Sécurité des réseaux et du Web
8INF135

Rapport de projet de session du groupe 3

présenté à
M. Valère Plantevin

Par :
Julien Haegman ()
Mathieu Blackburn (BLAM 0304 9208)
Sébastien Tremblay (TRES 1408 8101)

Le lundi 18 décembre 2017

UQAC
Université du Québec
à Chicoutimi

Présentation du projet

Ce projet reprend en partie le travail effectué pour le Travail Pratique #3 (TP3) du cours. Dans les grandes lignes, il s'agit d'une adaptation de ce travail, dans laquelle le modèle de sécurité a été amélioré. Ce projet est divisé en 3 applications :

1) La première est une application web de gestion d'authentification implémentant le protocole OAuth2. Cette application Java se nomme « **GOASP** » (pour Group3 OAuth2 Security Provider).

2) La seconde est une modification de l'API REST, nommée « **RESTodo** », qui avait été développée pour le TP3. Elle a été modifiée pour déléguer la gestion des utilisateurs à **GOASP**. Elle est également développée en Java.

3) La dernière, une application web Angular JS de type "Single Page Application", nommée simplement « **RESTodo FrontEnd** » est une adaptation du FrontEnd développée au départ pour le TP3. Il s'agit d'une unique page web basée exclusivement sur les technologies HTML, CSS et JavaScript (utilisant les frameworks Angular JS, JQuery et Bootstrap). Il est à noter que pour cette raison, le code source de l'application est complètement accessible au public, mais que la sécurité n'en est pas compromise pour autant.

Analyse et implémentation du protocole OAuth2

Pour comprendre le protocole OAuth2, sur lequel repose le serveur d'authentification **GOASP**, il faut d'abord définir les 4 intervenants impliqués dans le processus.

Intervenants :

1) L'application : Il s'agit de l'application (web ou autre) utilisée par un utilisateur et qui désire accéder à un ou plusieurs services protégés par le serveur d'authentification. Dans notre cas il s'agit de l'application **RESTodo FrontEnd**, une application web « Single Page Application ».
2) Le service : Il s'agit d'une application proposant des services (ex : une API REST), enregistrée auprès du serveur d'authentification à qui elle délègue l'authentification de ses utilisateurs. Dans notre cas, il s'agit de l'API REST **RESTodo**.
3) L'utilisateur. C'est la personne physique qui désire utiliser une application. L'humain qui entrera ses informations d'authentification, qui approuvera les requêtes d'accès aux services, etc.
4) Le serveur d'authentification : Il est responsable de gérer l'authentification de l'utilisateur (création de compte, login), les demandes d'approbation de l'utilisateur pour laisser une application accéder à un service, etc. Dans notre cas, il s'agit de l'application **GOASP**.

Résumé

En résumé, l'accès à une ressource protégée d'un service, par un utilisateur via une application se fait en 3 grandes étapes :

1) L'application demande l'approbation de l'utilisateur pour accéder au service. Si l'utilisateur accepte, un code d'autorisation est retourné à l'application.
2) L'application envoi le code d'autorisation au serveur d'authentification. Le serveur retourne un

jeton d'accès.

3) L'application demande une ressource à un service en utilisant le jeton d'accès comme méthode d'authentification. Le service retourne la ressource demandée (ou effectue l'action demandée).

Requête de privilèges et authentification

La première partie du fonctionnement d'OAuth2 consiste à obtenir un jeton d'accès pour que l'application ait l'autorisation nécessaire pour accéder aux services dont il a besoin. Cette demande de jeton se fait en 7 étapes (qui correspondent au étapes 1 et 2 du résumé de la section précédente) :

