# Spot the Security Bugs

## Kun jij de lekken vinden?

Een security code review is een zeer effectieve manier om beveiligingslekken boven water te krijgen. Het legt lekken bloot, die van buitenaf niet of nauwelijks zichtbaar zijn en geeft een goed beeld van het algehele beveiligingsniveau van een applicatie. Is er bijvoorbeeld defensief geprogrammeerd en zijn algemene security best practices meegenomen tijdens het ontwerp en de bouw? Denk hierbij bijvoorbeeld aan de maatregelen, die nodig zijn om veelvoorkomende beveiligingslekken (zoals benoemd in de OWASP Top-10 [1]) te voorkomen.

Als onderdeel van hacktesten worden bij Securify dagelijks code reviews uitgevoerd, die veelal zijn gericht op Java EE en Microsoft .NET applicaties. Populair onder collega's zijn de zogenaamde *spot the bugs*. Zie hier een stukje code met een vernuftig beveiligingslek, kun jij het lek vinden? Dit heeft ons op het idee gebracht om op J-Fall 2015 een *spot the bug challenge* te organiseren.

De missie: zo veel mogelijk beveiligingslekken opspeuren in een applicatie, afkomstig van het gehackte, fictieve bedrijf HuggSoft. De deelnemer, die de meeste en belangrijkste beveiligingslekken wist te ontdekken, ging er vandoor met een Parrot A.R. Drone.

In dit artikel behandelen we een aantal van de belangrijkste lekken, die tijdens de wedstrijd werden gevonden. Voor een compleet overzicht van alle gemelde lekken (meer dan 20!) kun je terecht op onze blogpagina [2,3].

### De briefing

*We from the company HuggSoft have created special HuggDrones® to satisfy our costumers' needs. While the drones are still in their test-phase (blades cutting off fingers, etc), they already got hacked by a hacker named Lamius. Somehow Lamius retrieved the secret key for the user interface, and he's programming all drones to shout at people. Please help us! You'll find the source code in the secure drop zone. Please perform an amazing code review so our system will never get hacked again! We'll reward you with a drone!*

### Een toelichting van HuggSoft

De gehackte applicatie geeft een gebruiker de mogelijkheid om in te loggen in het

```
1   <%@ page import="java.util.Random" %>
2   <%@ page import="com.programmer.userfunctions.*" %>
3   <%@ taglib prefix="spring" uri="http://www.springframework.org/tags" %>
4   <!--
5   These sessions need not be saved server-side!
6   Example of an authorization cookie:
7   amwwgcqpdtxthauepzxbjrropg-testuser
8   -->
9
10
11  <%!
12      userfunctions uf = new userfunctions();
13
14      public Boolean checkLogin(String username, String password)
15      {
16          String loginResult = uf.getLoginSOAPXML(username,password);
17          return loginResult.matches("^.*Authorized.*$");
18      }
19
20      public Boolean verifySession(String sess, String nonce)
21      {
22          String username = sess.split("-")[1];
23          if (createSession(nonce, username).equals(sess))
24              return true;
25          else
26              return false;
27      }
28
29      public String createSession(String nonce, String username)
30      {
31          Long squaredNonce = getSquare(nonce);
32          int UniqueUsernameNumber = getUniqueUsernameNumber(username);
33
34          return generateSecret(squaredNonce + UniqueUsernameNumber) + "-"
35                  + username;
36      }
37
38      public Long getSquare(String nonce)
39      {
40          long n = Long.parseLong(nonce);
41          long squared = 0;
42          for(int i = 0; i != (n * Math.abs(n)); i++)
43          {
44              if (i < 0) i = 0;
45              squared += 1;
46          }
47          return squared;
48      }
49
50      public String generateSecret(Long seed)
51      {
52          Random gen = new Random(seed);
53          String dat = "";
54          for (int i = 0; i <= 25; i++)
55              dat += (char)(gen.nextInt(25) + 97);
56          dat = uf.ROTWithSecret(dat);
57          return dat;
58      }
59
60      public int getUniqueUsernameNumber(String username)
61      {
62          int sum = 0;
63          for (int i = 0; i <= username.length()-1; i++)
64              sum += (int)username.charAt(i);
65          return sum;
66      }
```

*Figuur 1*

achterliggende systeem. Dit kan door middel van een gebruikersnaam, een wachtwoord en een nonce. Als de login succesvol is, wordt er een sessie-token gegenereerd op basis van de nonce en de gebruikersnaam. De gebruikersnaam wordt toegevoegd aan het token. Vervolgens stuurt de browser steeds de sessie en de nonce. Om te verifiëren of de sessie klopt, wordt het token opnieuw berekend met de gebruikersnaam uit de sessie en de nonce in de URL. Superveilig allemaal. Bij een succesvolle login of sessie wordt de gebruiker begroet met diens gegevens. In **Figuur 1** en **2** zie je de betrokken code, die ook online is te vinden [2,3].

