Chez les Barbus – Java & Sécurité

Romain Pelisse
[Sustain Developer @Red Hat]

François Le Droff
[Achitecte @Adobe]

« Chez les Barbus - Java & Sécurité » c'est le titre de la conférence donnée par François Le Droff et Romain Pelisse à l'occasion de Devoxx France 2015. Cet article propose d'en reprendre le contenu de manière plus didactique, et adaptée à ce nouveau support, sous la forme d'un dialogue.

A l'aide d'un cas pratique, cet article décrit, sous la forme d'un dialogue entre deux experts, les deux auteurs, l'ensemble des problématiques liées à la mise en place de la sécurité autour d'une application « web » - aussi simple et peu critique soit elle.

François: Je suis en train de monter une sympathique petite application en interne, mais je m'inquiète un peu de l'aspect sécurité, je ne me sens pas armé pour un audit ! On en est encore au stade du prototype, mon architecture n'est pas sèche et mes développements sont encore en cours... Je n'ai même pas encore *staffé* toute l'équipe... Bref, comment m'y prendre ?

Romain: Les audits *adhoc*, qui arrivent à la fin des développements, quand justement les personnes clés du projet sont parties, ça ne marche pas. La sécurité se fait en équipe, dans la durée, tout au long du projet. Cependant, il est vrai qu'il est difficile de trouver un « bon point de départ ».

1 Threat Modelling

François: Comme « point de départ » je te propose une méthode que je trouve très intéressante. Elle est poussée par **Microsoft** et **OWASP**. Elle s'appelle le *« Threat Modeling »*. Elle donne une bonne vision d'ensemble; elle permet d'une part d'identifier les menaces, et d'autre part, de définir la priorité des points à adresser à l'aide de deux acronymes: **STRIDE** et **DREAD**.

Romain: Commençons alors par STRIDE, qu'est-ce que ça veut dire ?

François: L'acronyme STRIDE reflète une catégorisation des attaques possibles selon les types d'exploitation qui leurs sont associés (**S** pour *Spoofing*, **T** pour *Tampering*, **R** pour *Repudiation*, **I** pour *Information disclosure*, **D** pour *Denial of service*, **E** pour *Elevation of privilege*). À chaque catégorie correspond des mesures à mettre en place.

Romain: Restons didactiques et brefs, sinon, on va y passer tout l'article ! Prenons donc juste un exemple concret: que signifie le **R** de *Repudiation* ?

François: La menace, le risque de *Repudiation* est celui de voir ses utilisateurs contester les transactions faites avec leurs identifiants et depuis leur machine. Une des solutions pour réduire cette menace est de mettre en place un *« audit trail »* sur chacun de tes tiers applicatifs.

Romain: En effet, si on doit imputer à quelqu'un une transaction malhonnête, c'est bien de pouvoir le prouver sans l'ombre d'un doute. Passons au second acronyme: DREAD.

François: Avec l'acronyme DREAD, le *Threat Modeling* propose un système de classification pour comparer, qualifier et définir l'importance des risques associés à chaque menace. C'est une mesure du risque de sécurité. Elle se calcule sur la base de la moyenne des valeurs assignées à chacune des catégories suivantes: *Damage potential*, *Reproductibility*, *Exploitability*, *Affected users*, *Discoverability*.

Romain: Le résultat final: un schéma d'architecture orientée sécurité offrant une vue d'ensemble des menaces pesant sur ton système.

François: C'est aussi un bon outil d'aide à la décision… malheureusement souvent ignoré par les

développeurs.

Romain: Triste réalité, car cette méthode semble être une arme parfaite pour prendre conscience des menaces qui pèsent sur un système. Et ceci, dans un langage et un format qu'un *manager* (même récalcitrant) pourra comprendre, et réutiliser pour convaincre sa propre hiérarchie.

François: Et comme par un effet miroir, ce même inventaire peut permettre au *manager* de vérifier que ses développeurs n'ont pas (par naïveté, par manque de connaissance ou simplement par manque de temps) négligé l'aspect sécurité.

Romain: Vaste sujet que le *Threat modeling*. Maintenant, que nous avons une méthodologie pour avoir un bon point de départ, regardons donc ta petite application.

Notre cas d'étude

François: C'est une application Web très simple. Elle n'est pas conçue pour spéculer sur les marchés financiers, ni pour aider au marketing ou toute autre surveillance généralisée. Elle n'est donc pas très *big data*, tu n'y trouveras ni *hadoop*, ni *data lake*, ni *machine learning*.

