

Attacking Session Management

The session management mechanism is a fundamental security component in the majority of web applications. It is what enables the application to uniquely identify a given user across a number of different requests and to handle the data that it accumulates about the state of that user's interaction with the application. Where an application implements login functionality, session management is of particular importance, because it is what enables the application to persist its assurance of any given user's identity beyond the request in which he supplies his credentials.

Because of the key role played by session management mechanisms, they are a prime target for malicious attacks against the application. If an attacker can break an application's session management, she can effectively bypass its authentication controls and masquerade as other application users without knowing their credentials. If an attacker compromises an administrative user in this way, the attacker can own the entire application.

As with authentication mechanisms, a wide variety of defects can commonly be found in session management functions. In the most vulnerable cases, an attacker simply needs to increment the value of a token issued to him by the application to switch his context to that of a different user. In this situation, the application is wide open for anyone to access all areas. At the other end of the spectrum, an attacker may have to work extremely hard, deciphering several layers of obfuscation and devising a sophisticated automated attack, before finding a chink in the application's armor.

This chapter looks at all the types of weakness the authors have encountered in real-world web applications. It sets out in detail the practical steps you need to take to find and exploit these defects. Finally, it describes the defensive measures that applications should take to protect themselves against these attacks.

COMMON MYTH

“We use smartcards for authentication, and users’ sessions cannot be compromised without them.”

However robust an application’s authentication mechanism, subsequent requests from users are only linked back to that authentication via the resulting session. If the application’s session management is flawed, an attacker can bypass the robust authentication and still compromise users.

The Need for State

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. In the early days of the Web, there was no need for any such mechanism: websites were used to publish static HTML pages for anyone to view. Today, things are very different.

The majority of web “sites” are in fact web applications. They allow you to register and log in. They let you buy and sell goods. They remember your preferences the next time you visit. They deliver rich multimedia experiences with content created dynamically based on what you click and type. To implement any of this functionality, web applications need to use the concept of a *session*.

The most obvious use of sessions is in applications that support logging in. After entering your username and password, you can use the application as the user whose credentials you have entered, until you log out or the session expires due to inactivity. Without a session, a user would have to reenter his password on every page of the application. Hence, after authenticating the user once, the application creates a session for him and treats all requests belonging to that session as coming from that user.

Applications that do not have a login function also typically need to use sessions. Many sites selling merchandise do not require customers to create accounts. However, they allow users to browse the catalog, add items to a shopping basket, provide delivery details, and make a payment. In this scenario, there is no need to authenticate the user’s identity: for the majority of his visit, the application does not know or care who the user is. But to do business with him, it needs to know which series of requests it receives originated from the same user.

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.

In most cases, applications use HTTP cookies as the transmission mechanism for passing these session tokens between server and client. The server's first response to a new client contains an HTTP header like the following:

```
Set-Cookie: ASP.NET_SessionId=mza2ji454s04cwbgbw2ttj55
```

and subsequent requests from the client contain this header:

```
Cookie: ASP.NET_SessionId=mza2ji454s04cwbgbw2ttj55
```

This standard session management mechanism is inherently vulnerable to various categories of attack. An attacker's primary objective in targeting the mechanism is to somehow hijack the session of a legitimate user and thereby masquerade as that person. If the user has been authenticated to the application, the attacker may be able to access private data belonging to the user or carry out unauthorized actions on that person's behalf. If the user is unauthenticated, the attacker may still be able to view sensitive information submitted by the user during her session.

As in the previous example of a Microsoft IIS server running ASP.NET, most commercial web servers and web application platforms implement their own off-the-shelf session management solution based on HTTP cookies. They provide APIs that web application developers can use to integrate their own session-dependent functionality with this solution.

Some off-the-shelf implementations of session management have been found to be vulnerable to various attacks, which results in users' sessions being compromised (these are discussed later in this chapter). In addition, some developers find that they need more fine-grained control over session behavior than is provided for them by the built-in solutions, or they want to avoid some vulnerabilities inherent in cookie-based solutions. For these reasons, it is fairly common to see bespoke and/or non-cookie-based session management mechanisms used in security-critical applications such as online banking.

The vulnerabilities that exist in session management mechanisms largely fall into two categories:

- Weaknesses in the generation of session tokens
- Weaknesses in the handling of session tokens throughout their life cycle

We will look at each of these areas in turn, describing the different types of defects that are commonly found in real-world session management mechanisms, and practical techniques for discovering and exploiting these. Finally, we will describe measures that applications can take to defend themselves against these attacks.

HACK STEPS

In many applications that use the standard cookie mechanism to transmit session tokens, it is straightforward to identify which item of data contains the token. However, in other cases this may require some detective work.

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

Not every web application employs sessions, and some security-critical applications containing authentication mechanisms and complex functionality opt to use other techniques to manage state. You are likely to encounter two possible alternatives:

- **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. Hence, when HTTP-based authentication is used, it is possible

for an application to reidentify the user across multiple requests without using sessions. However, HTTP authentication is rarely used on Internet-based applications of any complexity, and the other versatile benefits that fully fledged session mechanisms offer mean that virtually all web applications do in fact employ these mechanisms.

- **Sessionless state mechanisms** — Some applications do not issue session tokens to manage the state of a user's interaction with the application. Instead, they transmit 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. Chapter 5 describes in more detail secure mechanisms for transmitting data via the client.

HACK STEPS

1. If HTTP authentication is being used, it is possible that no session management mechanism is implemented. Use the methods described previously to examine the role played by any token-like items of data.
2. If the application uses a sessionless state mechanism, transmitting all data required to maintain state via the client, this may sometimes be difficult to detect with certainty, but the following are strong indicators that this kind of mechanism is being used:
 - Token-like data items issued to the client are fairly long (100 or more bytes).
 - The application issues a new token-like item in response to every request.
 - The data in the item appears to be encrypted (and therefore has no discernible structure) or signed (and therefore has a meaningful structure accompanied by a few bytes of meaningless binary data).
 - The application may reject attempts to submit the same item with more than one request.
3. If the evidence suggests strongly that the application is not using session tokens to manage state, it is unlikely that any of the attacks described in this chapter will achieve anything. Your time probably would be better spent looking for other serious issues such as broken access controls or code injection.

Weaknesses in Token Generation

Session management mechanisms are often vulnerable to attack because tokens are generated in an unsafe manner that enables an attacker to identify the values of tokens that have been issued to other users.

NOTE 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

The considerations in this chapter relating to weaknesses in token generation apply to all these cases. In fact, because many of today's applications rely on mature platform mechanisms to generate session tokens, it is often in these other areas of functionality that exploitable weaknesses in token generation are found.

Meaningful Tokens

Some session tokens are created using a transformation of the user's username or e-mail address, or other information associated with that person. This information may be encoded or obfuscated in some way and may be combined with other data.

For example, the following token may initially appear to be a long random string:

```
757365723d6461663b6170703d61646d696e3b646174653d30312f31322f3131
```

However, on closer inspection, you can see that it contains only hexadecimal characters. Guessing that the string may actually be a hex encoding of a string of ASCII characters, you can run it through a decoder to reveal the following:

```
user=daf;app=admin;date=10/09/11
```

Attackers can exploit the meaning within this session token to attempt to guess the current sessions of other application users. Using a list of enumerated or common usernames, they can quickly generate large numbers of potentially valid tokens and test these to confirm which are valid.

Tokens that contain meaningful data often exhibit a structure. In other words, they contain several components, often separated by a delimiter, that can be extracted and analyzed separately to allow an attacker to understand their function and means of generation. Here are some components that may be encountered within structured tokens:

- The account username
- The numeric identifier that the application uses to distinguish between accounts
- The user's first and last names
- The user's e-mail address
- The user's group or role within the application
- A date/time stamp
- An incrementing or predictable number
- The client IP address

Each different component within a structured token, or indeed the entire token, may be encoded in different ways. This can be a deliberate measure to obfuscate their content, or it can simply ensure safe transport of binary data via HTTP. Encoding schemes that are commonly encountered include XOR, Base64, and hexadecimal representation using ASCII characters (see Chapter 3). It may be necessary to test various decodings on each component of a structured token to unpack it to its original form.

NOTE When an application handles a request containing a structured token, it may not actually process every component with the token or all the data contained in each component. In the previous example, the application may Base64-decode the token and then process only the "user" and "date" components. In cases where a token contains a blob of binary data, much of this data may be padding. Only a small part of it may actually be relevant to the validation that the server performs on the token. Narrowing down the sub-parts of a token that are actually required can often considerably reduce the amount of apparent entropy and complexity that the token contains.

HACK STEPS

1. Obtain a single token from the application, and modify it in systematic ways to determine whether the entire token is validated or whether some of its subcomponents are ignored. Try changing the token's value one byte at a time (or even one bit at a time) and resubmitting the modified token to the application to determine whether it is still accepted. If you find that certain portions of the token are not actually required to be correct, you can exclude these from any further analysis, potentially reducing the amount of work you need to perform. You can use the "char frobber" payload type in Burp Intruder to modify a token's value in one character position at a time, to help with this task.
2. Log in as several different users at different times, and record the tokens received from the server. If self-registration is available and you can choose your username, log in with a series of similar usernames containing small variations between them, such as A, AA, AAA, AAAA, AAAB, AAAC, AABA, and so on. If other user-specific data is submitted at login or stored in user profiles (such as an e-mail address), perform a similar exercise to vary that data systematically, and record the tokens received following login.
3. Analyze the tokens for any correlations that appear to be related to the username and other user-controllable data.
4. Analyze the tokens for any detectable encoding or obfuscation. Where the username contains a sequence of the same character, look for a corresponding character sequence in the token, which may indicate the use of XOR obfuscation. Look for sequences in the token containing only hexadecimal characters, which may indicate a hex encoding of an ASCII string or other information. Look for sequences that end in an equals sign and/or that contain only the other valid Base64 characters: a to z, A to Z, 0 to 9, +, and /.
5. If any meaning can be reverse-engineered from the sample of session tokens, consider whether you have sufficient information to attempt to guess the tokens recently issued to other application users. Find a page of the application that is session-dependent, such as one that returns an error message or a redirect elsewhere if accessed without a valid session. Then use a tool such as Burp Intruder to make large numbers of requests to this page using guessed tokens. Monitor the results for any cases in which the page is loaded correctly, indicating a valid session token.

