Fuzzer

Projet pour le cours Python pour la Sécurité.

Authors	email
Maroua Mesbahi	maroua.mesbahi@epita.fr
Melissa Li	melissa.li@epita.fr

Lien du github : https://github.com/mmesba85/Fuzzer

Sommaire

Introduction

Nous avons choisi comme projet d'implémenter un fuzzer multi protocole.
Dans cette introduction, nous allons présenter ce qu'est un fuzzer et les différents types de fuzzers.

Qu'est ce que le fuzzing ?

Le fuzzing est une technique pour tester des logiciels. Elle est utilisée pour identifier les vulnérabilités potentielles dans un programme.

L'idée est d'injecter des données dans les entrées d'un programme. Si le programme échoue (par exemple en plantant ou en générant une erreur), alors il y a des défauts à corriger.

Presque tous les logiciels attendant une entrée peuvent être fuzzés (navigateurs, application visualisant ou modifiant des fichiers, kernels, la plupart des APIs, ...).

Le grand avantage du fuzzing est que l'écriture de tests est extrêmement simple, ne demande aucune connaissance du fonctionnement du système et permet de trouver des vulnérabilités facilement.

Le fuzzing peut-être divisé en six étapes différentes :

1. Identifier la cible
2. Identifier les entrées
3. Générer les données fuzzées (Il existe 2 méthodes qui seront décrites plus bas)
4. Exécuter les donnée fuzzées
5. Observer les erreurs chez la cible
6. Déterminer la vulnérabilité

Mutation Based Fuzzing

Aussi appelé *dump fuzzing* est la plus rapide à mettre en place, mais avec cette technique il nous est plus compliqué de trouver des bugs profond dans l'application. C'est simple, on va partir d'une donnée valide puis de manière aléatoire aller la modifier. L'exemple le plus courant est celui du protocole HTTP ;
Celui-ci attend une requête de ce type :

GET /index.html HTTP/1.0

Dans notre script on va aléatoirement augmenter la répétition de certains caractères et tester les différentes combinaisons possibles exemple :

GEEEEEEEEET /index.html HTTP/1.0
GET ///////index.html HTTP/1.0

Et ainsi de suite, jusqu'à ce qu'on révèle une mauvaise gestion de la taille d'un des paramètres de la commande.

Generation Based Fuzzing

Aussi appelé *smart fuzzing*, contrairement au *dump fuzzing* on ne va pas partir d'une donnée valide, mais on va analyser ce qu'attend l'application pour ensuite modifier de manière intelligente la donnée et essayer de révéler un bug.

Si on reprend l'exemple du protocole HTTP, on va se référer à la norme RFC d'une commande pour en créer une valide mais malformée. Son plus gros problème est qu'elle demande beaucoup de travail et d'analyse.

Exemple 1 :
Imaginons que la cible soit un logiciel propriétaire qui n'a pas de documentation sur les différents types de données qu'il accepte en entrée. Que faire ?
Avant de passer au fuzzing, il faut analyser au maximum les données qui entrent, et ainsi appliquer des modifications logiques sur ces données et pouvoir générer une erreur dans l'application.

Exemple 2 :
Un fichier contenant 18 champs obligatoires ou optionnels alors que le template en comporte que 4, le *smart fuzzing* ne générera pas les champs non présents. Il va seulement modifier les champs présents.

Worker

Le *worker* envoie les données à la cible et detecte les comportements anormaux.

Exemple :
Executer un binaire qui est compilé en C (enclin à avoir des problèmes de getion de la mémoire). Dans ce cas, un bug peut etre facilement détecté.

Logger

Le *logger* conserve toutes les traces de bugs trouvés et leurs cas de test.
Logging peut faciliter l'analyse des bugs grâce aux traces. Conserver les cas de tests permet de reproduire les bugs.

Avantages et inconvénients

Fuzzing peut etre très utile, mais ce n'est pas la solution miracle. Voici quelques avantages et inconvénients :

Avantages

- Il fournit des résultats avec peu d'efforts : une fois que le fuzzer est en marche, il est autonaume et cherche des bugs avec très peu d'interactions utilisateur.
- Il révèle les bugs qui ont été oubliés lors d'un audit manuel
- Il fournit une image globale de la robustesse du programme cible

