Rapport Project - VV

https://github.com/kevin35ledy/Fuzzing-Swagger

Pierre-Henri COLLIN
Kévin LEDY

I. Introduction

Swagger est un méta-modèle de service REST. Il permet de développer des API de services selon un standard. Le méta-modèle est défini en json ou en yaml et permet d'implémenter différents programmes :

- Des templates de code serveur selon différents langages et frameworks
- Une partie cliente compatible avec la partie serveur
- De la documentation interactive

Malheureusement, ce type d'API a des limites. La principale réside dans la responsabilité des utilisateurs de cette interface d'implémenter les fonctions et services proposés dans l'API. Cela peut entraîner la présence de failles. Un moyen efficace de le mettre en lumière est le Fuzzing.

Le Fuzzing est une méthode de test visant à générer des tests avec des données aléatoires. Ce type de test est souvent utilisé pour détecter des bugs à corriger dans les applications. La génération de ce type de test est relativement simple, et ne nécessite pas le fait de connaître le système que l'on teste. En sortie, les tests rendront l'erreur ou la faille trouvée, ainsi que leur origine.

Ce projet consiste donc à implémenter un programme qui va tester une API Swagger. Nous avons décidé de nous concentrer sur la correspondance entre le code de réponse du serveur attendu et le code de réponse reçu. Pour la génération de valeurs, nous avons essayé de générer des incohérences de types, des bufferOverflow sur les opérations de type GET. Nous avons également essayé de trouver des failles sur les méthodes de type POST.

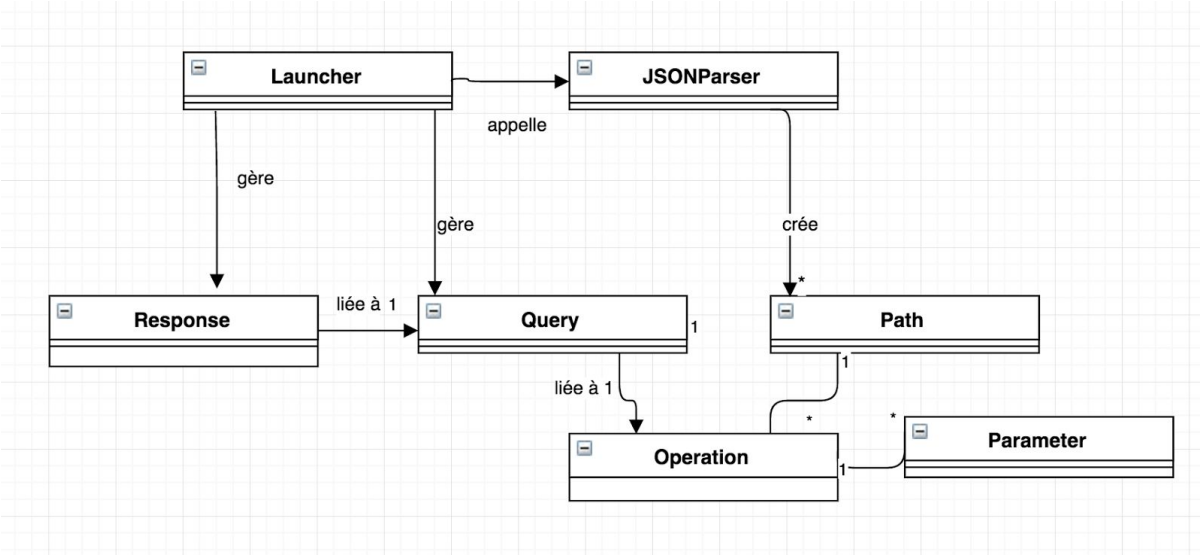
En terme d'évaluation, nous avons appliqué ce fuzzer sur une api disponible à l'adresse suivante : http://petstore.swagger.io/ qui est une application représentant un site e-commerce d'animaux de compagnie. On a ici trois interfaces principales : les animaux, la gestion des devis, et les utilisateurs.

II. Solution

La première chose à faire était de nous familiariser avec les APIs Swagger. Nous avons donc généré notre propre API, qui se contentait de faire un hello-world. Ensuite, nous avons déterminé les tâches à effectuer pour mener à bien ce projet, et le programme à implémenter. Nous avons donc choisi d'implémenter notre programme en Java.