TRY IT!

```
http://mdsec.net/auth/321/  
http://mdsec.net/auth/329/  
http://mdsec.net/auth/331/
```


Predictable Tokens

Some session tokens do not contain any meaningful data associating them with a particular user. Nevertheless, they can be guessed because they contain sequences or patterns that allow an attacker to extrapolate from a sample of tokens to find other valid tokens recently issued by the application. Even if the extrapolation involves some trial and error (for example, one valid guess per 1,000 attempts), this would still enable an automated attack to identify large numbers of valid tokens in a relatively short period of time.

Vulnerabilities relating to predictable token generation may be much easier to discover in commercial implementations of session management, such as web servers or web application platforms, than they are in bespoke applications. When you are remotely targeting a bespoke session management mechanism, your sample of issued tokens may be restricted by the server's capacity, the activity of other users, your bandwidth, network latency, and so on. In a laboratory environment, however, you can quickly create millions of sample tokens, all precisely sequenced and time-stamped, and you can eliminate interference caused by other users.

In the simplest and most brazenly vulnerable cases, an application may use a simple sequential number as the session token. In this case, you only need to obtain a sample of two or three tokens before launching an attack that will quickly capture 100% of currently valid sessions.

Figure 7-1 shows Burp Intruder being used to cycle the last two digits of a sequential session token to find values where the session is still active and can be hijacked. Here, the length of the server's response is a reliable indicator that a valid session has been found. The extract grep feature has also been used to show the name of the logged-in user for each session.

In other cases, an application's tokens may contain more elaborate sequences that take some effort to discover. The types of potential variations you might encounter here are open-ended, but the authors' experience in the field indicates that predictable session tokens commonly arise from three different sources:

- Concealed sequences
- Time dependency
- Weak random number generation

We will look at each of these areas in turn.

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.

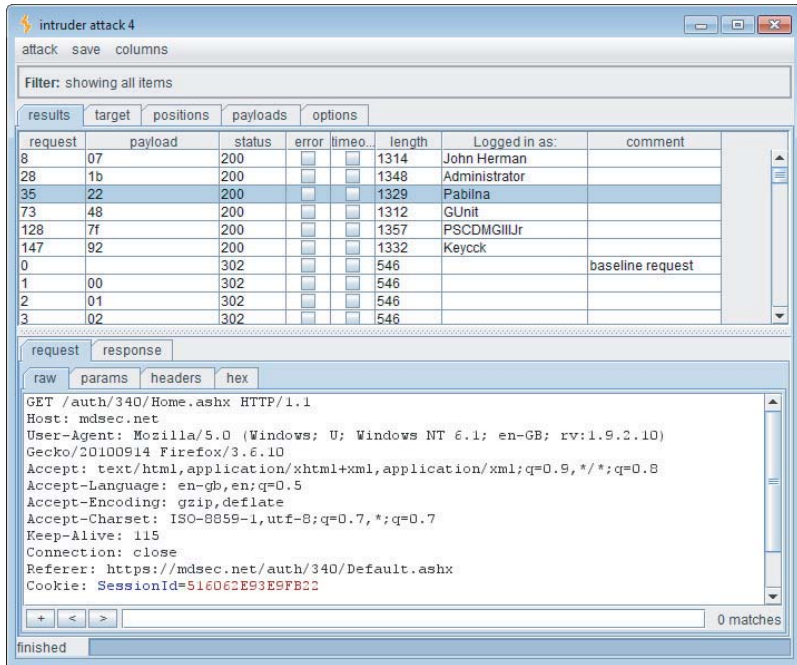


Figure 7-1: An attack to discover valid sessions where the session token is predictable

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. In addition to the mixed-case alphabetic and numeric characters, there is a + character, which is also valid in a Base64-encoded string. Running the tokens through a Base64 decoder reveals the following:

--Ö\$
.íÀŽ
Æ'«ø
^W-b
ö,î
?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 is still no visible pattern. However, if you subtract each number from the previous one, you arrive at the following:

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

which immediately reveals the concealed pattern. 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. Although any given sequence of tokens on its own may appear to be random, the same sequence coupled with information about the time at which each token was generated may contain a discernible pattern. In a busy application with a large number of sessions being created each second, a scripted attack may succeed in identifying large numbers of other users' tokens.

When testing the web application of an online retailer, the authors encountered the following sequence of session tokens:

```
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 immediately obvious:

- 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.
- 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.

This second observation immediately alerts us to the role played by time in generating session tokens. Apparently, only five tokens have been issued between the two token-grabbing exercises. However, a period of approximately 10 minutes has elapsed. The most likely explanation is that the second number is time-dependent and is probably a simple count of milliseconds.

Indeed, our hunch is correct. In a subsequent phase of our testing we perform a code review, which reveals the following token-generation algorithm:

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

Given our analysis of how tokens are created, it is straightforward to construct 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.

TRY IT!