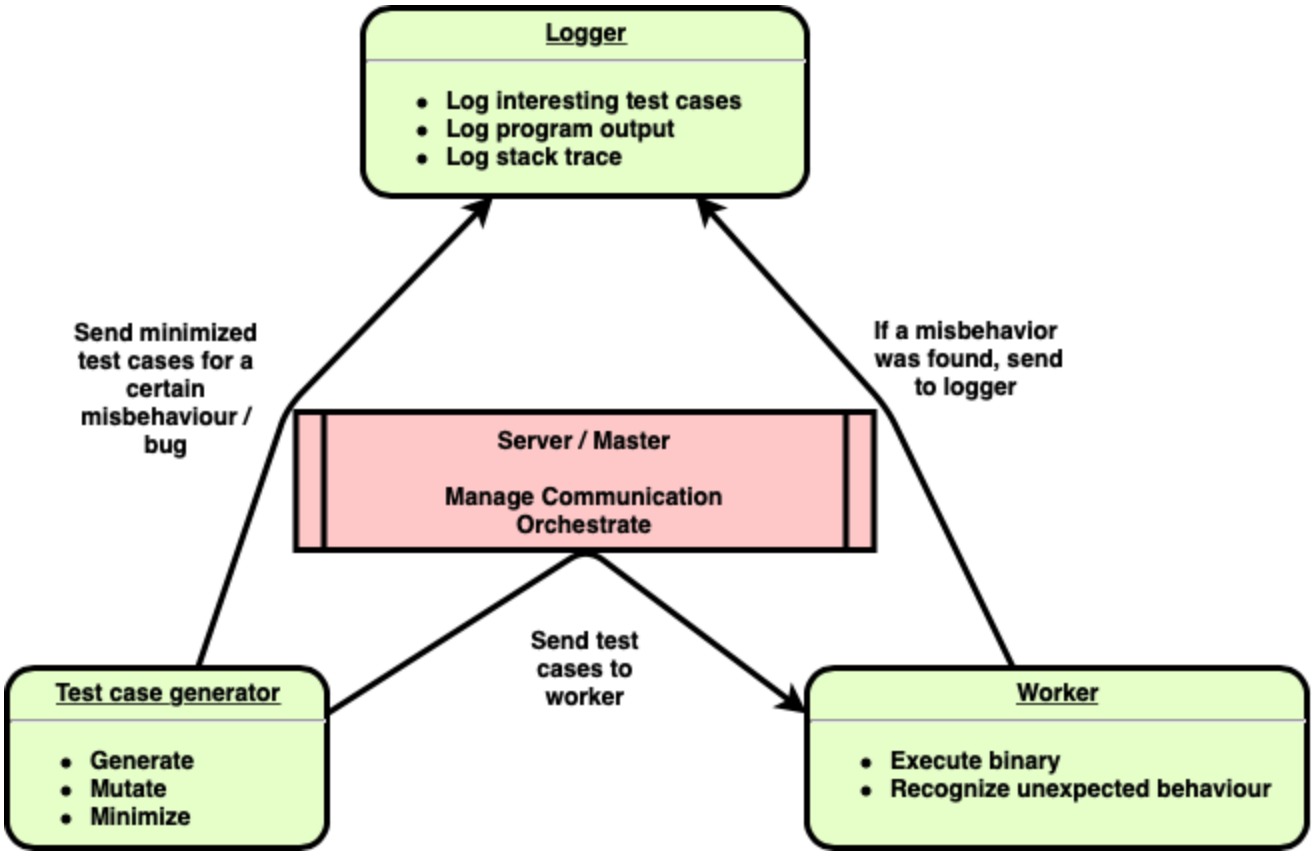
Inconvénients

- Ne va pas trouver tous les bugs : fuzzing peut louper des bugs, ceux qui ne font pas crasher le programme.
- Les tests cases qui font crasher le programme peuvent être difficiles à analyser : fuzzer ne donne pas de données sur la manière dont le programme fonctionne en interne.
- Les programmes prennant des inputs difficiles peuvent exiger plus de travail pour produire un fuzzer assez intelligent pour pouvoir couvrir tout le code

Fonctionnement de l'outil

Notre outil est un fuzzer multi protocole. Il est constitué de plusieurs modules:

- **Generator.py** : Génère, construit et mute les inputs
- **Worker.py** : Envoie les requêtes et traîte les réponses.
- **Fuzzer.py** : Main.



Protocoles

Les protocoles que nous avons exploité :