Romain: C'est pas très bon pour ton aura, rassure-moi, tu utilises au moins **Docker**, non ?

François: Non. C'est juste une bonne vieille application de gestion, une plomberie et une API http faite en **Java** avec une interface en **html**. Et pour gagner du temps je l'ai générée avec **jHipster [1]**.

Romain: C'est quoi jHipster ? Un énième *framework* MVC ? Un truc qui **SpringBoot** pour lancer un container **Scala** écrit en **JRuby** qui exécute des greffons **Groovy** ?

François: Presque! C'est un générateur d'applications Web basé sur **Yeoman**; l'application générée s'appuie sur une architecture assez simple et au goût du jour: Java avec SpringBoot pour la partie serveur, et **css**, html, **JavaScript** et **AngularJS** pour la partie cliente.

Romain: Très bien, mais du point de vue sécurité, tout ceci ne semble pas apporter grand-chose...

François: Si, sous le capot tu trouves **Spring Security [2]**, et ce *framework* Java offre une bonne base de travail pour non seulement gérer l'authentification, les autorisations et les rôles, mais aussi pour contrer des attaques assez « classiques » dans le monde Web.

Romain: Oui, comme les attaques de type *man in the middle*.

François: Oui, en mettant en place **HSTS [3]**, qui indique au client que seul le protocole https sera utilisé.

Romain: Oui, mais encore...

Francois: Tu y trouveras également des stratégies pour éviter le *click-jacking* et les attaques XSS et les attaques de type *Cross site scripting forgery*, pour lesquelles il fournit une solution à base de *CSRF Token* **[4]**.

Romain: Je vois : Spring Security te fournit donc plein d'acronymes en anglais pour te la péter... Et sinon niveau audit ?

François: Il fournit aussi une bonne trame pour mettre en place l'audit et les logs relatifs à la sécurité.

Romain: OK, tu me l'as vendu !

2 Firewall

François: Mais tout ça, au fond, n'est peut-être pas si utile: nos serveurs sont à l'abri, dans un VLAN cloisonné en intranet, derrière une belle ribambelle de *firewalls*.

Romain: Et alors ? Quel rapport ? Un firewall n'a jamais servi à sécuriser une infrastructure, ça se saurait... Un *firewall*, ça sert juste à faire de la qualité réseau ; en gros, à épargner le CPU du système: à ignorer chaque trame réseau correspondant à un port « non utilisé ».

François: Les *firewalls* réduiront donc le trafic « inutile » et donc potentiellement « nuisible », non ?

Romain: Oui, sauf que généralement quand tu as un service qui tourne sur un port, c'est que tu en as besoin, donc du coup, le port est ouvert. Se contenter de démarrer un *firewall* pour bloquer les ports inutilisés ne suffit pas !

François: Qu'est-ce qu'on fait des ports ouverts alors ?

Romain: On en vérifie l'usage ! Si tu as ouvert un port pour envoyer des mails, tu ne laisseras passer que du SMTP. Mais attention, une politique agressive de fermeture systématique des ports ne donne qu'une illusion de sécurité...

Francois: ... et casse les pieds à tout le monde dans la société...

Romain: On est bien d'accord. La bonne solution est d'implémenter un filtrage applicatif, et vérifier ainsi que le contenu de l'échange correspond au protocole utilisé. Certains *firewalls* savent le faire, mais ce filtrage sera souvent mis en place à coup de *reverse proxy*.

François: Force est de constater que les équipes de développement ont rarement la main sur la configuration du *firewall*, souvent sous la responsabilité exclusive des équipes d'infrastructures. De plus, ces équipes dédiées aux configurations de *firewalls* n'auront jamais une connaissance suffisante de la solution pour mettre en place son filtrage applicatif de manière efficace et pérenne. Je serai donc ravi de mettre en place un *reverse proxy*. Par où je commence ?

3 Reverse Proxy

Romain: Une solution simple est de mettre un serveur Web en frontal, et de s'en servir comme d'un *reverse proxy* (RP). Pour faire simple, tu diriges tout le trafic à travers ton serveur « Web » frontal: ce serveur frontal vérifiera que les requêtes sont conformes aux attentes de l'application, avant de les faire suivre au serveur d'application.

François: Tu pourras ainsi te servir de ton *reverse proxy* pour mettre en place la validation de requêtes http, pour ajouter et valider des *tokens CSRF*, ou encore pour implémenter du *throttling* ou du *rate limiting*... Mais ce *reverse proxy*, c'est comme tout le reste, on peut tout autant me le « hacker », non ?

