to manipulate the sequence of ciphertext blocks so as to modify the corresponding plaintext in meaningful ways.
- For example, if the second block is duplicated following the fourth block, the sequence of blocks will be as follows:

```
rnd=2458        68BAC980742B9EF8
992;app=        0A27CBBBC0618E38
iTradeEU        76FF3D6C6E6A7B9C
R_1;uid=        B8FCA486F9E11922
```

```
992;app=        0A27CBBBC0618E38

218;user        776F0307329140AA
name=daf        BD223F003A8309DD
ydd;time        B6B970C47BA2E249
=6344304        A0670592D74BCD07
23694715        D51A3E150EFC2E69
000;            885A5C8131E4210F
```

  - *The line above surrounded by blank lines is the modified line.*
  - The decrypted token now contains a modified `uid` value, and also a duplicated `app` value.
    - Exactly what happens depends on how the application processes the decrypted token.
    - Often, applications using tokens in this way inspect only certain parts of the decrypted token, such as the user identifier.
    - If the application behaves like this, then it will process the request in the context of the user who has a uid of 992, rather than the original 218.
  - The attack just described would depend on being issued with a suitable rnd value that corresponds to a valid uid value when the blocks are manipulated.

- An alternative and more reliable attack would be to register a username containing a numeric value at the appropriate offset, and duplicate this block so as to replace the existing `uid` value.
- Suppose you register the username `daf1`, and are issued with the following token:

```
9A5A47BF9B3B6603708F9DEAD67C7F4C76FF3D6C6E6A7B9CB8FCA486F9E11922A5BC430A
73B38C14BD223F003A8309DDF29A5A6F0DC06C53905B5366F5F4684C0D2BBBB08BD834BB
ADEBC07FFE87819D
```

- The blocks of plaintext and ciphertext for this token are as follows:

```
rnd=9224        9A5A47BF9B3B6603
856;app=        708F9DEAD67C7F4C
iTradeEU        76FF3D6C6E6A7B9C
R_1;uid=        B8FCA486F9E11922
219;user        A5BC430A73B38C14
name=daf        BD223F003A8309DD
1;time=6        F29A5A6F0DC06C53
34430503        905B5366F5F4684C
61065250        0D2BBBB08BD834BB
0;              ADEBC07FFE87819D
```

- If you then duplicate the seventh block following the fourth block, your decrypted token will contain a `uid` value of 1:

```
rnd=9224        9A5A47BF9B3B6603
856;app=        708F9DEAD67C7F4C
iTradeEU        76FF3D6C6E6A7B9C
R_1;uid=        B8FCA486F9E11922

1;time=6        F29A5A6F0DC06C53

name=daf        BD223F003A8309DD
1;time=6        F29A5A6F0DC06C53
34430503        905B5366F5F4684C
61065250        0D2BBBB08BD834BB
0;              ADEBC07FFE87819D
```

- By registering a suitable range of usernames and reperforming this attack, you could potentially cycle through the entire range of valid `uid` values, and so masquerade as every user of the application.

CBC Ciphers

- Cipher Block Chaining (CBC) Cipher is an improvement upon ECB.
  - With CBC cipher, before each block of plaintext is encrypted it is XORed against the preceding block of ciphertext.
    - This prevents identical plaintext blocks from being encrypted into identical ciphertext blocks.
  - During decryption, the XOR operation is applied in reverse, and each decrypted block is XORed against the preceding block of ciphertext to recover the original plaintext.

