# PLAY SESSION INJECTION

*By Louis Nyffenegger <Louis@PentesterLab.com>*

# Table of Content

# Introduction

This course details the exploitation of a session injection in the Play framework. This issue can be used to inject arbitrary content inside the session and therefore modify the application logic to escalate privileges.

# About this exercise

## License



# Syntax of this course

The red boxes provide information on mistakes/issues that are likely to happen while testing:

An issue that you may encounter...

The green boxes provide tips and information if you want to go further.

You should probably check...

The blue boxes are "homework": things you can work on once you are done with this exercise:

You should probably work on...

# The web application

Once the system has booted, you can then retrieve the current IP address of the system using the command `ifconfig`:

In this example the IP address is 10.0.2.15.

Throughout the training, the hostname `vulnerable` is used for the vulnerable machine, you can either replace it by the IP address of the machine, or you can just add an entry to your host file with this name and the corresponding IP address. It can be easily done by modifying:

- on Windows, your `C:\Windows\System32\Drivers\etc\hosts` file;
- on Unix/Linux and Mac OS X, your `/etc/hosts` file.

The IP address can change if you restart the system, don't forget to update your hosts file.

# The Play Framework

The Play Framework is a web framework that allows developers to quickly build web application in Java or Scala. The way the code is organised and the URL are mapped are very similar to Ruby-on-Rails.

## Fingerprinting

Here the Play application is deployed on the port 80 and running as root (to be able to bind on the port 80). We can observe that Play is used by looking at the `Server` header:

Running Play or any web server that doesn't drop privileges during its startup is obviously a terrible idea. A vulnerability (directory traversal, RCE, ...) in any application hosted will provide the attacker with full server compromise (access to `/etc/shadow` for directory traversal, running command as `root` for RCE).

Since we can register an account, we will try to create a 'test:test1' account to look into the session's handling.

After registering this account, we receive the following cookie:

```
PLAY_SESSION=dc76c24ea96cdf0009188367583f07bee1126aff-
%00___AT%3A103939fbeba60071e96c5dd505a7916d8b49c9c9%00%00user%3Atest%00
```

We can see that information is stored in clear text in the session: `user%3atest` (`user:test`). Unfortunately, no sensitive information is stored as part of the session.

# Play session

The session mechanism used by Play is really similar to Rack sessions (used by Ruby-on-Rails by default). The content of the session is sent back to the clients instead of being stored server side. To prevent an attacker from tampering his session. A signature of the session is calculated. We will now dive into this mechanism by reviewing the code involved in this bug.

This mechanism is used to easily load-balance requests between multiple servers without keeping a central sessions' pool shared between the multiple servers. By sharing the key used to sign the session, any servers with the key can edit and trust the content of the session.

Since this vulnerability has been discovered (http://www.playframework.com/security/vulnerability/20130806-SessionInjection), the session management has been fully rewritten and does not use this format anymore.

We will now see how the sessions mechanism worked before the vulnerability was reported.

# White Box approach

If we look at how sessions were handled (https://github.com/playframework/play1/blob/1.2.5/framework/src/play/mvc/Scope.java):

```
static Pattern sessionParser =
Pattern.compile("\u0000([^:]*):([^\u0000]*)\u0000");
[...]
static Session restore() {
  try {
    Session session = new Session();
    Http.Cookie cookie = Http.Request.current().cookies.get(COOKIE_PREFIX +
"_SESSION");
    final int duration = Time.parseDuration(COOKIE_EXPIRE) ;
    final long expiration = (duration * 1000l);
    if (cookie != null && Play.started && cookie.value != null &&
!cookie.value.trim().equals("")) {
      String value = cookie.value;
      int firstDashIndex = value.indexOf("-");
      if(firstDashIndex > -1) {
        String sign = value.substring(0, firstDashIndex);
        String data = value.substring(firstDashIndex + 1);
        if (sign.equals(Crypto.sign(data, Play.secretKey.getBytes()))) {
          String sessionData = URLDecoder.decode(data, "utf-8");
          Matcher matcher = sessionParser.matcher(sessionData);
          while (matcher.find()) {
              session.put(matcher.group(1), matcher.group(2));
          }
        }
      }
   [...]
```

First, the code retrieve the cookie used for the session, for example `PLAY_SESSION`. If the cookie is present it will then split it into 2 parts:

- `sign`: the signature.
- `data`: the data.

It will then verify the signature using the method `Crypto.sign` and the secret key `Play.secretKey`.

The `Play.secretKey` is stored in `conf/application.conf`, if you can get access to this file (for example using a directory traversal), you will be able to forge sessions.

We can see that the session's signature is based on HMAC (the code is available in `Crypto.java` https://github.com/playframework/play1/blob/1.2.5/framework/src/play/libs/Crypto.java) as we can see in the code below:

```
public static String sign(String message, byte[] key) {
 if (key.length == 0) {
     return message;
 }

 try {
     Mac mac = Mac.getInstance("HmacSHA1");
```

Using HMAC will prevent attacks like length extension attacks (http://en.wikipedia.org/wiki/Length_extension_attack), as opposed to using simple hash functions (like MD5 for example).