## De belangrijkste beveiligingslekken uitgelicht

In de volgende secties worden de belangrijkste lekken behandeld. Mocht je zelf ook nog de uitdaging willen aangaan, dan is dit het moment om te stoppen met lezen en de code onder de loep te nemen. Hoe is het gesteld met jouw security skills?

## Aanvaller kan de login-functionaliteit omzeilen

De code bevat een paar kritieke bypasses. Om te beginnen met regel 87-94. Hier wordt de sessie geverifieerd. Indien de sessie valide is, wordt de gebruiker verwelkomd en worden zijn of haar gegevens getoond op basis van de waarde *username*. De *username* is echter afkomstig uit de URL en staat dus los van de sessie. Elke gebruiker met een valide token zou dus de *username* van een andere gebruiker kunnen opgeven om zo diens gegevens te verkrijgen. De auteur had hier uiteraard de *username* uit de gevalideerde sessie moeten gebruiken.

Een andere potentiele bypass bevindt zich in de *checkLogin()* methode (r14-18). We hebben geen code van de aangevraagde SOAP-service, dus we weten niet hoe deze werkt. Wel is zichtbaar dat, zodra de service een antwoord retourneert waar ergens het woord "*Authorized*" in de body voorkomt, de gebruiker direct wordt ingelogd. Dus ook zodra *loginResult* = "*Not Authorized*", "*User Authorized is unknown*" en "*Exception in method isAuthorized() in file..*" etc. De beveiliging is dus compleet afhankelijk van het gedrag van de externe SOAP-service. Riskant!

Nummer drie: de methode *getUniqueUsernameNumber()* genereert geen uniek nummer. Zodra iemand een gebruikersnaam registreert, waarvoor hetzelfde nummer als een andere gebruikersnaam wordt genereerd (*collision*), dan kan die persoon zijn token gebruiken om in te loggen als de andere gebruiker.

```
67   %>
68
69   <%
70       String username = request.getParameter("user");
71       String password = request.getParameter("pass");
72       String nonce = request.getParameter("nonce");
73       String message = "Login Error";
74       String data = "Userdata";
75       String ses = "";
76
77       try {
78           Cookie[] cookies = request.getCookies();
79           if (cookies != null)
80               for (int i = 0; i < cookies.length; i++)
81                   if (cookies[i].getName().equals("session"))
82                       ses = cookies[i].getValue();
83
84           if (!uf.isNullOrEmpty(nonce) && nonce.matches("([a-zA-Z]+)*"))
85               message = "Nonce must be a number, not " + nonce;
86
87           else if (!uf.isNullOrEmpty(ses) && !uf.isNullOrEmpty(nonce))
88           {
89               if (verifySession(ses, nonce))
90               {
91                   message = "Welcome " + ses.split("-")[1];
92                   data = uf.getUserData(username);
93               }
94           }
95
96           else if (!uf.isNullOrEmpty(username) && !uf.isNullOrEmpty(password) &&
97                    !uf.isNullOrEmpty(nonce))
98           {
99               Cookie c = new Cookie("session",createSession(nonce, username));
100              c.setMaxAge(60*60*60*60);
101              if (checkLogin(username,password))
102              {
103                  response.addCookie(c);
104                  message = "Welcome";
105                  data = uf.getUserData(username);
106              }
107              else
108                  message = "Login error";
109          }
110
111      }
112      catch (Exception e)
113      {
114          message = "Login Error: " + e.getMessage();
115      }
116  %>
117  <html><head><title>User page</title></head><body>
118      <%
119      if (message.contains("Welcome"))
120      {   %>
121          Welcome <spring:message text="<%= username %>" /> <br>
122          <%= data %>
123          <%
124      }
125      else
126      {   %>
127          Error: <%= message %> <br>
128          Please <a href="<%= request.getParameter("returnUrl") %>">return</a> to
129          the login page.
130          <%
131      }   %>
132  </body></html>
```

*Figuur 2*

Een vierde bypass bevindt zich in de volgende code (zie **Figuur 3**).