Romain: Oui, mais plus difficilement. Dans les faits, la plupart des « hacks » se basent sur une manipulation de la réaction du serveur distant. On abuse ce dernier, en lui envoyant des requêtes qu'il accepte, mais qui l'amènent à avoir un comportement invalide. Or le *reverse proxy* n'interagit pas directement avec ces requêtes. Il se contente d'en analyser le contenu et de, le cas échéant, ne pas les transmettre au serveur cible. Il reste possible d'abuser son mécanisme d'analyse, mais c'est une fenêtre de tir beaucoup moins grande qu'avant. En outre, comme il n'y a pas d'interaction directe entre le *reverse proxy* et l'attaquant, il n'est pas aisé pour l'attaquant de réaliser que ses requêtes malicieuses ne seront jamais transmises et interprétées par le serveur cible.

François: Pour résumer Romain, on peut dire qu'une des clés d'une bonne sécurité applicative sur une application « Web », c'est la mise en place d'un filtrage applicatif adapté. Si on se contente de simples *firewalls*, c'est un peu comme disposer d'une frontière sans douanier.

4 Données et chiffrement

Romain: On filtre, on nettoie mais on veut aussi recevoir et envoyer des données sur ce fameux réseau, non ?

François: Bien sûr et je ne veux surtout pas que ces données soient compromises. Mon application fournit un beau *mashup* de données, c'est un simple *« employee productivity tool »*, mais, mine de rien, voilà les informations qui y transitent:

- l'adresse, le numéro de téléphone des employés – soit du **PII** (*Personally Identifiable Information*), et çeci toute société est tenue de le garder confidentiel ;

- des infos sur les employés, comme leurs salaires – et ça, c'est aussi confidentiel, mais aussi super sensible ;

- Et enfin des informations internes sur des marchés en cours de négociation – c'est non seulement des informations « internes » mais en plus d'accès restreint !

Romain: Ce qui veut dire que tu ne peux rien faire circuler en clair. Mais ce n'est pas surprenant, c'est le lot commun de la plupart des applications d'entreprise.

François: Oui, il va falloir chiffrer de bout en bout: chiffrer les flux de données qui passent dans les câbles, mais aussi les données au repos, sur les disques.

Romain : Comment as-tu mis ça en place ?

5 Chiffrer le front

François: Alors, commençons par le front : on prévoit évidemment de tout servir en https.

Romain: SSL c'est *secure*, mais faire ça correctement c'est tout un art.

François: Oui, garantir une implémentation robuste du protocole https demande une surveillance et une veille techno constante. Même les géants du Web doivent gérer des situations de crise : faire face à des choix d'algorithmes de chiffrement vieillissants et donc vulnérables, faire face à des vols de certificats, ou encore à la propagation de vrais-faux certificats : certificats créés par des attaquants, associés à leurs domaines ou sous domaines et qui sont « trustés » par la plupart des navigateurs.

6 Chiffrer le back

Romain: Admettons que l'échange entre le serveur et ses clients soit correctement chiffré, il faut maintenant s'assurer que la donnée reste confidentielle, et donc chiffrée quand elle circule (entre ton serveur d'application et ta base de données) et quand elle est au repos, *persistée* sur un disque.

François: C'est fait, mais ce fut laborieux. Pour la base de données, nous avions choisi **MongoDB**; et à l'époque, le support SSL n'était pas fourni dans les distributions MongoDB ; il a donc fallu reconstruire MongoDB depuis les sources et *patcher* l'authentification par certificat (imposé par le transport SSL dans MongoDB) dans le *stack* « Spring-data ».

Romain: Bien, mais quid du chiffrage au repos ?

François: Là, j'hésite encore. On pourrait chiffrer le volume, mais je crains un impact sur les performances et une complexité dans l'administration de la base de données.

Romain: Reste l'option du chiffrage applicatif. C'est l'application, elle-même, qui va chiffrer ses données. Elle sera seule à pouvoir les déchiffrer.

François: Oui, du coup l'administration de MongoDB redevient plus facile, mais ça implique une charge de développement et une gestion d'un secret supplémentaire. De plus, si tu utilises un bon chiffrage (donc un chiffrage asymétrique) tu ne pourras faire ni recherche ni d'indexation sur les champs chiffrés.

Romain: Il n'y a pas de solution miracle. Comme toujours, c'est une question de compromis. Maintenant, l'avantage aussi du chiffrage applicatif, c'est aussi d'avoir une meilleure granularité. Si l'application est compromise, seules ses données pourront être exploitables par l'attaquant. Les données des autres applications resteront illisibles (du moins quelque temps).