```
http://mdsec.net/auth/339/  
http://mdsec.net/auth/340/  
http://mdsec.net/auth/347/  
http://mdsec.net/auth/351/
```

Weak Random Number Generation

Very little that occurs inside a computer is random. Therefore, when randomness is required for some purpose, software uses various techniques to generate numbers in a pseudorandom manner. Some of the algorithms used produce sequences that appear to be stochastic and manifest an even spread across the range of possible values. Nevertheless, they can be extrapolated forwards or backwards with perfect accuracy by anyone who obtains a small sample of values.

When a predictable pseudorandom number generator is used to produce session tokens, the resulting tokens are vulnerable to sequencing by an attacker.

Jetty is a popular web server written in 100% Java that provides a session management mechanism for use by applications running on it. In 2006, Chris Anley of NGSSoftware discovered that the mechanism was vulnerable to a session token prediction attack. The server used the Java API `java.util.Random` to generate session tokens. This implements a “linear congruential generator,” which generates the next number in the 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.

Knowing this algorithm and a single number generated by it, we can easily derive the sequence of numbers that the algorithm will generate next. With a little number theory, we also can derive the sequence that it generated previously. This means that an attacker who obtains a single session token from the server can obtain the tokens of all current and future sessions.

NOTE Sometimes when tokens are created based on the output of a pseudorandom number generator, developers decide to construct each token by concatenating several sequential outputs from the generator. The perceived rationale for this is that it creates a longer, and therefore “stronger,” token. However, this tactic is usually a mistake. If an attacker can obtain several consecutive outputs from the generator, this may enable him to infer some information about its internal state. In fact, it may be easier for the attacker to extrapolate the generator’s sequence of outputs, either forward or backward.

Other off-the-shelf application frameworks use surprisingly simple or predictable sources of entropy in session token generation, much of which is deterministic. For example, in PHP frameworks 5.3.2 and earlier, the session token is generated

based on the client's IP address, epoch time at token creation, microseconds at token creation, and a linear congruential generator. Although there are several unknown values here, some applications may disclose information that allows them to be inferred. A social networking site may disclose the login time and IP address of site users. Additionally, the seed used in this generator is the time when the PHP process started, which could be determined to lie within a small range of values if the attacker is monitoring the server.

NOTE This is an evolving area of research. The weaknesses in PHP's session token generation were pointed out on the Full Disclosure mailing list in 2001 but were not demonstrated to be actually exploitable. The 2001 theory was finally put into practice by Samy Kamkar with the `phpwn` tool in 2010.

Testing the Quality of Randomness

In some cases, you can identify patterns in a series of tokens just from visual inspection, or from a modest amount of manual analysis. In general, however, you need to use a more rigorous approach to testing the quality of randomness within an application's tokens.

The standard approach to this task applies the principles of statistical hypothesis testing and employs various well-documented tests that look for evidence of nonrandomness within a sample of tokens. The high-level steps in this process are as follows:

1. Start with the hypothesis that the tokens are randomly generated.
2. Apply a series of tests, each of which observes specific properties of the sample that are likely to have certain characteristics if the tokens are randomly generated.
3. For each test, calculate the probability of the observed characteristics occurring, working on the assumption that the hypothesis is true.
4. If this probability falls below a certain level (the "significance level"), reject the hypothesis and conclude that the tokens are not randomly generated.

The good news is you don't have to do any of this manually! The best tool that is currently available for testing the randomness of web application tokens is Burp Sequencer. This tool applies several standard tests in a flexible way and gives you clear results that are easy to interpret.

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, as shown in Figure 7-2. You can also

configure various options that affect how tokens are collected, and then click the start capture button to begin capturing tokens. If you have already obtained a suitable sample of tokens through other means (for example, by saving the results of a Burp Intruder attack), you can use the manual load tab to skip the capturing of tokens and proceed straight to the statistical analysis.

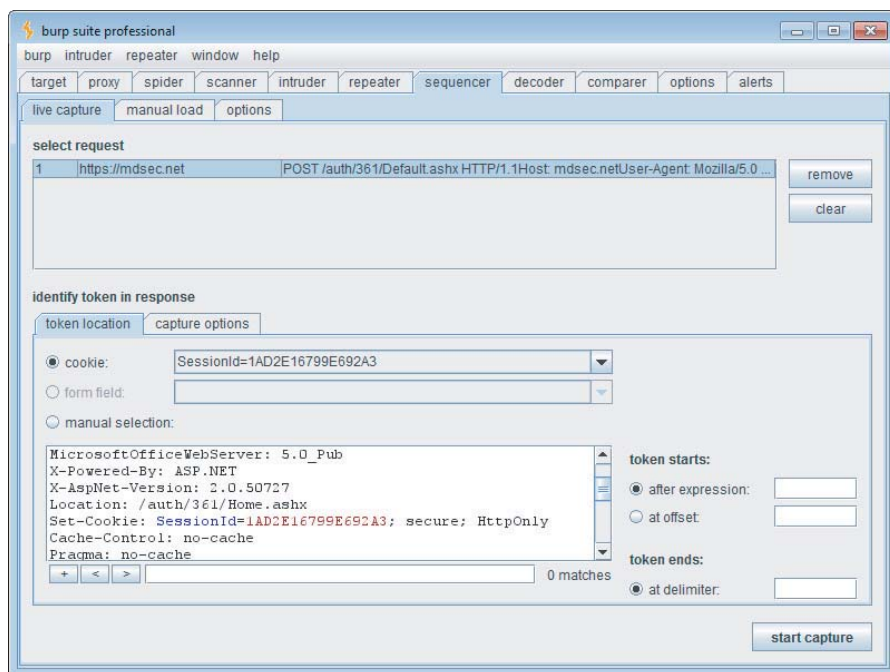


Figure 7-2: Configuring Burp Sequencer to test the randomness of a session token

When you have obtained a suitable sample of tokens, you can perform the statistical analysis on the sample. You can also perform interim analyses while the sample is still being captured. In general, obtaining a larger sample improves the reliability of the analysis. The minimum sample size that Burp requires is 100 tokens, but ideally you should obtain a much larger sample than this. If the analysis of a few hundred tokens shows conclusively that the tokens fail the randomness tests, you may reasonably decide that it is unnecessary to capture further tokens. Otherwise, you should continue capturing tokens and re-perform the analysis periodically. If you capture 5,000 tokens that are shown to pass the randomness tests, you may decide that this is sufficient. However, to achieve compliance with the formal FIPS tests for randomness, you need to obtain a sample of 20,000 tokens. This is the largest sample size that Burp supports.

Burp Sequencer performs the statistical tests at character level and bit level. The results of all tests are aggregated to give an overall estimate of the number

of bits of effective entropy within the token; this the key result to consider. However, you can also drill down into the results of each test to understand exactly how and why different parts of the token passed or failed each test, as shown in Figure 7-3. The methodology used for each type of test is described beneath the test results.

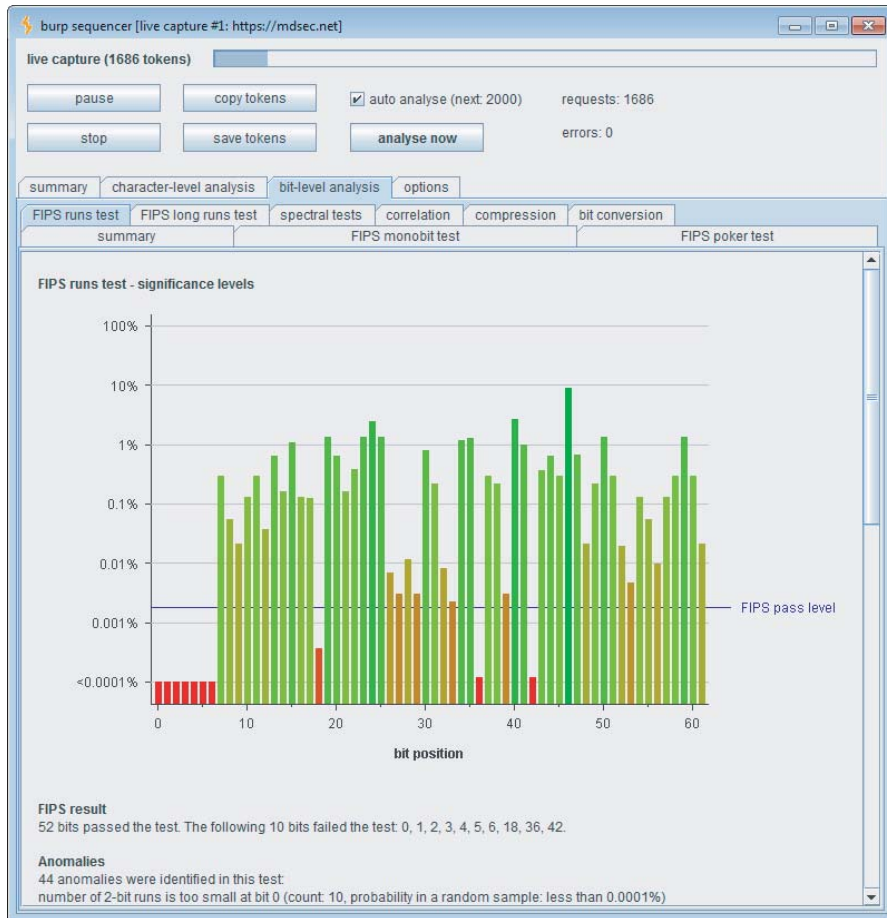


Figure 7-3: Analyzing the Burp Sequencer results to understand the properties of the tokens that were tested

Note that Burp performs all tests individually on each character and bit of data within the token. In many cases, you will find that large parts of a structured token are not random; this in itself may not present any kind of weakness. What matters is that the token contains a sufficient number of bits that do pass the randomness tests. For example, if a large token contains 1,000 bits of information, and only 50 of these bits pass the randomness tests, the token as a whole is no less robust than a 50-bit token that fully passes the tests.

NOTE Keep in mind two important caveats when performing statistical tests for randomness. These caveats affect the correct interpretation of the test results and their consequences for the application's security posture. First, tokens that are generated in a completely deterministic way may pass the statistical tests for randomness. For example, a linear congruential pseudorandom number generator, or an algorithm that computes the hash of a sequential number, may produce output that passes the tests. 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.

Second, 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. But attempting to predict the value of that bit in the next token, based on the observed characteristics, may be little more reliable than blind guesswork. Multiplying this unreliability across a large number of bits that need to be predicted simultaneously may mean that the probability of making a correct prediction is extremely low.

HACK STEPS

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.

TRY IT!

`http://mdsec.net/auth/361/`

Encrypted Tokens

Some applications use tokens that contain meaningful information about the user and seek to avoid the obvious problems that this entails by encrypting the tokens before they are issued to users. Since the tokens are encrypted using a secret key that is unknown to users, this appears to be a robust approach, because users will be unable to decrypt the tokens and tamper with their contents.

However, in some situations, depending on the encryption algorithm used and the manner in which the application processes the tokens, it may nonetheless be possible for users to tamper with the tokens' meaningful contents without actually decrypting them. Bizarre as it may sound, these are actually viable attacks that are sometimes easy to deliver, and numerous real-world applications have proven vulnerable to them. The kinds of attacks that are applicable depend on the exact cryptographic algorithm that is being used.

ECB Ciphers

Applications that employ encrypted tokens use a symmetric encryption algorithm so that tokens received from users can be decrypted to recover their meaningful contents. Some symmetric encryption algorithms use an "electronic codebook" (ECB) 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. For some types of data, such as bitmap images, this means that meaningful information from the plaintext can be discerned within the ciphertext, as illustrated in Figure 7-4.

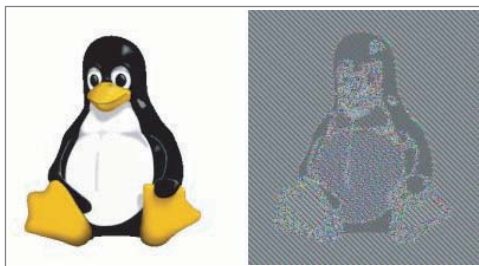


Figure 7-4: Patterns within plaintext that is encrypted using an ECB cipher may be visible within the resulting ciphertext.

In spite of this shortcoming with ECB, these ciphers are often used for encrypting information within web applications. Even in situations where the problem of patterns within plaintext does not arise, vulnerabilities can still exist. This is because of the cipher's behavior of encrypting identical plaintext blocks into identical ciphertext blocks.

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=634430423694715000;
```

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
```

Now, 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. Depending on how exactly the application processes the resulting decrypted token, this may enable the attacker to switch to a different user or escalate privileges.

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 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
219;user     A5BC430A73B38C14
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.

TRY IT!

```
http://mdsec.net/auth/363/
```

CBC Ciphers

The shortcomings in ECB ciphers led to the development of cipher block chaining (CBC) ciphers. With a CBC cipher, before each block of plaintext is encrypted it is XORed against the preceding block of ciphertext, as shown in Figure 7-5. 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.

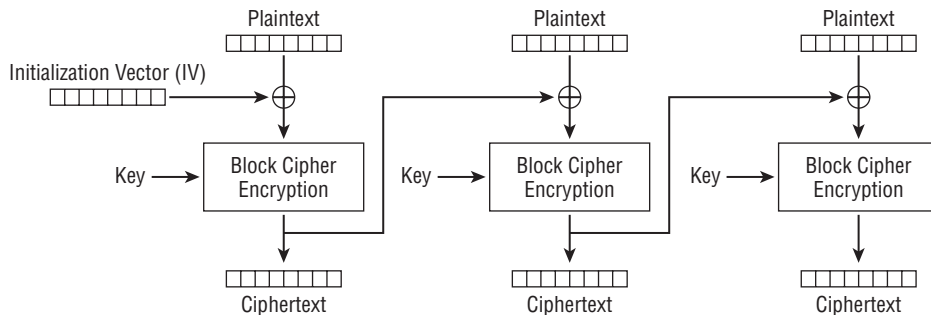


Figure 7-5: In a CBC cipher, each block of plaintext is XORed against the preceding block of ciphertext before being encrypted.

Because CBC ciphers avoid some of the problems with ECB ciphers, standard symmetric encryption algorithms such as DES and AES frequently are used in CBC mode. However, the way in which CBC-encrypted tokens are often employed in web applications means that an attacker may be able to manipulate parts of the decrypted tokens without knowing the secret key.

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.

Let's see how. 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:

```
????????32858301;app=eBankProdTC;uid=216;time=6343303;
????????32758321;app=eBankProdTC;uid=216;time=6343303;
rnd=1914????????;app=eBankProdTC;uid=216;time=6343303;
rnd=1914????????;app=eAankProdTC;uid=216;time=6343303;
rnd=191432758301????????nkPgodTC;uid=216;time=6343303;
rnd=191432758301????????nkProdTC;uid=216;time=6343303;
rnd=191432758301;app=eBa????????;uid=216;time=6343303;
rnd=191432758301;app=eBa????????;uid=226;time=6343303;
rnd=191432758301;app=eBankProdTC????????;time=6343303;
rnd=191432758301;app=eBankProdTC????????;time=6343503;
```

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. As already described, this difference occurs because the decrypted text is XORed against the preceding block of ciphertext, which the attacker has slightly modified.

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. If the application rejects tokens containing any invalid data, the attack fails. Often, however, 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 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. First, you need to log in to the application using your own account. Then you find a page of the application that depends on a logged-in session and shows the identity of the logged-in user within the response. Typically, the user's home landing page or account details page serves this purpose. Figure 7-6 shows Burp Intruder set up to target the user's home page, with the encrypted session token marked as a payload position.

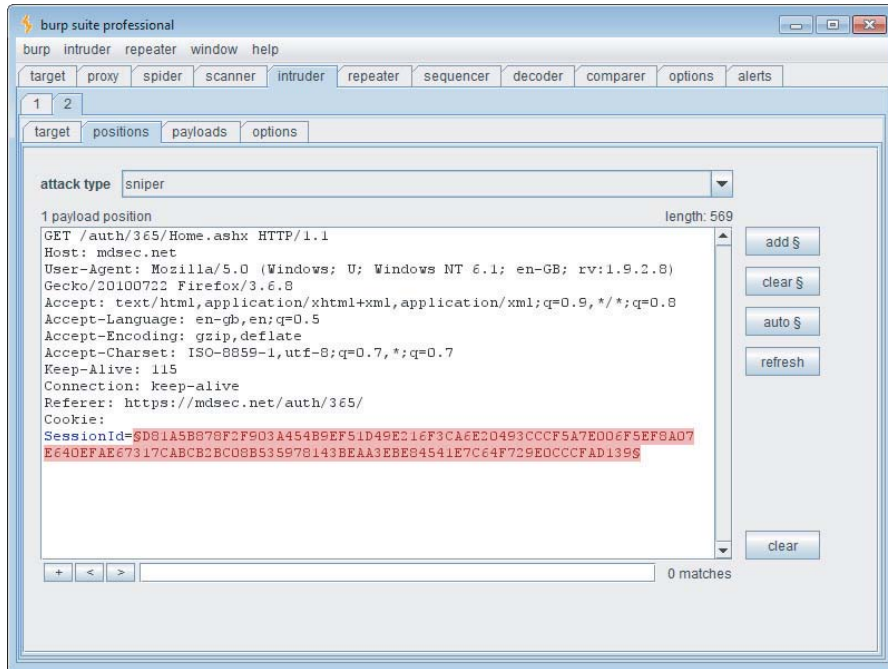


Figure 7-6: Configuring Burp Intruder to modify an encrypted session token

Figure 7-7 shows the required payload configuration. It tells Burp to operate on the token's original value, treating it as ASCII-encoded hex, and to flip each bit at each character position. This approach is ideal because it requires a relatively small number of requests (eight requests per byte of data in the token) and almost always identifies whether the application is vulnerable. This allows you to use a more focused attack to perform actual exploitation.

When the attack is executed, the initial requests do not cause any noticeable change in the application's responses, and the user's session is still intact. This is interesting in itself, because it indicates that the first part of the token is not being used to identify the logged-in user. Many of the requests later in the attack cause a redirection to the login page, indicating that modification has invalidated the token in some way. Crucially, there is also a run of requests where the response appears to be part of a valid session but is not associated with the original user identity. This corresponds to the block of the token that contains the `uid` value. In some cases, the application simply displays "unknown user," indicating that the modified uid did not correspond to an actual user, and so the attack failed. In other cases, it shows the name of a different registered user of the application, proving conclusively that the attack has succeeded. Figure 7-8 shows the results of the attack. Here we have defined an extract grep column to display the identity of the logged-in user and have set a filter to hide the responses that are redirections to the login page.

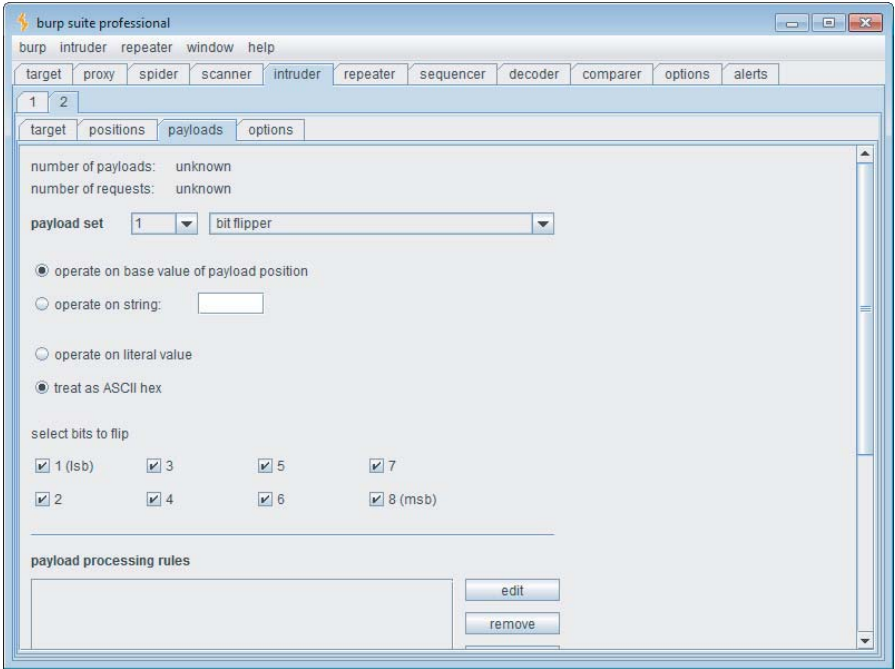


Figure 7-7: Configuring Burp Intruder to flip each bit in the encrypted token

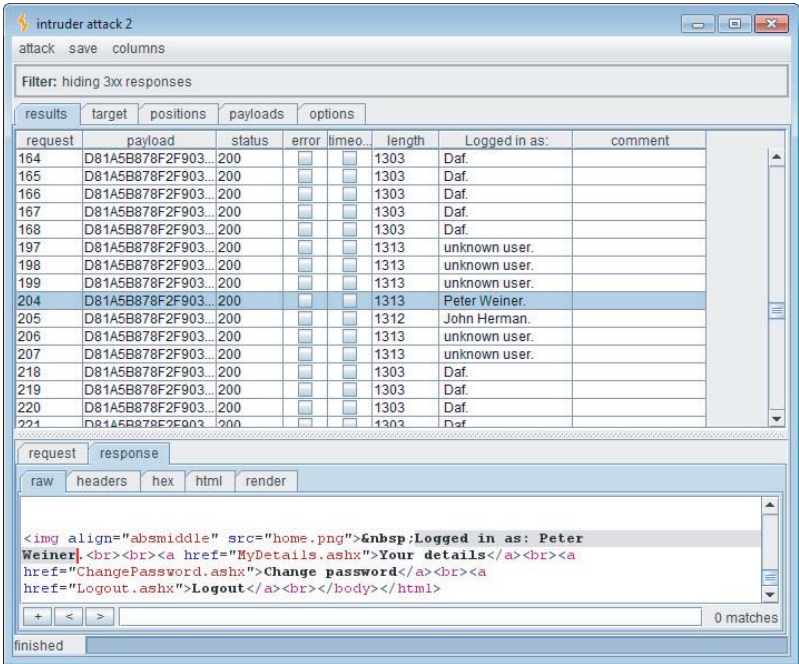


Figure 7-8: A successful bit flipping attack against an encrypted token

Having identified the vulnerability, you can proceed to exploit it with a more focused attack. To do this, you would determine from the results exactly which block of the encrypted token is being tampered with when the user context changes. Then you would deliver an attack that tests numerous further values within this block. You could use the numbers payload type within Burp Intruder to do this.

TRY IT!