1) L'utilisateur accède à une application et demande à se connecter en utilisant un service d'authentification de type OAuth 2 (**GOASP**, Google, Facebook, etc).
2) L'application redirige l'utilisateur vers une page d'approbation du serveur d'authentification en précisant les privilèges qu'il désire obtenir.
3) Le serveur d'authentification procède à l'identification de l'utilisateur (login). C'est également à cette étape que l'utilisateur peut se créer un compte utilisateur s'il n'en possède pas déjà un. Il est également possible que l'utilisateur soit déjà connecté s'il avait préalablement accédé à un autre (ou au même) service. Dans ce cas, le serveur d'authentification ne fait rien et passe directement à la prochaine étape.
4) Le serveur d'authentification affiche à l'utilisateur les informations de la requête d'approbation (l'application demandant l'accès aux services ainsi que les privilèges demandés) et lui propose d'accepter ou de refuser d'accorder les privilèges.
5) En cas d'approbation, le serveur d'authentification génère un code d'autorisation qu'il retourne à l'application aillant demandée des privilèges d'accès. En cas de refus, elle lui retourne plutôt un code d'erreur.
6) L'application récupère le code d'autorisation et l'utilise pour effectuer une requête AJAX de jeton (token) d'accès auprès du serveur d'authentification.
7) Le serveur d'authentification s'assure que le code d'autorisation reçu est en règle et si c'est le cas, génère un jeton d'accès qu'il retourne à l'application.

Détails et spécificités d'implémentation

À l'étape 1, l'application **RESTodo FrontEnd** génère un code aléatoire (nommé l'état) qui est stocké dans un cookie pour comparaison ultérieure, et qui est envoyé au serveur **GOASP**.

Aux étapes 2 et 3, **GOASP** vérifie si un utilisateur est déjà connecté en testant la présence et la validité d'un cookie de session. Si une session est ouverte, il propose d'emblée la page d'approbation des privilèges demandés. Sinon, le serveur redirige l'utilisateur vers la page de login où celui-ci pourra se connecter ou encore se créer un compte s'il n'en a pas déjà un. Après connexion, **GOASP** crée un cookie de session pour qu'à la prochaine demande d'approbation de privilège, l'utilisateur soit déjà connecté et n'ait plus qu'à accepter ou refuser la demande. Il redirige ensuite l'utilisateur vers la page d'approbation des privilèges.

À l'étape 5, si l'utilisateur accepte la demande de privilèges, le serveur redirige l'utilisateur vers l'application **RESTodo FrontEnd** en lui passant 2 paramètres : l'état (le même que l'application a fourni à l'étape 1) et le code d'autorisation unique généré aléatoirement. À noter que le code d'autorisation n'est valide que 5 minutes, ce qui est amplement suffisant puisque l'application effectuera immédiatement une demande de jeton d'accès (le tout ne devrait prendre que quelques

secondes). Si par contre l'utilisateur refuse la demande, le serveur redirige l'utilisateur vers l'application en lui retournant un code d'erreur « access_denied ».

À l'étape 6, **RESTodo FrontEnd** commence par comparer l'état qui lui a été passé en paramètre à celui stocké dans son cookie pour s'assurer que le code d'autorisation est authentique. Il supprime ensuite le cookie qui est désormais inutile. (À noter que par soucis de sécurité, ce cookie est supprimé au prochain chargement de la page, qu'il ait reçu un code d'autorisation ou non. La séquence d'appels légitime fera en sorte qu'après une demande de privilèges, le prochain chargement de la page de **RESTodo FrontEnd** contiendra soit un code d'autorisation, soit un code d'erreur dans l'URL.)

À l'étape 7, **GOASP** génère un jeton d'accès. Il est à noter ici que le protocole OAuth2 ne prévoit pas le format du jeton d'accès alors pour des questions pratiques (et pour un usage future abandonné en cours de route, voir section « Accès au service »), nous avons utilisé un jeton JWT dans lequel nous avons intégré certaines informations utiles, dont le nom d'utilisateur **GOASP** et l'adresse courriel. L'application **RESTodo FrontEnd** utilise le nom d'utilisateur inséré au jeton JWT pour l'afficher dans la barre d'entête. Le service **RESTodo** l'utilise conjointement avec l'identifiant du serveur d'authentification (la chaîne « goasp ») comme identifiant unique de l'utilisateur connecté. Par exemple, pour un utilisateur **GOASP** enregistré sous le nom d'utilisateur « User1 », le service **RESTodo** utilisera comme identifiant unique d'utilisateur la chaîne « goasp_User1 ». Nul besoin ainsi de gérer une table d'utilisateurs du côté du service en plus de la gérer du côté du serveur d'authentification.