Pour commencer, nous avons utilisé un librairie qui permet de parser le Swagger.json et de le faire correspondre avec nos objets. Nous traitons ainsi chaque url.

Nous pourrions résumer l'architecture du projet par ce schéma :



Pour chaque Path de l'API Swagger, nous avons des Operations. Qu'elles soient de type GET, POST, DELETE ou PUT, chaque Operation possède 0,1 ou plusieurs Parameter.

Ce sont ces même paramètres que nous générons.

Le Launcher demande au JSONParser de parser le fichier swagger.json, cela génère nos Paths, Operations, et Parameters.

Ensuite, le Launcher va générer des Queries selon le type de l'Operation. Chaque fuzzing test est représenté par une Query.

Une fois toutes les requêtes générées, le Launcher va exécuter ces Queries et récupérer les informations provenant du serveur dans un objet Response.

Enfin, le Launcher crée un fichier csv regroupant toutes les informations qui peuvent intéresser l'utilisateur, comme la comparaison du code attendu et reçu, les commentaires d'erreurs reçus, l'url testée avec les valeurs testées, etc.

Pour la génération des requêtes pour les méthodes GET :
Nous nous occupons seulement des paramètres obligatoires. Nous observons leur type.

- Si c'est un int : on teste la valeur vide, 0, négative, une chaîne de caractères, et des nombres de plus en plus grands.
- Si c'est une string : on teste la chaîne vide, des caractères spéciaux, une chaîne très longue.

Pour la génération des requêtes pour les méthodes POST :
Ici nous regardons à la fois les paramètres obligatoires dans l'url, comme par exemple l'id dans /petstore/{petId}, mais également les paramètres dans le body, et donc à insérer en json.

- Pour les paramètres obligatoires, nous regardons le type : integer, on met une valeur arbitraire, pour faire correspondre les tests d'après (ex: le POST pour ajouter un pet, et le POST qui se sert de ce pet)
- Pour les paramètres à ajouter dans le json, on regarde le type et on génère une valeur aléatoire.

Pour la génération des requêtes pour les méthodes DELETE : ici nous nous intéressons uniquement aux paramètres obligatoires qui apparaissent dans le path. Pour ces paramètres, nous testons différentes valeurs dans le cas d'un entier et d'une chaîne de caractère.

Pour la génération des requêtes pour les méthodes PUT : nous agissons de façon similaire aux méthodes POST en nous attardant sur les paramètres obligatoires dans l'url et les paramètres présents dans le body.

Concernant la documentation, nous avons rédigé un Readme expliquant le projet, comment utiliser cet outil de fuzzing et comment l'adapter.

Parsing du swagger yaml vers json

Parseur yaml -> json `brew install swagger-codegen` Se placer dans le dossier du projet swagger `cd api/swagger/` on génère le fichier json `swagger-codegen generate -i swagger.yaml -l swagger`

Fuzzer

Le but principal du projet consiste à fuser une api Swagger. Par conséquent nous avons trouvé une api représentant un service e-commerce d'animaux de compagnie, vous pourrez trouver cette API à l'adresse suivante : http://petstore.swagger.io/

Ainsi pour installer notre projet, il faut télécharger les sources, ajouter les jars :

`commons-lang3-3.5` , `httpclient-4.5.2` , `httpmime-4.5.2` , `jackson-annotations-2.8.5` , `jackson-core-2.8.5` , `jackson-databind-2.8.5` , `jackson-dataformat-yaml-2.8.5` , `json-20160810` , `slf4j-api-1.7.21` , `swagger-core-1.5.10` , `swagger-models-1.5.10` , `swagger-parser-1.0.23`

Et runner la classe Launcher.

Celle-ci va générer et exécuter les requêtes pour fuser l'api swagger du petstore.

Le Launcher fait appel au JSONParser qui va récupérer le swagger.json (méta-modèle de l'api), à l'adresse suivante : http://petstore.swagger.io/v2/swagger.json

Ensuite, celui-ci parse le json pour fabriquer des Paths avec des Opérations de type GET, POST, PUT, DELETE, et leurs paramètres.

Les résultats des tests sont enregistrés dans un fichier csv.