```
http://mdsec.net/auth/365/
```

NOTE Some applications use the technique of encrypting meaningful data within request parameters more generally in an attempt to prevent tampering of data, such as the prices of shopping items. In any location where you see apparently encrypted data that plays a key role in application functionality, you should try the bit-flipping technique to see whether you can manipulate the encrypted information in a meaningful way to interfere with application logic.

In seeking to exploit the vulnerability described in this section, your objective would of course be to masquerade as different application users — ideally an administrative user with higher privileges. If you are restricted to blindly manipulating parts of an encrypted token, this may require a degree of luck. However, in some cases the application may give you more assistance. When an application employs symmetric encryption to protect data from tampering by users, it is common for the same encryption algorithm and key to be used throughout the application. In this situation, if any application function discloses to the user the decrypted value of an arbitrary encrypted string, this can be leveraged to fully decrypt any item of protected information.

One application observed by the authors contained a file upload/download function. Having uploaded a file, users were given a download link containing a filename parameter. To prevent various attacks that manipulate file paths, the application encrypted the filename within this parameter. However, if a user requested a file that had been deleted, the application displayed an error message showing the *decrypted* name of the requested file. This behavior could be leveraged to find the plaintext value of any encrypted string used within the application, including the values of session tokens. The session tokens were found to contain various meaningful values in a structured format that was vulnerable to the type of attack described in this section. Because these values included textual usernames and application roles, rather than numeric identifiers, it would have been extremely difficult to perform a successful exploit using only blind bit flipping. However, using the filename decryptor function, it was possible to systematically manipulate bits of a token while viewing the results.

This allowed the construction of a token that, when decrypted, specified a valid user and administrative role, enabling full control of the application.

NOTE Other techniques may allow you to decrypt encrypted data used by the application. A “reveal” encryption oracle can be abused to obtain the cleartext value of an encrypted token. Although this can be a significant vulnerability when decrypting a password, decrypting a session token does not provide an immediate means of compromising other users’ sessions. Nevertheless, the decrypted token provides useful insight into the cleartext structure, which is useful in conducting a targeted bit-flipping attack. See Chapter 11 for more details about “reveal” encryption oracle attacks.

Side channel attacks against padding oracles may be used to compromise encrypted tokens. See Chapter 18 for more details.

HACK STEPS

In many situations where encrypted tokens are used, actual exploitability may depend on various factors, including the offsets of block boundaries relative to the data you need to attack, and the application’s tolerance of the changes that you cause to the surrounding plaintext structure. Working completely blind, it may appear difficult to construct an effective attack, however in many situations this is in fact possible.

1. Unless the session token is obviously meaningful or sequential in itself, always consider the possibility that it might be encrypted. You can often identify that a block-based cipher is being used by registering several different usernames and adding one character in length each time. If you find a point where adding one character results in your session token jumping in length by 8 or 16 bytes, then a block cipher is probably being used. You can confirm this by continuing to add bytes to your username, and looking for the same jump occurring 8 or 16 bytes later.
2. ECB cipher manipulation vulnerabilities are normally difficult to identify and exploit in a purely black-box context. You can try blindly duplicating and moving the ciphertext blocks within your token, and reviewing whether you remain logged in to the application within your own user context, or that of another user, or none at all.
3. You can test for CBC cipher manipulation vulnerabilities by running a Burp Intruder attack over the whole token, using the “bit flipping” payload source. If the bit flipping attack identifies a section within the token, the manipulation of which causes you to remain in a valid session, but as a different or nonexistent user, perform a more focused attack on just this section, trying a wider range of values at each position.

4. During both attacks, monitor the application's responses to identify the user associated with your session following each request, and try to exploit any opportunities for privilege escalation that may result.
5. If your attacks are unsuccessful, but it appears from step 1 that variable-length input that you control is being incorporated into the token, you should try generating a series of tokens by adding one character at a time, at least up to the size of blocks being used. For each resulting token, you should reperform steps 2 and 3. This will increase the chance that the data you need to modify is suitably aligned with block boundaries for your attack to succeed.

Weaknesses in Session Token Handling

No matter how effective an application is at ensuring that the session tokens it generates do not contain any meaningful information and are not susceptible to analysis or prediction, its session mechanism will be wide open to attack if those tokens are not handled carefully after generation. For example, if tokens are disclosed to an attacker via some means, the attacker can hijack user sessions even if predicting the tokens is impossible.

An application's unsafe handling of tokens can make it vulnerable to attack in several ways.

COMMON MYTH

"Our token is secure from disclosure to third parties because we use SSL."

Proper use of SSL certainly helps protect session tokens from being captured. But various mistakes can still result in tokens being transmitted in cleartext even when SSL is in place. And various direct attacks against end users can be used to obtain their tokens.

COMMON MYTH

"Our token is generated by the platform using mature, cryptographically sound technologies, so it is not vulnerable to compromise."

An application server's default behavior is often to create a session cookie when the user first visits the site and to keep this available for the user's entire interaction with the site. As described in the following sections, this may lead to various security vulnerabilities in how the token is handled.

Disclosure of Tokens on the Network

This area of vulnerability arises when the session token is transmitted across the network in unencrypted form, enabling a suitably positioned eavesdropper to obtain the token and therefore masquerade as the legitimate user. 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 each case, this includes both authorized personnel of the relevant organization and any external attackers who have compromised the infrastructure concerned.

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 this situation, an attack against the user's session is often unnecessary because the attacker can already view privileged information and can log in using captured credentials to perform other malicious actions. However, there may still be instances where the user's session is the primary target. For example, if the captured credentials are insufficient to perform a second login (for example, in a banking application, they may include a number displayed on a changing physical token, or specific digits from the user's PIN), the attacker may need to hijack the eavesdropped session to perform arbitrary actions. Or if logins are audited closely, and the user is notified of each successful login, an attacker may want to avoid performing his own login to be as stealthy as possible.

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:

- Some applications elect to use HTTPS to protect the user's credentials during login but then revert to HTTP for the remainder of the user's session. Many web mail applications behave in this way. In this situation, an eavesdropper cannot intercept the user's credentials but may still capture the session token. The Firesheep tool, released as a plug-in for Firefox, makes this an easy process.
- Some applications use HTTP for preauthenticated areas of the site, such as the site's front page, but switch to HTTPS from the login page onward. However, in many cases the user is issued a session token at the first page visited, and this token is not modified when the user logs in. The user's session, which is originally unauthenticated, is upgraded to an authenticated session after login. In this situation an eavesdropper can intercept a user's token before login, wait for the user's communications to switch to

HTTPS, indicating that the user is logging in, and then attempt to access a protected page (such as My Account) using that token.

- Even if the application issues a fresh token following successful login, and uses HTTPS from the login page onward, the token for the user's authenticated session may still be disclosed. This can happen if the user revisits a preauthentication page (such as Help or About), either by following links within the authenticated area, by using the back button, or by typing the URL directly.
- In a variation on the preceding case, the application may attempt to switch to HTTPS when the user clicks the Login link. However, it may still accept a login over HTTP if the user modifies the URL accordingly. In this situation, a suitably positioned attacker can modify the pages returned in the preauthenticated areas of the site so that the Login link points to an HTTP page. Even if the application issues a fresh session token after successful login, the attacker may still intercept this token if he has successfully downgraded the user's connection to HTTP.
- Some applications use HTTP for all static content within the application, such as images, scripts, style sheets, and page templates. This behavior is often indicated by a warning within the user's browser, as shown in Figure 7-9. When a browser shows this warning, it has already retrieved the relevant item over HTTP, so the session token has already been transmitted. The purpose of the browser's warning is to let the user decline to process response data that has been received over HTTP and so may be tainted. As described previously, an attacker can intercept the user's session token when the user's browser accesses a resource over HTTP and use this token to access protected, nonstatic areas of the site over HTTPS.




Figure 7-9: Browsers present a warning when a page accessed over HTTPS contains items accessed over HTTP.

- Even if an application uses HTTPS for every page, including unauthenticated areas of the site and static content, there may still be circumstances in which users' tokens are transmitted over HTTP. If an attacker can somehow induce a user to make a request over HTTP (either to the HTTP

service on the same server if one is running or to `http://server:443/` otherwise), his token may be submitted. Means by which the attacker may attempt this include sending the user a URL in an e-mail or instant message, placing autoloading links into a website the attacker controls, or using clickable banner ads. (See Chapters 12 and 13 for more details about techniques of this kind for delivering attacks against other users.)

HACK STEPS

1. Walk through the application in the normal way from first access (the “start” URL), through the login process, and then through all of the application’s functionality. Keep a record of every URL visited, and note every instance in which a new session token is received. Pay particular attention to login functions and transitions between HTTP and HTTPS communications. This can be achieved manually using a network sniffer such as Wireshark or partially automated using the logging functions of your intercepting proxy, as shown in Figure 7-10.



The screenshot shows the Burp Suite v1.01 professional interface. The 'intercept' tab is selected, displaying a table of intercepted requests. The table has columns for target, method, URL, type, SSL, IP, status, and length. The data shows several requests to www.blogger.com and google-analytics.com, including GET requests for /, /start, /css/blogger.css, /css/blogger_main.css, /css/flexible_buttons.css, and /__utm.gif?utmwv=1&utm=13...gif. The status is 200 for all requests, and the length is 461, 20,532, 1,043, 23,933, 8,484, and 329 respectively. The cookies are listed as NSC_cmphhfs-fyu=0a1402010050 and JSESSIONID=E86731C7CFF0B2F9300.

target	method	URL	type	SSL	IP	status	length
www.blogger.com:80	GET	/			66.102.15.100	302	461
www.blogger.com:80	GET	/start			66.102.15.100	200	20,532
www.blogger.com:80	GET	/css/blogger.css	css		66.102.15.100	200	1,043
www.blogger.com:80	GET	/css/blogger_main.css	css		66.102.15.100	200	23,933
www.blogger.com:80	GET	/css/flexible_buttons.css	css		66.102.15.100	200	8,484
www.google-analytics.co...	GET	/__utm.gif?utmwv=1&utm=13...gif	gif		64.233.183.103	200	329
www.blogger.com:80	GET	/app/scripts/dom.common.js	js		66.102.15.100	200	6,386
www.blogger.com:80	GET	/app/scripts/detect.js	js		66.102.15.100	200	2,899

Figure 7-10: Walking through an application to identify locations where new session tokens are received.

2. If HTTP cookies are being used as the transmission mechanism for session tokens, verify whether the `secure` flag is set, preventing them from ever being transmitted over unencrypted connections.
3. Determine whether, in the normal use of the application, session tokens are ever transmitted over an unencrypted connection. If so, they should be regarded as vulnerable to interception.
4. Where the start page uses HTTP, and the application switches to HTTPS for the login and authenticated areas of the site, verify whether a new token is issued following login, or whether a token transmitted during the HTTP stage is still being used to track the user’s authenticated session. Also verify whether the application will accept login over HTTP if the login URL is modified accordingly.

5. Even if the application uses HTTPS for every page, verify whether the server is also listening on port 80, running any service or content. If so, visit any HTTP URL directly from within an authenticated session, and verify whether the session token is transmitted.
6. In cases where a token for an authenticated session is transmitted to the server over HTTP, verify whether that token continues to be valid or is immediately terminated by the server.

TRY IT!