We can also see that the code in Scope.java used a non time-constant string comparison:

```
   if (sign.equals(Crypto.sign(data, Play.secretKey.getBytes()))) {
```

This issue could potentially be exploited to create a valid signature using brute-force by comparing the time used to compare the signature provided to the one generated. It is pretty unlikely to be exploited due to network latency and the computing of the signature but the same issue has been fixed in Ruby-on-Rails in early 2013...

> Also, as with most (all?) session mechanisms that send back a signed cookie to the client, the session cannot be immediately invalidated since an attacker will still be able to use the old session even if a new empty one has been sent to him.

Once the signature is verified, Play parses the session's data using the following mechanism:

```
static Pattern sessionParser =
Pattern.compile("\u0000([^:]*):([^\u0000]*)\u0000");
[...]
  Matcher matcher = sessionParser.matcher(sessionData);
  while (matcher.find()) {
      session.put(matcher.group(1), matcher.group(2));
  }
```

If you have already look in the sessions parsing of few web frameworks, you quickly realise that this mechanism is fairly different to a lot of frameworks. Most framework will rely on known formats (like YAML, JSON, object serialisation) to store session's information. Here the data is only splitted using a regular expression and a loop.

Let's now see how information is added to the server side session:

```
public void put(String key, String value) {
  if (key.contains(":")) {
    throw new IllegalArgumentException("Character ':' is invalid in a
session key.");
  }
  change();
  if (value == null) {
     data.remove(key);
  } else {
     data.put(key, value);
  }
}
```

We can see here that we can't use a key that contains a `:`. However nothing prevent a value from containing a NULL Byte. If an attacker is able to inject NULL Bytes in a value (for example in his/her username), he/she can potentially inject additional variables inside the session (without knowing the secret key used for the signature).

# Black Box approach

In a Black Box approach, you would have to find that the delimiter is an encoded NULL byte and try to inject one in your username to see what happens. By using trial and error, you should be able to find the same bug.

Think of most injection problems as:

| | | | | |
|---|---|---|---|---|
| delimiter1 | keyword1 | delimiter2 | **data1** | delimiter3 |
| delimiter1 | keyword2 | delimiter2 | **data2** | delimiter3 |

If you have control over `data1` and/or `data2`, you need to try to inject the various delimiters and keywords to see if you can trigger unexpected behavior.

# Session Injection

## Details

As we just saw, the data in the session follow this pattern:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| %00 | **key1** | : | **value1** | %00 | %00 | **key2** | : | **value2** | %00 |

Which can be re-arranged as below:

| | | | | |
|---|---|---|---|---|
| %00 | **key1** | : | **value1** | %00 |
| %00 | **key2** | : | **value2** | %00 |

Our goal now will be to inject NULL BYTEs (`%00`) and separator (`%3a`) to add arbitrary keys and values within the session. The session will stay valid since this injection is performed by the server before it signs the data. And the modified session will only trigger unexpected behavior when it gets sent back to the server.

# Exploitation

Since you will probably have to try multiple times before finding the right value, it's probably worth writing a small script to automate the process:

- Register a user.
- Access the website logged as this user.

Here, you get logged in as soon as you register to keep things easy. You could also use QA tools for browser automation (like Selenium) if the registration process had multiple complex steps.

## Login with admin privileges

In this part, we will make the assumption that the application uses a session's variable named `admin` to know if a user is an administrator.

Our goal is to inject a variable `admin` equals to `1` inside the session. Based on what we saw before, we know that we will need to use: `%00admin%3a1%00`. We will also need to properly finish the declaration of the current variable (namely `user`).

The current session looks like:

| | | | | |
|---|---|---|---|---|
| %00 | **___AT** | : | **103939fbeba60071e96c5dd505a7916d8b49c9c9** | %00 |
| %00 | **user** | : | **{INJECTION]** | %00 |

Our goal is to end up with:

| | | | | |
|---|---|---|---|---|
| %00 | **___AT** | : | **103939fbeba60071e96c5dd505a7916d8b49c9c9** | %00 |
| %00 | **user** | : | **test** | **%00** |
| **%00** | **admin** | **:** | **1** | %00 |

The last NULL BYTE comes from the server code so you don't need to end your payload with a NULL byte. By using the payload above, you should be able to create a user with admin privileges.

## Login as another user

In this part, we will try to login as another user: `admin1`. If you remember the method used to parse the session, you see that the latest variable in the session will always overwrite any previous declaration (due to the usage of an HashMap):

```
  // Code used for the parsing
  while (matcher.find()) {
     session.put(matcher.group(1), matcher.group(2));
  }
[...]
  // Session's storage backend
Map<String, String> data = new HashMap<String, String>();

[...]
public void put(String key, String value) {
  [...]
    data.put(key, value);
  [...]
```

We are therefore able to inject another key `user` that will overwrite the one from the application. The following table shows what result you want to achieve:

| | | | | |
|---|---|---|---|---|
| %00 | **___AT** | : | **103939fbeba60071e96c5dd505a7916d8b49c9c9** | %00 |
| %00 | **user** | : | **test** | **%00** |
| **%00** | **user** | **:** | **admin1** | %00 |

# Conclusion

This exercise explained how to exploit a session injection in the Play framework. This bug is pretty interesting since we use the server to create a forged session and we then use it to gain access to administrator privileges or to log in as another user. Once again, re-inventing the wheel can be dangerous, and you should probably rely on known formats if you want to store information. I hope you enjoyed learning with PentesterLab.