- http
- ftp

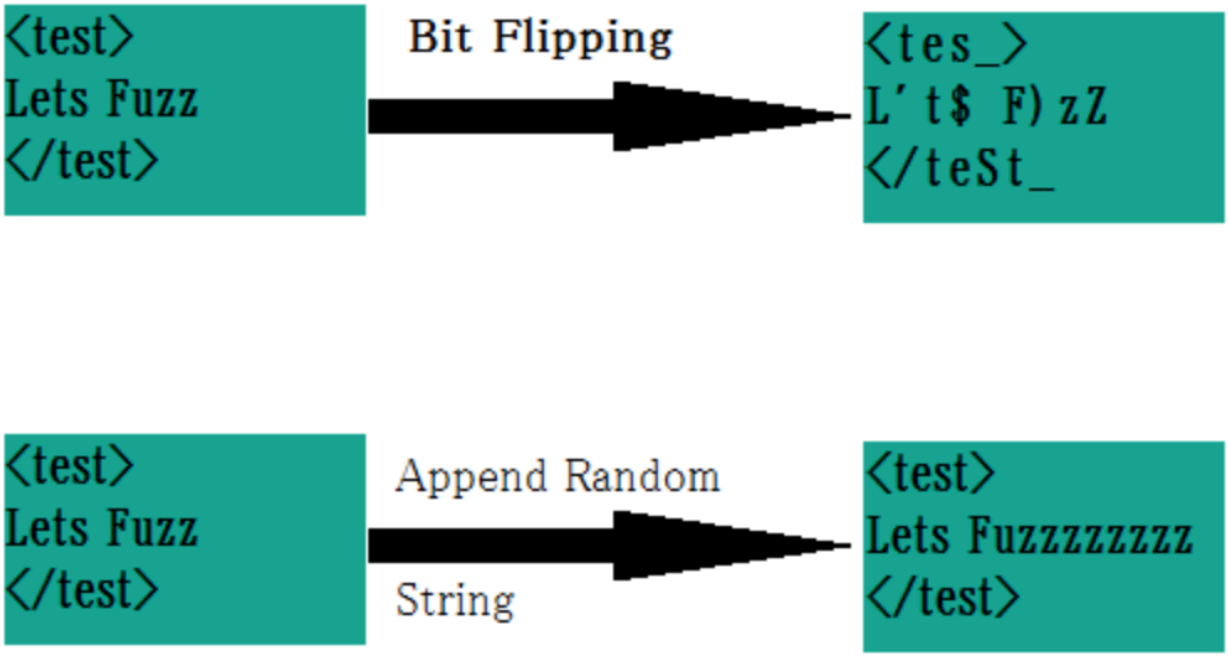
Input

- Génération de chaîne par taille
- Entrée utilisateur: Fichier de configuration (JSON)

Fuzzer types

Mutation

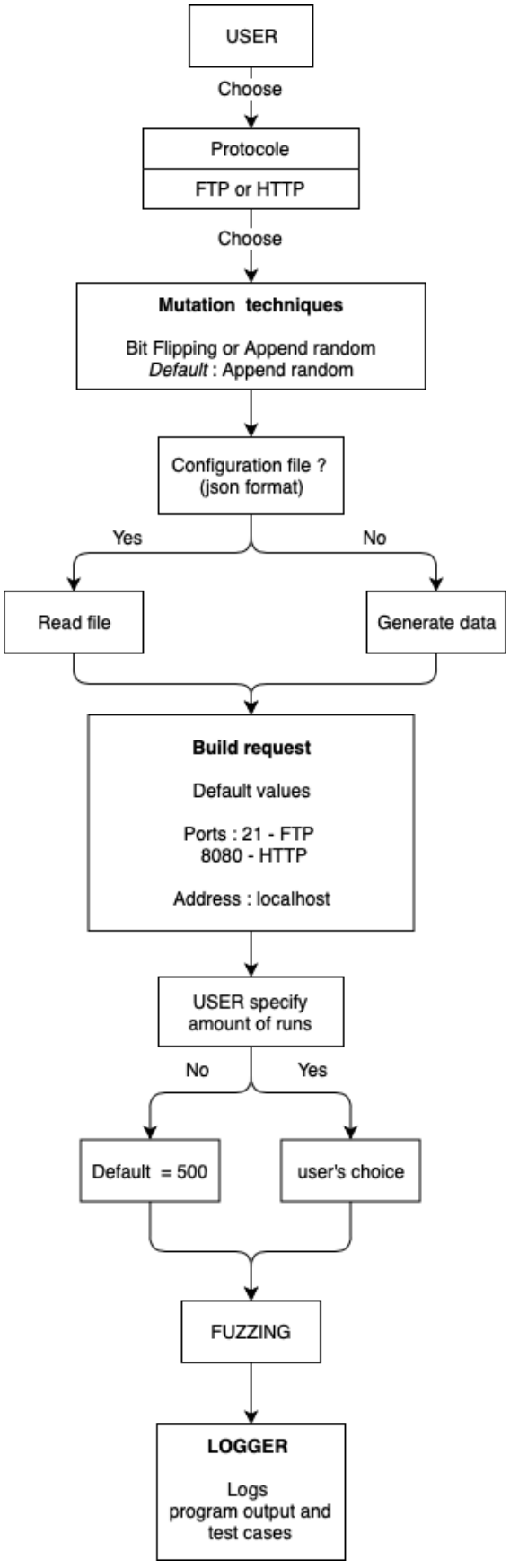
Append random : on choisit un caractère dans la chaine que l'on répète X fois.
Bit flipping : les bits de la chaine sont changés (flip) aléatoirement.



Génération

On prend en compte le protocole, les champs donnés (optionnels et obligatoires) et l'on produit une chaine adaptée.