```
http://mdsec.net/auth/369/  
http://mdsec.net/auth/372/  
http://mdsec.net/auth/374/
```

Disclosure of Tokens in Logs

Aside from the clear-text transmission of session tokens in network communications, the most common place where tokens are simply disclosed to unauthorized view is in system logs of various kinds. Although it is a rarer occurrence, the consequences of this kind of disclosure are usually more serious. Those logs may be viewed by a far wider range of potential attackers, not just by someone who is suitably positioned to eavesdrop on the network.

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. Or an administrator may consult a log of recent sessions in the course of investigating a security breach. Often, this kind of monitoring and control functionality discloses the actual session token associated with each session. And often, the functionality is poorly protected, allowing unauthorized users to access the list of current session tokens, and thereby hijack the sessions of all application users.

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, as shown in Figure 7-11

Some of these vulnerabilities arise even if HTTPS is used throughout the application.

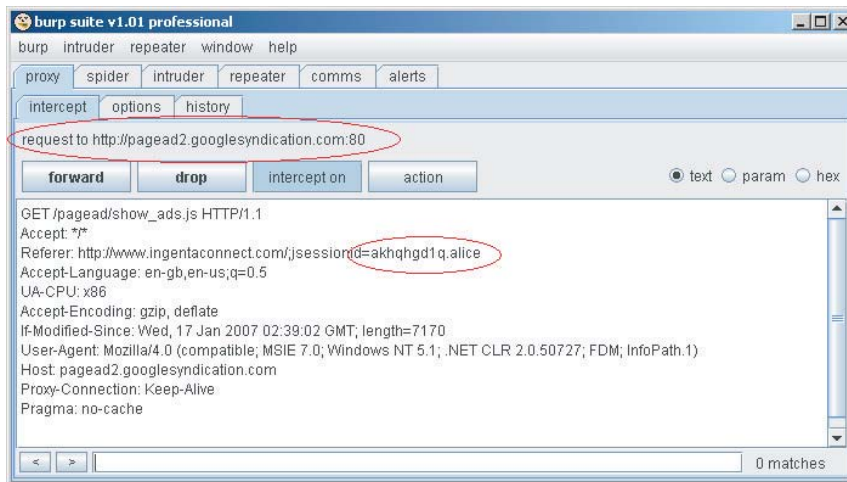


Figure 7-11: When session tokens appear in URLs, these are transmitted in the Referer header when users follow an off-site link or their browser loads an off-site resource.

The final case just described presents an attacker with a highly effective means of capturing session tokens in some applications. 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.

NOTE Current versions of Internet Explorer do not include a Referer header when following off-site links contained in a page that was accessed over HTTPS. In this situation, Firefox includes the Referer header provided that the off-site link is also being accessed over HTTPS, even if it belongs to a different domain. Hence, sensitive data placed in URLs is vulnerable to leakage in Referer logs even where SSL is being used.

HACK STEPS

1. Identify all the functionality within the application, and locate any logging or monitoring functions where session tokens can be viewed. Verify who can access this functionality—for example, administrators, any authenticated user, or any anonymous user. See Chapter 4 for techniques for discovering hidden content that is not directly linked from the main application.
2. Identify any instances within the application where session tokens are transmitted within the URL. It may be that tokens are generally transmitted in a more secure manner but that developers have used the URL in specific cases to work around particular difficulties. For example, this behavior is often observed where a web application interfaces with an external system.
3. If session tokens are being transmitted in URLs, attempt to find any application functionality that enables you to inject arbitrary off-site links into pages viewed by other users. Examples include functionality implementing a message board, site feedback, question-and-answer, and so on. If so, submit links to a web server you control and wait to see whether any users' session tokens are received in your Referer logs.
4. If any session tokens are captured, attempt to hijack user sessions by using the application as normal but substituting a captured token for your own. You can do this by intercepting the next response from the server and adding a Set-Cookie header of your own with the captured cookie value. In Burp, you can apply a single Suite-wide configuration that sets a specific cookie in all requests to the target application to allow easy switching between different session contexts during testing.
6. If a large number of tokens are captured, and session hijacking allows you to access sensitive data such as personal details, payment information, or user passwords, you can use the automated techniques described in Chapter 14 to harvest all desired data belonging to other application users.

TRY IT!

```
http://mdsec.net/auth/379/
```

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. In virtually every application, there is no legitimate reason why any user should have more than one session active at one time. Of course, it is fairly common for a user to abandon an active session and start a new one — for example, because he closes a browser window or moves to a different computer. But 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.

A related but distinct weakness is for applications to use “static” tokens. These look like session tokens and may initially appear to function like them, but in fact they are no such thing. In these applications, 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. Applications like this really involve a misunderstanding about the whole concept of what a session is, and the benefits it provides for managing and controlling access to the application. Sometimes, applications operate like this as a means of implementing poorly designed “remember me” functionality, and the static token is accordingly stored in a persistent cookie (see Chapter 6). Sometimes the tokens themselves are vulnerable to prediction attacks, making the vulnerability far more serious. Rather than compromising the sessions of currently logged-in users, a successful attack compromises, for all time, the accounts of all registered users.

Other kinds of strange application behavior are also occasionally observed that demonstrate a fundamental defect in the relationship between tokens and sessions. One example is where a meaningful token is constructed based on a username and a random component. For example, consider the token:

```
dXNlcj1kYWY7cjE9MTMwOTQxODEyMTM0NTkwMTI=
```

which Base64-decodes to:

```
user=daf;r1=13094181213459012
```

After extensive analysis of the `r1` component, we may conclude that this cannot be predicted based on a sample of values. However, if the application's session processing logic is awry, it may be that an attacker simply needs to submit *any* valid value as `r1` and *any* valid value as `user` to access a session under the security context of the specified user. This is essentially an access control vulnerability, because decisions about access are being made on the basis of user-supplied data outside of the session (see Chapter 8). It arises because the application effectively uses session tokens to signify that the requester has established *some* kind of valid session with the application. However, the user context in which that session is processed is not an integral property of the session itself but is determined per-request through some other means. In this case, that means can be directly controlled by the requester.

HACK STEPS

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.**

TRY IT!

```
http://mdsec.net/auth/382/  
http://mdsec.net/auth/385/
```

Vulnerable Session Termination

Proper termination of sessions is important for two reasons. First, keeping the life span of a session as short as is necessary reduces the window of opportunity within which an attacker may capture, guess, or misuse a valid session token.

Second, it provides users with a means of invalidating an existing session when they no longer require it. This enables them to reduce this window further and to take some responsibility for securing their session in a shared computing environment. The main weaknesses in session termination functions involve failures to meet these two key objectives.

Some applications do not enforce effective session expiration. Once created, a session may remain valid for many days after the last request is received, before the server eventually expires the session. If tokens are vulnerable to some kind of sequencing flaw that is particularly difficult to exploit (for example, 100,000 guesses for each valid token identified), an attacker may still be able to capture the tokens of every user who has accessed the application in the recent past.

Some applications do not provide effective logout functionality:

- In some cases, a logout function is simply not implemented. Users have no means of causing the application to invalidate their session.
- In some cases, the logout function does not actually cause the server to invalidate the session. The server removes the token from the user's browser (for example, by issuing a `Set-Cookie` instruction to blank the token). However, if the user continues to submit the token, the server still accepts it.
- In the worst cases, when a user clicks Logout, this fact is not communicated to the server, so the server performs no action. Rather, a client-side script is executed that blanks the user's cookie, meaning that subsequent requests return the user to the login page. An attacker who gains access to this cookie could use the session as if the user had never logged out.

Some applications that do not use authentication still contain functionality that enables users to build up sensitive data within their session (for example, a shopping application). Yet typically they do not provide any equivalent of a logout function for users to terminate their session.

HACK STEPS

1. **Do not fall into the trap of examining actions that the application performs on the client-side token (such as cookie invalidation via a new `Set-Cookie` instruction, client-side script, or an expiration time attribute). In terms of session termination, nothing much depends on what happens to the token within the client browser. Rather, investigate whether session expiration is implemented on the server side:**
 - a. **Log in to the application to obtain a valid session token.**
 - b. **Wait for a period without using this token, and then submit a request for a protected page (such as "my details") using the token.**

- c. If the page is displayed as normal, the token is still active.
 - d. Use trial and error to determine how long any session expiration time-out is, or whether a token can still be used days after the last request using it. Burp Intruder can be configured to increment the time interval between successive requests to automate this task.
2. Determine whether a logout function exists and is prominently made available to users. If not, users are more vulnerable, because they have no way to cause the application to invalidate their session.
 3. Where a logout function is provided, test its effectiveness. After logging out, attempt to reuse the old token and determine whether it is still valid. If so, users remain vulnerable to some session hijacking attacks even after they have “logged out.” You can use Burp Suite to test this, by selecting a recent session-dependent request from the proxy history and sending it to Burp Repeater to reissue after you have logged out from the application.

TRY IT!

```
http://mdsec.net/auth/423/  
http://mdsec.net/auth/439/  
http://mdsec.net/auth/447/  
http://mdsec.net/auth/452/  
http://mdsec.net/auth/457/
```

Client Exposure to Token Hijacking

An attacker can target other users of the application in an attempt to capture or misuse the victim’s session token in various ways:

- An obvious payload for cross-site scripting attacks is to query the user’s cookies to obtain her session token, which can then be transmitted to an arbitrary server controlled by the attacker. All the various permutations of this attack are described in detail in Chapter 12.
- Various other attacks against users can be used to hijack the user’s session in different ways. With session fixation vulnerabilities, an attacker feeds a known session token to a user, waits for her to log in, and then hijacks her session. With cross-site request forgery attacks, an attacker makes a crafted request to an application from a web site he controls, and he exploits the fact that the user’s browser automatically submits her current cookie with this request. These attacks are also described in Chapter 12.

HACK STEPS

1. Identify any cross-site scripting vulnerabilities within the application, and determine whether these can be exploited to capture the session tokens of other users (see Chapter 12).
2. If the application issues session tokens to unauthenticated users, obtain a token and perform a login. If the application does not issue a fresh token *following* a successful login, it is vulnerable to session fixation.
3. Even if the application does not issue session tokens to unauthenticated users, obtain a token by logging in, and then return to the login page. If the application is willing to return this page even though you are already authenticated, submit another login as a different user using the same token. If the application does not issue a fresh token after the second login, it is vulnerable to session fixation.
4. Identify the format of session tokens used by the application. Modify your token to an invented value that is validly formed, and attempt to log in. If the application allows you to create an authenticated session using an invented token, it is vulnerable to session fixation.
5. If the application does not support login, but processes sensitive user information (such as personal and payment details), and allows this to be displayed after submission (such as on a “verify my order” page), carry out the previous three tests in relation to the pages displaying sensitive data. If a token set during anonymous usage of the application can later be used to retrieve sensitive user information, the application is vulnerable to session fixation.
6. If the application uses HTTP cookies to transmit session tokens, it may well be vulnerable to cross-site request forgery (XSRF). First, log in to the application. Then confirm that a request made to the application but originating from a page of a different application results in submission of the user’s token. (This submission needs to be made from a window of the same browser process that was used to log in to the target application.) Attempt to identify any sensitive application functions whose parameters an attacker can determine in advance, and exploit this to carry out unauthorized actions within the security context of a target user. See Chapter 13 for more details on how to execute XSRF attacks.

Liberal Cookie Scope

The usual simple summary of how cookies work is that the server issues a cookie using the HTTP response header `Set-cookie`, and the browser then resubmits this cookie in subsequent requests to the same server using the `Cookie` header. In fact, matters are rather more subtle than this.

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`.

NOTE A server cannot specify just any domain using this attribute. First, the domain specified must be either the same domain that the application is running on or a domain that is its parent (either immediately or at some remove). Second, the domain specified cannot be a top-level domain such as `.com` or `.co.uk`, because this would enable a malicious server to set arbitrary cookies on any other domain. If the server violates one of these rules, the browser simply ignores the `Set-cookie` instruction.

If an application sets a cookie's domain scope as unduly liberal, this may expose the application to various security vulnerabilities.

For example, consider a blogging application that allows users to register, log in, write blog posts, and read other people's blogs. The main application is located at the domain `wahh-blogs.com`. When users log in to the application, they receive a session token in a cookie that is scoped to this domain. Each user can create blogs that are accessed via a new subdomain that is prefixed by his username:

```
herman.wahh-blogs.com  
solero.wahh-blogs.com
```

Because cookies are automatically resubmitted to every subdomain within their scope, when a user who is logged in browses the blogs of other users, his session token is submitted with his requests. If blog authors are permitted to place arbitrary JavaScript within their own blogs (as is usually the case in

real-world blog applications), a malicious blogger can steal the session tokens of other users in the same way as is done in a stored cross-site scripting attack (see Chapter 12).

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.

A different version of this vulnerability arises when an application explicitly sets the domain scope of its cookies to a parent domain. For example, suppose that a security-critical application is located at the domain `sensitiveapp.wahh-organization.com`. When it sets cookies, it explicitly liberalizes their domain scope, as follows:

```
Set-cookie: sessionId=12df098ad809a5219; domain=wahh-organization.com
```

The consequence of this is that the sensitive application's session token cookies will be submitted when a user visits *every* subdomain used by `wahh-organization.com`, including:

```
www.wahh-organization.com  
testapp.wahh-organization.com
```

Although these other applications may all belong to the same organization as the sensitive application, it is undesirable for the sensitive application's cookies to be submitted to other applications, for several reasons:

- The personnel responsible for the other applications may have a different level of trust than those responsible for the sensitive application.
- The other applications may contain functionality that enables third parties to obtain the value of cookies submitted to the application, as in the previous blogging example.
- The other applications may not have been subjected to the same security standards or testing as the sensitive application (because they are less important, do not handle sensitive data, or have been created only for test purposes). Many kinds of vulnerability that may exist in those applications (for example, cross-site scripting vulnerabilities) may be irrelevant to the security posture of those applications. But they could enable an external attacker to leverage an insecure application to capture session tokens created by the sensitive application.

NOTE Domain-based segregation of cookies is not as strict as the same-origin policy in general (see Chapter 3). In addition to the issues already described in the handling of hostnames, browsers ignore both the protocol and port number when determining cookie scope. If an application shares a hostname with an untrusted application and relies on a difference in protocol or port number to segregate itself, the more relaxed handling of cookies may undermine this segregation. Any cookies issued by the application will be accessible by the untrusted application that shares its hostname.

HACK STEPS

Review all the cookies issued by the application, and check for any domain attributes used to control the scope of the cookies.

1. If an application explicitly liberalizes its cookies' scope to a parent domain, it may be leaving itself vulnerable to attacks via other web applications.
2. If an application sets its cookies' domain scope to its own domain name (or does not specify a domain attribute), it may still be exposed to applications or functionality accessible via subdomains.

Identify all the possible domain names that will receive the cookies issued by the application. Establish whether any other web application or functionality is accessible via these domain names that you may be able to leverage to obtain the cookies issued to users of the target application.

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.

As with domain-based restrictions on cookie scope, 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.

In contrast to domain-based scoping of cookies, this path-based restriction is much stricter than what is imposed by the same-origin policy. As such, it is almost entirely ineffective if used as a security mechanism to defend against untrusted

applications hosted on the same domain. Client-side code running at one path can open a window or iframe targeting a different path on the same domain and can read from and write to that window without any restrictions. Hence, obtaining a cookie that is scoped to a different path on the same domain is relatively straightforward. See the following paper by Amit Klein for more details:

http://lists.webappsec.org/pipermail/websecurity_lists.webappsec.org/2006-March/000843.html

Securing Session Management

The defensive measures that web applications must take to prevent attacks on their session management mechanisms correspond to the two broad categories of vulnerability that affect those mechanisms. To perform session management in a secure manner, an application must generate its tokens in a robust way and must protect these tokens throughout their life cycle from creation to disposal.

Generate Strong Tokens

The tokens used to reidentify a user between successive requests should be generated in a manner that does not provide any scope for an attacker who obtains a large sample of tokens from the application in the usual way to predict or extrapolate the tokens issued to other users.

The most effective token generation mechanisms are those that:

- Use an extremely large set of possible values
- Contain a strong source of pseudorandomness, ensuring an even and unpredictable spread of tokens across the range of possible values

In principle, any item of arbitrary length and complexity may be guessed using brute force given sufficient time and resources. The objective of designing a mechanism to generate strong tokens is that it should be extremely unlikely that a determined attacker with large amounts of bandwidth and processing resources should be successful in guessing a single valid token within the life span of its validity.

Tokens should consist of nothing more than an identifier used by the server to locate the relevant session object to be used to process the user's request. The token should contain no meaning or structure, either overtly or wrapped in layers of encoding or obfuscation. All data about the session's owner and status should be stored on the server in the session object to which the session token corresponds.

Be careful when selecting a source of randomness. Developers should be aware that the various sources available to them are likely to differ in strength

significantly. Some, like `java.util.Random`, are perfectly useful for many purposes where a source of changing input is required. But they can be extrapolated in both forward and reverse directions with perfect certainty on the basis of a single item of output. Developers should investigate the mathematical properties of the actual algorithms used within different available sources of randomness and should read relevant documentation about the recommended uses of different APIs. In general, if an algorithm is not explicitly described as being cryptographically secure, it should be assumed to be predictable.

NOTE Some high-strength sources of randomness take some time to return the next value in their output sequence because of the steps they take to obtain sufficient entropy (such as from system events). Therefore, they may not deliver values fast enough to generate tokens for some high-volume applications.

In addition to selecting the most robust source of randomness that is feasible, a good practice is to introduce as a source of entropy some information about the individual request for which the token is being generated. This information may not be unique to that request, but it can be effective at mitigating any weaknesses in the core pseudorandom number generator being used. Here are some examples of information that may be incorporated:

- The source IP address and port number from which the request was received
- The `User-Agent` header in the request
- The time of the request in milliseconds

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 at the time of this writing) to produce a manageable fixed-length string that can be used as a token. (Placing the most variable items toward the start of the hash's input maximizes the "avalanche" effect within the hashing algorithm.)

TIP Having chosen an algorithm for generating session tokens, a useful "thought experiment" is to imagine that your source of pseudorandomness is broken and always returns the same value. In this eventuality, would an attacker who obtains a large sample of tokens from the application be able to extrapolate tokens issued to other users? Using the formula described here, in general this is highly unlikely, even with full knowledge of the algorithm used. The source IP, port number, `User-Agent` header, and time of request together generate a vast amount of entropy. And even with full knowledge of these, the attacker will be unable to produce the corresponding token without knowing the secret string used by the server.

Protect Tokens Throughout Their Life Cycle

Now that you've created a robust token whose value cannot be predicted, this token needs to be protected throughout its life cycle from creation to disposal, to ensure that it is not disclosed to anyone other than the user to whom it is issued:

- The token should only be transmitted over HTTPS. Any token transmitted in cleartext should be regarded as tainted — that is, as not providing assurance of the user's identity. If HTTP cookies are being used to transmit tokens, these should be flagged as `secure` to prevent the user's browser from ever transmitting them over HTTP. If feasible, HTTPS should be used for every page of the application, including static content such as help pages, images, and so on. If this is not desired and an HTTP service is still implemented, the application should redirect any requests for sensitive content (including the login page) to the HTTPS service. Static resources such as help pages usually are not sensitive and may be accessed without any authenticated session. Hence, the use of secure cookies can be backed up using cookie scope instructions to prevent tokens from being submitted in requests for these resources.
- Session tokens should never be transmitted in the URL, because this provides a simple vehicle for session fixation attacks and results in tokens appearing in numerous logging mechanisms. In some cases, developers use this technique to implement sessions in browsers that have cookies disabled. However, a better means of achieving this is to use `POST` requests for all navigation and store tokens in a hidden field of an HTML form.
- Logout functionality should be implemented. This should dispose of all session resources held on the server and invalidate the session token.
- Session expiration should be implemented after a suitable period of inactivity (such as 10 minutes). This should result in the same behavior as if the user had explicitly logged out.
- Concurrent logins should be prevented. Each time a user logs in, a different session token should be issued, and any existing session belonging to the user should be disposed of as if she had logged out from it. When this occurs, the old token may be stored for a period of time. Any subsequent requests received using the token should return a security alert to the user stating that the session has been terminated because she logged in from a different location.
- If the application contains any administrative or diagnostic functionality that enables session tokens to be viewed, this functionality should be robustly defended against unauthorized access. In most cases, there is no need for this functionality to display the actual session token. Rather, it should contain sufficient details about the owner of the session for any