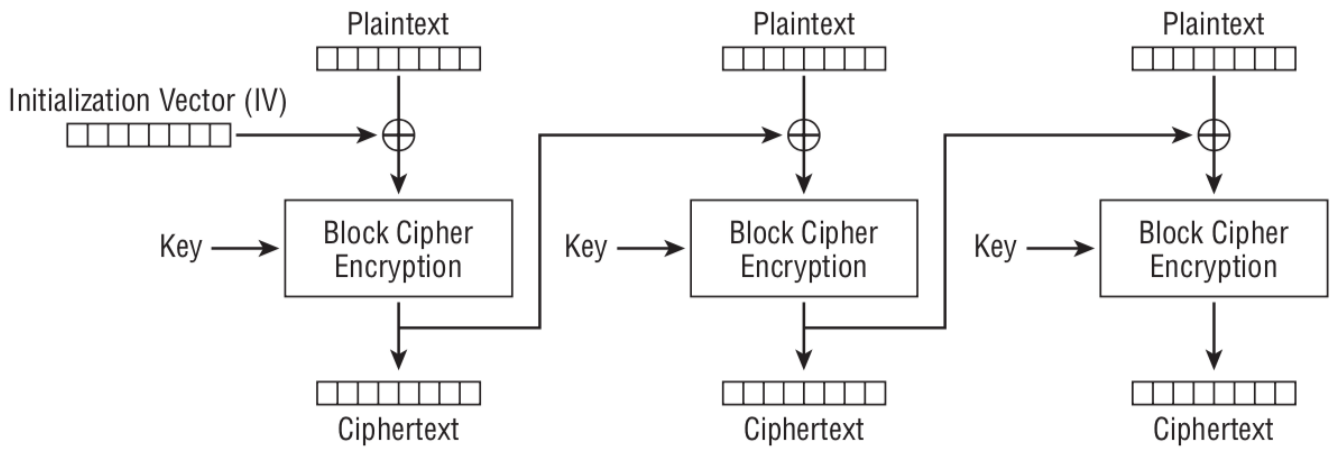

Figure 7-5: In a CBC cipher, each block of plaintext is XORed against the preceding block of ciphertext before being encrypted.

- Consider a variation on the preceding application whose tokens contain several different meaningful components, including a numeric user identifier:

```
rnd=191432758301;app=eBankProdTC;uid=216;time=6343303;
```

- As before, when this information is encrypted, it results in an apparently meaningless token:

```
0FB1F1AFB4C874E695AAFC9AA4C2269D3E8E66BBA9B2829B173F255D447C51321586257C
6E459A93635636F45D7B1A43163201477
```

- Because this token is encrypted using a CBC cipher, when the token is decrypted, each block of ciphertext is XORed against the following block of decrypted text to obtain the plaintext.
  - Now, if an attacker modifies parts of the ciphertext (the token he received), this causes that specific block to decrypt into junk.
  - However, it also causes the following block of decrypted text to be XORed against a different value, resulting in modified but still meaningful plaintext.
    - In other words, by manipulating a single individual block of the token, the attacker can systematically modify the decrypted contents of the block that follows it.
    - Depending on how the application processes the resulting decrypted token, this may enable the attacker to switch to a different user or escalate privileges.
- In the example described, the attacker works through the encrypted token, changing one character at a time in arbitrary ways and sending each modified token to the application. This involves a large number of requests.
- The following is a selection of the values that result when the application decrypts each modified token:

```
????????32 8 58301;app=eBankProdTC;uid=216;time=6343303;
????????327583 2 1;app=eBankProdTC;uid=216;time=6343303;
rnd=1914????????;a q p=eBankProdTC;uid=216;time=6343303;
rnd=1914????????;app=e A ankProdTC;uid=216;time=6343303;
rnd=191432758301????????nkP q odTC;uid=216;time=6343303;
rnd=191432758301????????nkProd U C;uid=216;time=6343303;
rnd=191432758301;app=eBa????????;ui e =216;time=6343303;
rnd=191432758301;app=eBa????????;uid=2 2 6;time=6343303;
rnd=191432758301;app=eBankProdTC????????;tim d =6343303;
rnd=191432758301;app=eBankProdTC????????;time=6343 5 03;
```

  - *The separated character in each line is the modified character.*
  - In each case, the block that the attacker has modified decrypts into junk, as expected (indicated by `????????`).
  - However, the following block decrypts into meaningful text that differs slightly from the original token.

    - Although the attacker does not see the decrypted values, the application attempts to process them, and the attacker sees the results in the application's responses.
        - Exactly what happens depends on how the application handles the part of the decrypted token that has been corrupted.
    - Often, however, applica- tions using tokens in this way inspect only certain parts of the decrypted token, such as the user identifier.
        - If the application behaves like this, then the eighth example shown in the preceding list succeeds, and the application processes the request in the context of the user who has a `uid` of 226, rather than the original 216.
- You can easily test applications for this vulnerability using the "bit flipper" payload type in Burp Intruder (more info page 228).

Weaknesses in Session Token Handling

Disclosure of Tokens on the Network

- Suitable positions for eavesdropping include:
    - the user's local network,
    - within the user's IT department,
    - within the user's ISP,
    - on the Internet backbone,
    - within the application's ISP, and
    - within the IT department of the organization hosting the application.
- In the simplest case, where an application uses an unencrypted HTTP connection for communications, an attacker can capture all data transmitted between client and server, including login credentials, personal information, payment details, and so on.
- In other cases, an application may use HTTPS to protect key client-server communications yet may still be vulnerable to interception of session tokens on the network.
    - This weakness may occur in various ways, many of which can arise specifically when HTTP cookies are used as the transmission mechanism for session tokens.

Disclosure of Tokens in Logs

- Another common place where tokens are simply disclosed to unauthorized view is in system logs of various kinds.
- Many applications provide functionality for administrators and other support personnel to monitor and control aspects of the application's runtime state, including user sessions.
    - For example, a helpdesk worker assisting a user who is having problems may ask for her username, locate her current session through a list or search function, and view relevant details about the session.