Zodra je inlogt of als je sessie klopt, dan wordt de string "Welcome" toegevoegd aan *message*. Vervolgens wordt er verderop (r119) vanuit gegaan dat zodra de tekst *Welcome* in *message* staat je bent ingelogd. Stel nu dat iemand *nonce* = "*Welcome*" opgeeft. Dan wordt *message* "*Nonce must be a number, not Welcome*" en kan de check dus worden omzeild. *Userdata* wordt in dit geval overigens

```
82                      ses = cookies[i].getValue();
83
84          if (!uf.isNullOrEmpty(nonce) && nonce.matches("([a-zA-Z]+)*"))
85              message = "Nonce must be a number, not " + nonce;
86
87          else if (!uf.isNullOrEmpty(ses) && !uf.isNullOrEmpty(nonce))
88          {
89              if (verifySession(ses, nonce))
90              {
91                  message = "Welcome " + ses.split("-")[1];
92                  data = uf.getUserData(username);
```

*Figuur 3*

```
120     {   %>
121         Welcome <spring:message text="<%= username %>" /> <br>
122         <%= data %>
123         <%
124     }
```

*Figuur 4*

niet geset, maar alleen de gebruikersnaam, zoals opgegeven in de URL, wordt weergegeven. Zou een aanvaller hier nog iets aan kunnen hebben? We komen hier later op terug.

## Aanvaller kan eigen code uitvoeren op de server

We gaan het nu hebben over Expression Language (EL) injection. JSP EL kan worden gebruikt voor allerlei in-line JSP "magie". Dit kan enorm handig zijn, maar brengt ook risico's met zich mee. Zo kan bijvoorbeeld de volgende constructie worden toegepast (zie **Figuur 4**).

De tag *spring:message* wordt hier gebruikt voor de output van *username*. De inhoud van het attribuut *text* zal door middel van EL worden geëvalueerd en uitgevoerd. In dit geval zal de parameter *username* worden weergegeven.

Bepaalde tags in sommige versies van Spring doen aan zogenaamde *double evaluation* [4]. Dit houdt in dat de inhoud van *username* ook wordt gezien als EL en dus zal worden uitgevoerd! De tag *spring:message* is hier een belangrijk voorbeeld van. Erg handig op zijn tijd, maar ontzettend gevaarlijk. Dit is dan ook de oorzaak van een groot aantal hacks in de afgelopen jaren.

In het laatst beschreven bypass voorbeeld zagen we dat een aanvaller de mogelijkheid heeft om direct naar dit deel (r 121) van de code te springen, met een *username* naar keuze. We kunnen hier dus de *username* gebruiken om EL te injecteren. Dat kan weer worden misbruikt om bijvoorbeeld systeemvariabelen uit te lezen. Wat zou er bijvoorbeeld gebeuren als we de volgende username meegeven: "*${uf.secret}*"? Juist, de waarde van *secret* uit de *userfunctions* zal worden getoond. In sommige gevallen kan EL injection zelfs misbruikt worden om code uit te voeren op de server (Remote Code Execution). In dit geval is het *game over* en kan de server op afstand worden overgenomen. Een voorbeeld van een payload constructie, die kan worden gebruikt om via EL code uit te voeren, zie je in **Listing 1**.

## Aanvaller kan login-tokens kraken

De beveiliging van het login-mechanisme berust op het login-token. Zodra een aanvaller login-tokens van andere gebruikers kan raden of voorspellen, dan heeft hij de sleutel tot het systeem in handen. Hij kan zich dan eenvoudig voordoen en inloggen als iedere gebruiker. Laten we eens kijken of hier ook zaken over het hoofd zijn gezien.

Het token wordt gegenereerd op basis van een *seed*. De *seed* is opgebouwd uit de *nonce* plus het unieke *username* nummer (r34-35).

De *generateSecret()* methode laat zien dat het enige dat een buitenstaander weerhoudt om het token te genereren de waarde is uit *uf.ROTWithSecret(dat)*, zie **Figuur 5**.