support and diagnostic tasks to be performed, without divulging the session token being submitted by the user to identify her session.

- The domain and path scope of an application's session cookies should be set as restrictively as possible. Cookies with overly liberal scope are often generated by poorly configured web application platforms or web servers, rather than by the application developers themselves. No other web applications or untrusted functionality should be accessible via domain names or URL paths that are included within the scope of the application's cookies. Particular attention should be paid to any existing subdomains to the domain name that is used to access the application. In some cases, to ensure that this vulnerability does not arise, it may be necessary to modify the domain- and path-naming scheme employed by the various applications in use within the organization.

Specific measures should be taken to defend the session management mechanism against the variety of attacks that the application's users may find themselves targets of:

- The application's codebase should be rigorously audited to identify and remove any cross-site scripting vulnerabilities (see Chapter 12). Most such vulnerabilities can be exploited to attack session management mechanisms. In particular, stored (or *second-order*) XSS attacks can usually be exploited to defeat every conceivable defense against session misuse and hijacking.
- Arbitrary tokens submitted by users the server does not recognize should not be accepted. The token should be immediately canceled within the browser, and the user should be returned to the application's start page.
- Cross-site request forgery and other session attacks can be made more difficult by requiring two-step confirmation and /or reauthentication before critical actions such as funds transfers are carried out.
- Cross-site request forgery attacks can be defended against by not relying solely on HTTP cookies to transmit session tokens. Using the cookie mechanism introduces the vulnerability because cookies are automatically submitted by the browser regardless of what caused the request to take place. If tokens are always transmitted in a hidden field of an HTML form, an attacker cannot create a form whose submission will cause an unauthorized action unless he already knows the token's value. In this case he can simply perform an easy hijacking attack. Per-page tokens can also help prevent these attacks (see the following section).
- A fresh session should always be created after successful authentication, to mitigate the effects of session fixation attacks. Where an application does not use authentication but does allow sensitive data to be submitted, the threat posed by fixation attacks is harder to address. One possible approach

is to keep the sequence of pages where sensitive data is submitted as short as possible. Then you can create a new session at the first page of this sequence (where necessary, copying from the existing session any required data, such as the contents of a shopping cart). Or you could use per-page tokens (described in the following section) to prevent an attacker who knows the token used in the first page from accessing subsequent pages. Except where strictly necessary, personal data should not be displayed back to the user. Even where this is required (such as a “confirm order” page showing addresses), sensitive items such as credit card numbers and passwords should *never* be displayed back to the user and should always be masked within the source of the application’s response.

Per-Page Tokens

Finer-grained control over sessions can be achieved, and many kinds of session attacks can be made more difficult or impossible, by using per-page tokens in addition to session tokens. Here, a new page token is created every time a user requests an application page (as opposed to an image, for example) and is passed to the client in a cookie or a hidden field of an HTML form. Each time the user makes a request, the page token is validated against the last value issued, in addition to the normal validation of the main session token. In the case of a non-match, the entire session is terminated. Many of the most security-critical web applications on the Internet, such as online banks, employ per-page tokens to provide increased protection for their session management mechanism, as shown in Figure 7-12.

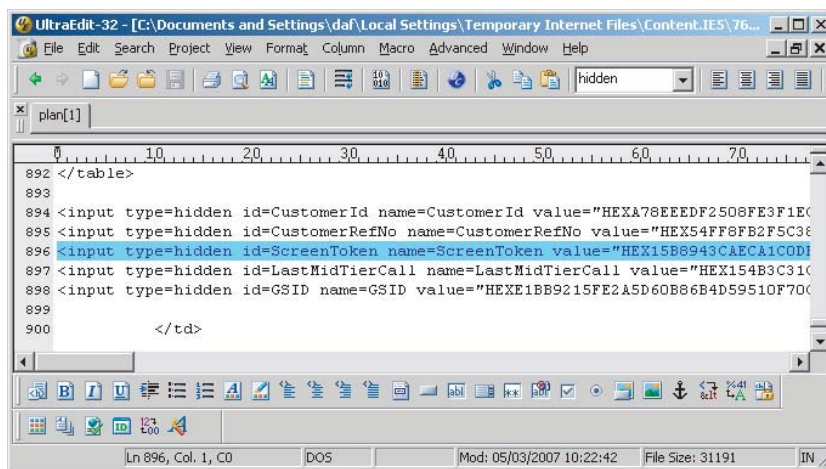


Figure 7-12: Per-page tokens used in a banking application

The use of per-page tokens does impose some restrictions on navigation (for example, on use of the back and forward buttons and multiwindow browsing).

However, it effectively prevents session fixation attacks and ensures that the simultaneous use of a hijacked session by a legitimate user and an attacker will quickly be blocked after both have made a single request. Per-page tokens can also be leveraged to track the user's location and movement through the application. They also can be used to detect attempts to access functions out of a defined sequence, helping protect against certain access control defects (see Chapter 8).

Log, Monitor, and Alert

The application's session management functionality should be closely integrated with its mechanisms for logging, monitoring, and alerting to provide suitable records of anomalous activity and to enable administrators to take defensive actions where necessary:

- The application should monitor requests that contain invalid tokens. Except in the most predictable cases, a successful attack that attempts to guess the tokens issued to other users typically involves issuing large numbers of requests containing invalid tokens, leaving a noticeable mark in the application's logs.
- Brute-force attacks against session tokens are difficult to block altogether, because no particular user account or session can be disabled to stop the attack. One possible action is to block source IP addresses for an amount of time when a number of requests containing invalid tokens have been received. However, this may be ineffective when one user's requests originate from multiple IP addresses (such as AOL users) or when multiple users' requests originate from the same IP address (such as users behind a proxy or firewall performing network address translation).
- Even if brute-force attacks against sessions cannot be effectively prevented in real time, keeping detailed logs and alerting administrators enables them to investigate the attack and take appropriate action where they can.
- Wherever possible, users should be alerted to anomalous events relating to their session, such as concurrent logins or apparent hijacking (detected using per-page tokens). Even though a compromise may already have occurred, this enables the user to check whether any unauthorized actions such as funds transfers have taken place.

Reactive Session Termination

The session management mechanism can be leveraged as a highly effective defense against many kinds of other attacks against the application. Some security-critical applications such as online banking are extremely aggressive in terminating a user's session every time he or she submits an anomalous request.

Examples are any request containing a modified hidden HTML form field or URL query string parameter, any request containing strings associated with SQL injection or cross-site scripting attacks, and any user input that normally would have been blocked by client-side checks such as length restrictions.

Of course, any actual vulnerabilities that may be exploited using such requests need to be addressed at the source. But forcing users to reauthenticate every time they submit an invalid request can slow down the process of probing the application for vulnerabilities by many orders of magnitude, even where automated techniques are employed. If residual vulnerabilities do still exist, they are far less likely to be discovered by anyone in the field.

Where this kind of defense is implemented, it is also recommended that it can be easily switched off for testing purposes. If a legitimate penetration test of the application is slowed down in the same way as a real-world attacker, its effectiveness is dramatically reduced. Also, it is very likely that the presence of the mechanism will result in more vulnerabilities remaining in production code than if the mechanism were absent.

HACK STEPS

If the application you are attacking uses this kind of defensive measure, you may find that probing the application for many kinds of common vulnerabilities is extremely time-consuming. The mind-numbing need to log in after each failed test and renavigate to the point of the application you were looking at would quickly cause you to give up.

In this situation, you can often use automation to tackle the problem. When using Burp Intruder to perform an attack, you can use the Obtain Cookie feature to perform a fresh login before sending each test case, and use the new session token (provided that the login is single-stage). When browsing and probing the application manually, you can use the extensibility features of Burp Proxy via the `IBurpExtender` interface. You can create an extension that detects when the application has performed a forced logout, automatically logs back in to the application, and returns the new session and page to the browser, optionally with a pop-up message to tell you what has occurred. Although this by no means removes the problem, in certain cases it can mitigate it substantially.

Summary

The session management mechanism provides a rich source of potential vulnerabilities for you to target when formulating your attack against an application. Because of its fundamental role in enabling the application to identify the same user across multiple requests, a broken session management function usually

provides the keys to the kingdom. Jumping into other users' sessions is good. Hijacking an administrator's session is even better; typically this enables you to compromise the entire application.

You can expect to encounter a wide range of defects in real-world session management functionality. When bespoke mechanisms are employed, the possible weaknesses and avenues of attack may appear to be endless. The most important lesson to draw from this topic is to be patient and determined. Quite a few session management mechanisms that appear to be robust on first inspection can be found wanting when analyzed closely. Deciphering the method an application uses to generate its sequence of seemingly random tokens may take time and ingenuity. But given the reward, this is usually an investment well worth making.

Questions

Answers can be found at <http://mdsec.net/wahh>.

1. You log in to an application, and the server sets the following cookie:

```
Set-cookie: sessid=amltMjM6MTI0MT0xMTk0ODcwODYz;
```

An hour later, you log in again and receive the following:

```
Set-cookie: sessid=amltMjM6MTI0MT0xMTk0ODc1MTMy;
```

What can you deduce about these cookies?

2. An application employs six-character alphanumeric session tokens and five-character alphanumeric passwords. Both are randomly generated according to an unpredictable algorithm. Which of these is likely to be the more worthwhile target for a brute-force guessing attack? List all the different factors that may be relevant to your decision.
3. You log in to an application at the following URL:

```
https://foo.wahh-app.com/login/home.php
```

and the server sets the following cookie:

```
Set-cookie: sessionId=1498172056438227; domain=foo.wahh-app.com; path=/login; HttpOnly;
```

You then visit a range of other URLs. To which of the following will your browser submit the `sessionId` cookie? (Select all that apply.)

- (a) <https://foo.wahh-app.com/login/myaccount.php>
- (b) <https://bar.wahh-app.com/login>
- (c) <https://staging.foo.wahh-app.com/login/home.php>
- (d) <http://foo.wahh-app.com/login/myaccount.php>