Accès au service

Le protocole OAuth2 prévoit que le service qui reçoit un jeton d'accès DOIT valider le jeton avant d'autoriser l'accès à la ressource. Cependant, la il ne prévoit pas de quelle façon il doit le faire. En effet, la spécification précise ceci :

> *The resource server MUST validate the access token and ensure that it has not expired and that its scope covers the requested resource. The methods used by the resource server to validate the access token (as well as any error responses) are beyond the scope of this specification but generally involve an interaction or coordination between the resource server and the authorization server.*

La première approche qui avait été retenue pour valider le jeton d'accès qui, rappelons-le, est en fait un jeton JWT, était d'utiliser l'approche habituelle (refaire le hash et comparer les hashs). Cette idée supposait que le service **RESTodo** avait accès à la clé secrète de l'utilisateur, ce qui était possible alors que **RESTodo** et **GOASP** était une seule et même application partageant une même base de données. Cependant, après découpage des deux applications, cette approche n'était plus viable, et était de toute façon irréaliste. En situation réelle, le service n'a pas accès aux données secrètes de l'utilisateur comme son sel cryptographique utilisé pour le hash du jeton puisque ces informations sont conservées de façon confidentielle du côté du serveur d'authentification. C'est pourquoi nous avons finalement utilisé l'approche recommandée par la spécification d'OAuth2. Lorsque le service **RESTodo** reçoit une demande d'accès à une ressource protégée (tout l'API en fait), le service commence par faire une requête HTTPS au serveur **GOASP** pour lui demander de valider le jeton.

Sécurité

Plusieurs éléments ont été mis en place pour assurer la sécurité de tout le système.

OAuth2

La mise en place d'une authentification par le protocole OAuth2 nous assure que l'utilisateur a explicitement donné son accord pour que l'application accède au service demandé. En effet, l'accès à une ressource protégée d'un service requière un jeton d'accès. Ce jeton ne peut être obtenu qu'à partir d'un code d'autorisation. Et finalement, un code d'autorisation ne peut être généré qu'après que l'utilisateur connecté au serveur OAuth2 n'ait cliqué sur le bouton pour accepter la demande d'accès aux ressources (d'autres mesures ont été prises pour s'assurer qu'il s'agit d'un utilisateur authentique, voir section « Protection contre les attaques CSRF »).

À noter qu'il aurait été plus sécuritaire d'utiliser une librairie OAuth2 préexistante, comme Light OAuth2, mais pour le côté formateur de l'exercice, nous avons préféré tout développer à partir de zéro.

Protection contre l'injection SQL

Les applications interagissant directement avec une base de données (**RESTodo** et **GOASP**) utilisent un ORM (ORMLite) pour effectuer les requêtes auprès de la base de données. Bien que certaines données saisies par l'utilisateur soient explicitement validées, nous déléguons essentiellement le travail à l'ORM de veiller à construire correctement les requêtes SQL pour éviter les attaques d'injection SQL.

Protection contre les attaques CSRF

Le système développé n'était d'emblée que très peu vulnérable aux attaques CSRF.

D'abord, les requêtes GET ne provoquent aucune modification de données. L'application **RESTodo FrontEnd** réagit aux requêtes GET auxquelles un code d'authentification a été passé (en déclenchant une requête AJAX HTTPS au serveur **GOASP** pour demander un jeton d'accès), mais ne réagira que si un état est également passé et que si celui-ci correspond à celui généré aléatoirement lors d'une demande de connexion légitime. Une attaque exploitant cette fonctionnalité n'est pas possible sans connaître d'abord l'état généré aléatoirement par l'application ainsi qu'un code d'autorisation légitime généré par le serveur **GOASP**.

