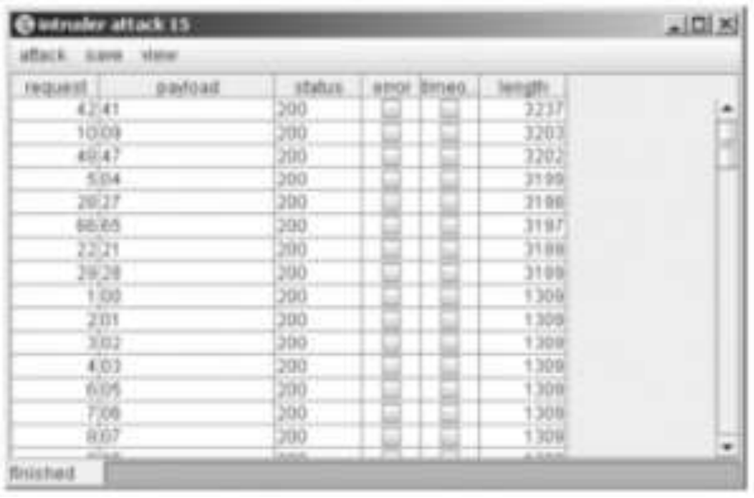
Predictable Tokens:

Some session tokens do not contain meaningful data associating them with a particular user but are nevertheless guessable 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 an amount of trial and error, this will still enable an automated attack to identify large numbers of valid tokens in a relatively short period.

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 in bespoke applications.

When you remotely target a bespoke session management mechanism, the server's capacity may restrict your sample of issued tokens, other users' activity, 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 can eliminate interference caused by other users.

An application may use a simple sequential number as the session token in the simplest and most brazenly vulnerable cases. 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.



request	payload	status	error	timeo	length
42	41	200	☐	☐	3237
10	09	200	☐	☐	3203
48	47	200	☐	☐	3202
5	04	200	☐	☐	3199
28	27	200	☐	☐	3198
66	65	200	☐	☐	3197
22	21	200	☐	☐	3188
29	28	200	☐	☐	3188
1	00	200	☐	☐	1309
2	01	200	☐	☐	1309
3	02	200	☐	☐	1309
4	03	200	☐	☐	1309
6	05	200	☐	☐	1309
7	06	200	☐	☐	1309
8	07	200	☐	☐	1309

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

Figure -1 shows Burp Intruder being used to cycling the last two digits of a sequential session token to find values where the session is still active and can be hijacked. The server's response length is a reliable indicator that a valid session has been found.

In other cases, an application's tokens may contain more elaborate sequences that take some effort to discover. The types of potential variations one 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

Concealed Sequences

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

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

No immediate pattern is discernible; however, a cursory inspection indicates that the tokens may contain Base64-encoded data — in addition to the mixed-case alphabetical 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:

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

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

```
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 for generating session tokens that use the time of generation to input 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 may appear completely 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 many sessions being created per second, a scripted attack may identify large numbers of other users' tokens. For example, 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 trivial to predict. The second number is increasing 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 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 different 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 apparent:

■ The first numeric sequence continues to progress incrementally; however, five values have been skipped since the end of our first sequence. This is presumably because the missing values have been issued to other users who logged into the application in the window between the two tests.

■ The second numeric sequence continues progressing by similar intervals; however, the first value we obtain is a massive 539,578 more significant than the previous value.

This second observation immediately alerts us to the role played by time in generating session tokens. Only five tokens have been issued between the two token-grabbing exercises. However, a period of approximately 10 minutes has also 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, and in a subsequent phase of our testing we perform a code review, which reveals the following token-generation algorithm:

```
String sessId = 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 quickly.
■ We monitor the increments in the first number. When this increases by more than one, 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 issued to them because we possess the tokens issued immediately before and after theirs. However, because we obtain 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.