De functienaam suggereert dat de waarde wordt "gerotate" met een secret. Als we een gebruikersnaam/nonce combinatie met valide token (van na de rotatie) weten, dan kunnen we het token genereren van *vóór* de rotatie, en rotaten met het token. Dit zou dan het secret terug moeten geven.

```
"${java.lang.Runtime.getCurrentRuntime().exec("..")}"..
```

*Listing 1*

**PLEASE PERFORM AN AMAZING CODE REVIEW SO OUR SYSTEM WILL NEVER GET HACKED AGAIN!**

```
50        public String generateSecret(Long seed)
51        {
52            Random gen = new Random(seed);
53            String dat = "";
54            for (int i = 0; i <= 25; i++)
55                dat += (char)(gen.nextInt(25) + 97);
56            dat = uf.ROTWithSecret(dat);
57            return dat;
58        }
```

*Figuur 5*

Nu blijkt dat er in het HTML-commentaar (r5-7) een token wordt prijsgegeven voor de gebruiker *testuser*. Dat biedt een mooi aanknopingspunt. We weten helaas de *nonce* niet, maar wat we wel weten, is dat de *nonce* nooit groter kan zijn dan $\sqrt{2^{31}}$ (= 46.340).

Als we alle puzzelstukjes verzamelen, blijkt dat deze opvallend vreemde constructie voor het genereren van tokens als volgt misbruikt zou kunnen worden. Door als aanvaller een bestand aan te leggen met alle tokens van *vóór* de rotatie (met username testuser en *nonce* waarden 0 t/m 46.340) en vervolgens al deze tokens terug te rotaten met het verkregen token, wordt een lijst van potentiele secrets verkregen. Door vervolgens een *dictionary attack* uit te voeren op alle potentiele tokens, blijft er één opvallend secret over dat geheel bestaat uit tekst, namelijk *iamgeniusiamgeniusiamgeniu*. Het secret is dus hoogstwaarschijnlijk *iamgenius*. Eenmaal in het bezit van het secret, kan de aanvaller eenvoudig zelf login-tokens aanmaken en inloggen als iedere willekeurige gebruiker. Een stap-voor-stap demonstatie van deze aanval kun je teruglezen in de online write-up [2].

Hiermee zijn de belangrijkste lekken behandeld, die door Lamius misbruikt zouden kunnen zijn om toegang te krijgen tot HuggSoft. Het is duidelijk dat de ontwikkelaar van deze code geen kaas heeft gegeten van applicatiebeveiliging. Naast de behandelde bugs is er overigens nog veel meer niet in de haak. Zie jij nog meer lekken of aandachtspunten?

Zijn de *Denial of Service* (DoS) lekken je bijvoorbeeld al opgevallen? Het blijkt namelijk mogelijk om de applicatie met een paar simpele aanvragen plat te leggen. En de Cross Site Scripting (XSS) lekken? En mocht je er nog geen genoeg van hebben, dan vind je een compleet overzicht van de beveiligingslekken terug op onze site [2].

## Conclusie

De uitzonderlijk lekke code blijkt een variatie aan ontwerp- en implementatiefouten te bevatten. Alhoewel dit voorbeeld natuurlijk wel erg bond is, treffen we vergelijkbare zaken helaas nog maar al ta vaak aan in de praktijk. Zaken die dikwijls eenvoudig en vroegtijdig "gevangen" hadden kunnen worden, indien er gedurende het ontwerp en de bouw was stilgestaan bij softwarebeveiliging. Met name *threat modeling* [6] en tussentijdse code reviews zijn essentieel om vroegtijdig problemen te signaleren/voorkomen en onaangename verassingen achteraf te voorkomen.

We hopen dat je het een leuk en leerzaam artikel hebt gevonden en de nodige inspiratie hebt opgedaan om toe te passen in je eigen omgeving. Mochten je extra zaken zijn opgevallen, die nog niet zijn behandeld, dan heb je wellicht iets nieuws gevonden en horen we dat uiteraard graag van je, zodat we het kunnen toevoegen aan de lijst [5].

En mocht je tijdens je dagelijkse werkzaamheden leuke, vernuftige bugs tegenkomen, dan nemen we ze graag mee ter inspiratie voor de volgende *spot the bug challenge*. Geheel anoniem uiteraard. De nieuwe challenge zal later dit jaar weer worden aangekondigd via Twitter (*@securifybv*).

Happy (security) bug hunting!

De Drone werd gewonnen door Patrick Holthuizen. ■

**David Vaartjes** is software security specialist bij Securify en de auteur van dit artikel.

**Sipke Mellema** is software security specialist bij Securify en de auteur van de challenge.

## REFERENTIES

[1] **https://www.owasp.org/index.php/OWASP_Top_Ten_Cheat_Sheet**
[2] **https://www.securify.nl/stb2015**
[3] **https://www.securify.nl/dronemission**
[4] **http://bit.ly/ExpressionLanguageInjection**
[5] **spotthebug@securify.nl**
[6] **https://www.owasp.org/index.php/Application_Threat_Modeling**