De plus, les attaques CSRF exploitent le fait qu'un utilisateur soit déjà authentifié à un service pour effectuer des actions en son nom. Le service **RESTodo**, de par sa nature d'API REST utilisant un jeton comme authentification, n'est pas sensible à ce genre d'attaques. En effet, un site malicieux tentant d'insérer un lien où d'effectuer un POST sur une URL de cet API doit passer un entête « Authorization : Bearer token » dans la requête pour réussir son attaque. Contrairement à un cookie de session, le navigateur ne le fera pas automatiquement et cet entête doit être inséré « manuellement » par le code de l'application connaissant le jeton et effectuant l'appel légitime à l'API. Un site malicieux doit donc le jeton d'authentification pour l'insérer lui-même à la requête. S'il connait ce jeton, ce n'est plus une attaque CSRF.

GOASP, quant à lui, utilise le cookie de session pour maintenir sa connexion d'une requête d'autorisation à l'autre, ce qui le rend susceptible à ce genre d'attaque. Seule la page d'approbation des

demandes de privilèges est vulnérable puisque c'est la seule de ce serveur à nécessiter une session active. Pour cette raison, un jeton de sécurité généré aléatoirement à chaque affichage de la page est inséré dans le formulaire demandant l'approbation ou le rejet de la demande. Avant de prendre en considération la réponse de l'utilisateur, le serveur s'assure que le jeton de sécurité correspond au dernier jeton généré pour cet utilisateur (et qu'il est toujours valide).

Protection contre les attaques XSS

Les applications **GOASP** et **RESTodo** filtrent systématiquement toutes les entrées de toutes les requêtes, que ce soit les données soumises dans l'URL pour les requêtes GET (la partie après le « ? ») ou le corps de la requête pour les autres méthodes de requêtes. Le mot « script » est systématiquement banni de toutes les entrées. De manière similaires, toutes les sorties sont également filtrées pour bannir le mot « script » des sections « non-contrôlées » de la réponse. En effet, pour les pages HTML, le framework utilise un système de modèle de page à substitution de tags (Stsp pour Simple Tag Substitution Page) pour les parties variables du modèle. Lors de la validation de la sortie, le validateur considère le contenu des tags comme des sections « non-contrôlées », mais le reste du modèle (qui peut inclure des balises « <script> » légitimes) est considéré comme contrôlé puisque fourni par le développeur de l'application.

GOASP valide rigoureusement certaines entrées utilisateurs comme le nom d'usager, l'adresse courriel et le mot de passe lors de l'enregistrement, mais sinon, notre système de modèle de page nous permet de nous assurer que toute portion « non-contrôlée » du modèle est échappée correctement (ex : « & » est remplacé par « & », « < » par « < », « > » par « > »), et ne provoquera pas d'injection de code HTML malicieux.

Modèle de sécurité

Le modèle de sécurité des applications **GOASP** et **RESTodo** en est un en « pipeline ». Chaque requête passe par une suite d'étapes décrites ci-dessous :

1) Le serveur web « nginx » s'assure que la connexion est sécurisée. Si le client tente une connexion sur le port HTTP (80), celui-ci redirige le client vers la même adresse, mais sur le port sécurisé HTTPS (443).
2) Le serveur web achemine la requête vers l'application web correspondante.
3) L'application commence par déterminer le « Router » à utiliser selon l'URL demandée. Si l'URL ne correspond à aucune route enregistrée, l'application transfert automatiquement le contrôle à un Router « NotFound » qui ne sert qu'à retourner une erreur 404 « page not found ».
4) La même URL peut être accédée via différentes méthodes HTTP (GET, POST, DELETE, etc.) alors le Router correspondant à l'URL commence par chercher un gestionnaire de route (« Handler ») correspondant à la méthode demandée. Si aucun gestionnaire n'est enregistré pour cette méthode, une erreur 405 « bad request » est retournée.
5) Avant d'appeler le gestionnaire de route lui-même, tous ses filtres de pré-traitement sont appelés un à un, jusqu'à ce que l'un d'eux échoue. Si l'un échoue, le traitement se termine (on passe à l'étape 8), sinon le traitement se poursuit à la prochaine étape. Plusieurs filtres de pré-traitements existent et ne sont pas nécessairement utilisés par tous les gestionnaires de routes et se divisent en 2 catégories :
    a. Filtres d'autorisation :
        i. Session **GOASP** requise : Filtre utilisé dans l'application **GOASP** pour s'assurer

qu'une session **GOASP** est ouverte (cookie de session présent et valide). Si ce n'est pas le cas, ce filtre redirige l'utilisateur vers la page de login.