- The other main cause of session tokens appearing in system logs is where an application uses the URL query string as a mechanism for transmitting tokens, as opposed to using HTTP cookies or the body of POST requests.
    - For example, Googling `inurl:jsessionid` identifies thousands of applications that transmit the Java platform session token (called jsessionid) within the URL:

```
http://www.webjunction.org/do/Navigation;jsessionid=F27ED2A6AAE4C6DA409A3044E79B8B48?category=327
```

    - When applications transmit their session tokens in this way, it is likely that their session tokens will appear in various system logs to which unauthorized parties may have access:
        - Users' browser logs
        - Web server logs
        - Logs of corporate or ISP proxy servers
        - Logs of any reverse proxies employed within the application's hosting environment
        - The Referer logs of any servers that application users visit by following off-site links
            - For example, if a web mail application transmits session tokens within the URL, an attacker can send e-mails to users of the application containing a link to a web server he controls. If any user accesses the link (because she clicks it, or because her browser loads images contained within HTML-formatted e-mail), the attacker receives, in real time, the user's session token.
            - The attacker can run a simple script on his server to hijack the session of every token received and perform some malicious action, such as send spam e-mail, harvest personal information, or change passwords.

Vulnerable Mapping of Tokens to Sessions

- Various common vulnerabilities in session management mechanisms arise because of weaknesses in how the application maps the creation and processing of session tokens to individual users' sessions themselves.
- The simplest weakness is to allow multiple valid tokens to be concurrently assigned to the same user account.
    - If a user appears to be using two different sessions simultaneously, this usually indicates that a security compromise has occurred: either the user has disclosed his credentials to another party, or an attacker has obtained his credentials through some other means
        - In both cases, permitting concurrent sessions is undesirable, because it allows users to persist in undesirable practices without inconvenience and because it allows an attacker to use captured credentials without risk of detection.
- Static Tokens present another security risk...

- In applications that use static tokens, each user is assigned a token, and this same token is reissued to the user every time he logs in.
- The application always accepts the token as valid regardless of whether the user has recently logged in and been issued with it.

- **HACK STEPS: Evalutating Token Mappings**

1. Log in to the application twice using the same user account, either from different browser processes or from different computers. Determine whether both sessions remain active concurrently. If so, the application supports concurrent sessions, enabling an attacker who has compromised another user's credentials to make use of these without risk of detection.
2. Log in and log out several times using the same user account, either from different browser processes or from different computers. Determine whether a new session token is issued each time or whether the same token is issued each time you log in. If the latter occurs, the application is not really employing proper sessions.
3. If tokens appear to contain any structure and meaning, attempt to separate out components that may identify the user from those that appear to be inscrutable. Try to modify any user-related components of the token so that they refer to other known users of the application, and verify whether the resulting token is accepted by the application and enables you to masquerade as that user.

Liberal Cookie Scope

- The cookie mechanism allows a server to specify both the domain and the URL path to which each cookie will be resubmitted.
  - To do this, it uses the domain and path attributes that may be included in the `Set-cookie` instruction.

Cookie Domain Restrictions

- When the application residing at `foo.wahh-app.com` sets a cookie, the browser by default resubmits the cookie in all subsequent requests to `foo.wahh-app .com`, and also to any subdomains, such as `admin.foo.wahh-app.com`.
- It does not submit the cookie to any other domains, including the parent domain `wahh-app.com` and any other subdomains of the parent, such as `bar.wahh-app.com`.
- A server can override this default behavior by including a domain attribute in the `Set-cookie` instruction.
- For example, suppose that the application at foo .wahh-app.com returns the following HTTP header:

```
Set-cookie: sessionId=19284710; domain=wahh-app.com;
```

    - The browser then resubmits this cookie to all subdomains of `wahh-app.com`, including `bar.wahh-app.com`.
- See full example on page 245
- The problem arises because user-authored blogs are created as subdomains of the main application that handles authentication and session management. There is no facility within HTTP cookies for the application to prevent cookies issued by the main domain from being resubmitted to its subdomains.
    - The solution is to use a different domain name for the main application (for example, `www.wahh-blogs.com`) and to scope the domain of its session token cookies to this fully qualified name. The session cookie will not then be submitted when a logged-in user browses the blogs of other users.

Cookie Path Restrictions

- When the application residing at `/apps/secure/foo-app/index.jsp` sets a cookie, the browser by default resubmits the cookie in all subsequent requests to the path `/apps/secure/foo-app/` and also to any subdirectories.
    - It does not submit the cookie to the parent directory or to any other directory paths that exist on the server.
- A server can override this default behavior by including a path attribute in the Set-cookie instruction.
    - For example, if the application returns the following HTTP header:

```
Set-cookie: sessionId=187ab023e09c00a881a; path=/apps/;
```

        - The browser resubmits this cookie to all subdirectories of the `/apps/` path.
- **Note:** If you don't feel like reading all these notes on chapter 7 you can read the summary notes on pages 248-254 and get the basic idea.