Pour tester une autre api swagger, vous devez modifier l'url du swagger.json dans le JSONParser, ainsi que l'adresse ou vont les requêtes dans le Launcher.

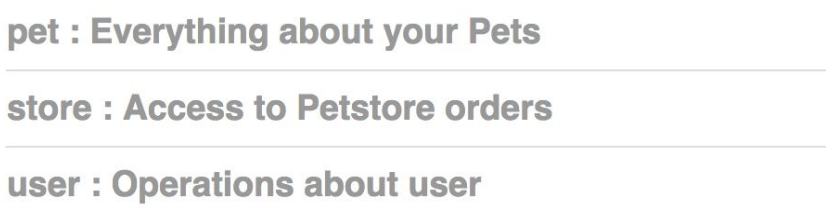
Nous avons également réalisé des tests sur notre programme, dans les classes LauncherTest et TestModel.

LauncherTest s'occupe de regarder si on a bien un code de réponse et donc que le serveur nous répond bien. Le TestModel, lui teste le parsing du JSON.

III. Evaluation

L'Api que nous testons est très complète, car elle fait des opérations de tous types. Elle décrit l'interface d'un service e-commerce d'animaux de compagnie. L'utilisateur peut s'inscrire, se logger, réaliser un devis, ajouter des animaux de compagnie les mettre à jour, etc.

Donc 3 principaux types d'interfaces :



Pour chaque chemin, on a différentes opérations de définies. Par exemple, pour le chemin pet/ on trouve :

Nous savons par expérience que notre programme n'est pas optimisé, par manque de temps. Que ce soit en terme de complexité, avec pour chaque Path, plusieurs Opérations et pour chaque Opérations, plusieurs Paramètres, nous avons une complexité de N^3.
Donc pour un passage à l'échelle, cela devient compliqué.

Pour exécuter ce programme qui effectue environ 55 requêtes en 10 secondes. Donc il y a du progrès à faire.

IV. Discussion

A ce jour, nous avons réussi à remplir notre modèle de données à partir d'une url d'entrée en utilisant la librairie JsonParser. Nous générons des requêtes pour tous les types d'opération présents dans le swagger récupéré en entrée. Pour chaque opération nous analysons les paramètres requis et nous testons différentes valeurs. Nous affichons les résultats des requêtes dans un fichier csv.

Nous avons rencontré plusieurs problèmes au cours de ce projet :

- Tout d'abord, il a fallu comprendre la technologie swagger et analyser la spécification afin de savoir quelles informations nous seront utiles pour nos tests à venir. La spécification s'est avérée être très dense et très diversifié notamment au niveau des paramètres. En effet, un paramètre peut se trouver à différents endroits (path, body, etc) et être de plusieurs types (string, integer, array, object, etc). Autant de cas particulier à gérer au moment de la génération de requêtes.
- En particulier les requêtes DELETE, POST et PUT se sont avérés difficiles à générer à cause des paramètres de type "body" qui correspondent à des objets et ceux-ci sont pas toujours évidents à modéliser comparés aux paramètres de type "path" pour les opérations GET.

Notre solution peut être améliorée sur plusieurs points :

- Au niveau de la génération de requêtes, nous gérons pour l'instant un nombre restreints de paramètres. Par exemple, nous testons les paramètres requis mais nous laissons de côté certains paramètres optionnels. Nous ne gérons pas non plus tous les types de paramètres (par exemple les paramètres de type form-data).
- Au niveau de l'affichage des résultats, nous pouvons nous améliorer sur la présentation en créant par exemple une interface web dynamique qui permet de visualiser clairement les résultats. Un fichier csv fonctionne bien mais n'est pas forcément très user-friendly.

V. Conclusion

Grâce à ce projet, nous avons pu découvrir le framework Swagger et son environnement. Nous avons également pu nous familiariser avec l'envoi de requête http en java et l'analyse de réponses http. A travers les tests d'une API, nous nous sommes rendus compte de l'importance des tests et nous avons constaté que celle-ci peut vite s'avérer être défaillante. Nous avons également utilisé des outils comme JUnit afin de réaliser des tests unitaires sur les méthodes utilisées dans notre programme. Il est primordial de tester le programme qui teste et ce projet nous en a donné la preuve.