7 Authentification

Romain: Je constate que jHipster propose par défaut sa propre base de données d'utilisateur. C'est pratique pour développer un prototype, mais alors par contre, c'est inenvisageable pour la production.

François: Oui, c'est une véritable plaie, non seulement pour la sécurité mais également pour l'expérience utilisateur. En tant qu'utilisateur, il est assez insupportable de devoir créer un énième nom d'utilisateur et mot de passe aux contraintes absurdes.

Romain: Surtout que la plupart des entreprises ont des solutions de gestions d'identité en place.

François: Oui, c'est notre but : notre serveur d'application se contentera de n'être qu'un fournisseur de service, l'authentification sera déléguée à un fournisseur d'identité mutualisé (s'appuyant sur le standard **SAML**) et l'autorisation à un serveur implémentant le standard **oAuth2**.

8 SAML

Romain: Voilà encore une pluie d'acronymes. Commençons par SAML - qu'est-ce que c'est exactement ?

François: C'est un standard, un protocole de sécurité, il permet de mettre en place l'authentification unique (en anglais *Single Sign-On* ou **SSO**) sur le Web.

Romain: Et si je comprends bien, une fois identifié sur un site, tu peux te connecter à d'autres sites, avec le même navigateur, sans avoir à entrer à nouveau tes identifiants. C'est effectivement bien mieux en termes d'expérience utilisateur. Est-ce compliqué à mettre en place ?

François: Pas tant que ça, pour démarrer je te conseille le site *« SSO Circle »* **[5]**, il t'aidera à tester ta solution.

Romain: Comment ça marche ? Qu'est-ce que je dois coder ? Est-ce qu'implémenter SAML est à la portée de n'importe quel développeur Java ?

François: Oui, et comme souvent, la seule complexité réside dans la configuration. Tu dois récupérer les métadonnées du fournisseur d'identité (en anglais *Identity Provider* ou **IdP**) et les transmettre au moteur de ton client SAML. Ces métadonnées contiennent certificats et URLs associés à la redirection en vue d'une authentification ou tout au moins d'une vérification d'identité.

Romain: Et de ton côté, j'imagine aussi que ton appli fournit quelques métadonnées supplémentaires à ton IdP, telles que des certificats aussi et d'autres URLs à associer à la redirection en vue d'une identification (*login*) et d'une déconnection (*logout*).

François: Exactement, tu as tout compris. Regardons l'enchaînement des échanges sur ce diagramme :

Romain: Donc si je me connecte à une page, à une ressource protégée, sans avoir été préalablement identifié, le fournisseur d'identité va retourner un *« Not authorized »* et me refuser l'accès à la ressource.

Francois: Oui, et de là tu vas devoir prouver ton identité auprès de ce fournisseur. Une fois ceci effectué, ce dernier retourne vers mon application, et si tu es probablement identifié, mon application te donnera accès à la ressource.

Romain: Enfin, j'imagine que l'authentification n'est pas le seul facteur, je n'ai peut-être pas le droit d'accès à cette ressource en particulier.

Francois: Oui, bien sûr, lors du retour du fournisseur d'identité, une enveloppe SAML est transmise, son contenu te permettra de déduire le rôle que tu associeras à l'utilisateur, et donc de lui donner ou non l'accès à un service ou à une ressource.

Romain: Mais alors dis-moi, ton super outil jHipster, il supporte le SAML ?

François: Non, mais il vient avec Spring Security qui lui le supporte, quelques lignes de code suffisent.

Romain: Cette authentification unique doit être l'objet de notre attention, elle se doit d'être robuste et fiable.

François: Oui, et c'est pour ça qu'on a aussi implémenté, dans notre serveur d'identité, une authentification à deux étapes **[6]**.

Romain: Excellent idée car c'est vraiment devenu essentiel de nos jours... Mais pour aborder ce sujet il faudrait un autre article !

Références

[1] Jhipster : **https://jhipster.github.io/**

[2] Spring Security : **http://projects.spring.io/spring-security/**

[3] HTTP Strict Transport Security (HSTS) : **https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security**

[4] Cross-site request forgery (CSRF) : **https://en.wikipedia.org/wiki/Cross-site_request_forgery/**

[5] Le site SSO Circle : **http://www.ssocircle.com**

[6] Authentification à deux étapes : **https://en.wikipedia.org/wiki/Two-factor_authentication**