ii. Jeton d'accès **GOASP** requis : Ce filtre est utilisé par le service **RESTodo** pour s'assurer qu'un jeton d'accès valide a été fourni. Il commence par s'assurer qu'un entête « Authorization : Bearer token » est présent. Si oui, il effectue la validation du jeton auprès de l'application **GOASP**. Si une de ces conditions n'est pas remplie, il retourne un code d'erreur 401 « unauthorized ».

b. Filtres de contenu :

i. Modèle interdit dans une requête : Ce filtre vérifie la partie paramètres de l'URL (tout ce qui suit le « ? » dans l'URL) pour les requêtes GET, sinon le corps de la requête pour toutes les autres méthodes, et cherche un modèle (expression régulière) correspondant à un modèle interdit. Ce filtre est systématiquement appliqué à toutes les requêtes, pour toutes les routes, en interdisant le pattern « .*[Ss][Cc][Rr][Ii][Pp][Tt].* », c'est-à-dire le mot « script », non sensible à la casse, peu importe où dans la requête. C'est une façon de limiter des attaques XSS potentielles.

ii. Corps de requête JSON : Utilisé par **RESTodo**, ce filtre s'assure que le corps de la requête correspond à un objet JSON valide, et qu'il correspond à un modèle en particulier. Par exemple, les API pour créer et modifier des todos s'attendent à recevoir un objet JSON avec 2 propriétés : content et done.

iii. Paramètres de requêtes obligatoire : Utilisé par **GOASP** pour s'assurer que des paramètres GET ou POST obligatoires ont été passés à la requête. Par exemple, un POST sur la page du login doit obligatoirement contenir les champs « username », « password » et « submit ».

6) Appel du gestionnaire de route. Si tous les filtres de pré-traitement ont réussi, le gestionnaire de route est appelé et peut au besoin effectuer des validations supplémentaires et retourner des erreurs en conséquence (ex : Todo à modifier n'existe pas). Sinon, le gestionnaire de route construit la réponse et la conserve en mémoire.
7) Les filtres de post-traitement sont appelés un à un jusqu'à ce que l'un échoue. Dans les fait, il n'existe qu'un seul filtre post-traitement :
    a. Modèle interdit dans la réponse : Tel que décrit dans la section « protection contre les attaques XSS », ce filtre valide toutes les sections dites « non-contrôlées » de la réponse, c'est-à-dire toute portion de la réponse générée par du code et provenant potentiellement de la base de données. Comme pour le filtre « Modèle interdit dans la requête », un filtre interdisant le mot « script » dans toute section non contrôlée de la réponse est systématiquement appliqué à toutes les requêtes traitées. Si le filtre détecte un modèle interdit, il remplace la réponse par une réponse d'erreur 403 « forbidden ».
8) La dernière étape consiste à envoyer au client la réponse générée, que ce soit une réponse de succès ou d'erreur. En effet, tel que précisé, les réponses générées avant cette étape sont seulement stockées en mémoire en attendant d'être passées par tous les filtres de post-traitement. Ce n'est qu'après avoir été traitée par tous les filtres nécessaires que la réponse finale est enfin envoyée à l'utilisateur.

Protection des données

Toute opération sur les données (CRUD) se fait par le biais de l'ORM « ORMLite ». Ceci nous aide à nous protéger contre les injectons SQL, garantissant ainsi l'intégrité des données et de la base de données elle-même, le tout en simplifiant la maintenance du code.