Usage


USER
Choose
Protocole
FTP or HTTP
Choose
Mutation techniques
Bit Flipping or Append random
Default : Append random
Configuration file ?
(json format)
Yes
No
Read file
Generate data
Build request
Default values
Ports : 21 - FTP
8080 - HTTP
Address : localhost
USER specify
amount of runs
No
Yes
Default = 500
user's choice
FUZZING
LOGGER
Logs
program output and
test cases

- Le programme se lance en ligne de commande, il prend en compte plusieurs arguments dont le protocole qui est obligatoire.

```
$ python3 Fuzzer.py -h
usage: Fuzzer.py [-h] [-H H] [-P P] -T {FTP,HTTP} [-f F] [-l L] [-run RUN]
                 [-u U] [-pw PW] [-msize MSIZE] [-m {random,flip}]

optional arguments:
  -h, --help        show this help message and exit
  -H H              Host to fuzz
  -P P              Port that listens
  -T {FTP,HTTP}     Protocol
  -f F              Config file
  -l L              Log file
  -run RUN          Number of runs
  -u U              FTP connection username
  -pw PW            FTP connection password
  -msize MSIZE      Maximum size of the input
  -m {random,flip}  Mutation technique
```

Résultats

Cas de tests:

- Lorsque le fuzzer se lance, il écrit dans le fichier de log la requete envoyée au serveur ainsi que la réponse reçu.

```
2019-07-09 20:37:50,632 DEBUG FTP   127.0.0.1 SEND REQ VERSION refergvvvvvvvvvvvvvvv

2019-07-09 20:37:50,632 DEBUG FTP   127.0.0.1 RESPONSE b'500 Syntax error, command unrecognized.\r\n'
```

- Pour illustrer le cas d'une vulnirabilité, lorsque le fuzzer ne reçoit pas de réponse, le niveau est 'ERROR'

```
2019-07-09 20:39:05,935 DEBUG FTP   127.0.0.1 SEND REQ TYPE nyDlvmYlKcuXln29egVitbjMRkcZr5M5lvfWKD0WY
MVYWTnbYI423llyvXG58t3peLAGKjpTkWmKj4NUcflYiKYQxBsLWjBCc9p9smIRUfI5UoWl8mtc2X9fREprS7d2qzO6bIPfuPDUzq
zKEWK0LTJ9VnDlSmSkMKtQhb3B3ZcGbJDUh7EKaHzKdbNE4qFxMq64IlTrBEdbxacvc5TZA3ldSutBnSm7t73iNQyFvJ6c3BMPoQN
1BjSJli5wnZ1OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO
OOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOOO

2019-07-09 20:39:06,078 ERROR FTP   127.0.0.1 RESPONSE Empty Response
```

- Hormis les cas de test de chaines aléatoires, nous testons un ensemble de charactères qui peuvent engendrer des bugs (%x, \0...)

```
2019-07-09 20:57:00,884 DEBUG FTP   127.0.0.1 SEND REQ VERSION %n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%
n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n
%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%
n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n

2019-07-09 20:57:00,885 DEBUG FTP   127.0.0.1 RESPONSE b'500 Syntax error, command unrecognized.\r\n'
```

- Lorsqu'on utilise la méthode bit flipping:

```
2019-07-09 22:23:41,811 DEBUG FTP   127.0.0.1 RESPONSE b'500 Syntax errØ=eÌ§z(Iď±□~3â:·æ"j\□□\r\n'
2019-07-09 22:23:41,815 DEBUG FTP   127.0.0.1 SEND REQ VERSION ¦□□Lç¯                     Y¢zÚcÇ
d¹,§
    ßT¿mî□□½ÙûÜ`º¹ö□                              □¡Ùj□+□XH□>£ËìÇ-[9□õÌ□ñÏÝ    ç¸ÉÀøÒè~RLèlu
□j*ä+ÊÕèx-$□ß□□ØÂ□ëCX□□µ□+É□□ªµ|´Ù□üþ¡□=-è□Wlā□mP□V^`uzùçÛ□ä□□²□□×□O¤□
```

Difficultés rencontrées

Nous avons pris beaucoup de temps au début du projet pour la compréhension des différents modes de fuzzing et la mise en place de l'architecture du projet.

Améliorations possibles

Nous voulons intégrer une IA entre le **Logger** et le **Test Case Generator** : le Logger enverra des cas de tests interessant de minimisation ou de mutation au Test Case Generator.

Nous pensons également à integrer d'autre protocoles réseau, appliquer le fuzzer à des executables C (comme cité dans l'exemple de la partie Worker)...

De plus, lors de nos cas de tests, nous avons testé des cas de chaines de caractères qui peuvent déclencher des bugs (format string value : '%n', '%s', etc ...; 0, -1, 1; caractère null ...) mais nous voulons encore enrichîr cette liste.