Test par Fuzzing

Yamir Alejandro Poldo Silva

2 Octobre 2025

Plan de la présentation

Introduction

- Les tests unitaires sont essentiels mais limités.
- Même avec de bons tests unitaires, des failles critiques peuvent exister.
- Le **fuzzing** essai de combler ce vide en injectant massivement des entrées.

Définition du fuzzing

Définition

Le **fuzzing** (ou fuzz testing) est une technique de test automatisée qui consiste à fournir massivement **des entrées aléatoires, corrompues ou spécialement construites** à un programme afin d'observer son comportement et déceler des erreurs **(crashs, fuites mémoire, exceptions, réponses anormales)**.

Source: `https://github.com/resources/articles/security/what-is-fuzz-testing`

Génération des données

Deux grandes approches :

- **Génération** : produire des entrées à partir de modèles/grammaires (JSON, protocoles...).
- **Mutation** : modifier une entrée valide (seed).

Source: `https://github.com/resources/articles/security/what-is-fuzz-testing`

Exemple : génération basée sur une grammaire

Une grammaire

```
<start>   ::= <expr>
<expr>    ::= <term> + <expr> | <term> - <expr> | <term>
<term>    ::= <term> * <factor> | <term> / <factor> | <factor>
<factor>  ::= +<factor> | -<factor> | (<expr>) | <integer>
<integer> ::= <digit><integer> | <digit>
<digit>   ::= 0|1|2|3|4|5|6|7|8|9
```

- Exemples générés : `(1+2)*3`, `4/(2-1)`, `12+34*5`, `-(3+4)`

Source: `https://www.fuzzingbook.org/html/Grammars.html`

Exemple avec Radamsa (mutation)

Principe

Radamsa prend une entrée valide et génère plusieurs variantes corrompues.

```
$ echo "1␣+␣(2␣+␣(␣3␣+␣4))" | radamsa ––seed 12 –n 4
1 + (2 + (2 + (3 + 4?))
1 + (2 + (3 + ?4))
18446744073709551615 + 4)))
1 + (2 + (3 + 170141183460469231731687303715884105727))
```

- Le fuzzer explore ainsi des cas non anticipés (valeurs extrêmes, caractères inattendus).

Source: `https://gitlab.com/akihe/radamsa`

Les principales approches

- **Boîte noire** : aucune connaissance du fonctionnement interne du programme, génération d'entrée aléatoires.
- **Boîte blanche** : accès au code source, génération d'entrées spécifiques.
- **Boîte grise** : Connaissances partielles du système.
- **Guidée par la couverture** : Génération et priorisation des entrées qui augmente la couverture du code.

Source: `https://github.com/resources/articles/security/what-is-fuzz-testing`

Cycle de fonctionnement d'un fuzzer

1. Définir les objectifs et la portée. **Les systèmes testés doivent accepter une entrée**.
2. Choisir l outil selon les contraintes de tests et langage de programmation.
3. Générer ou muter des entrées et les injecter dans la cible.
4. Observer et enregistrer les anomalies **(plantage, fuites de mémoires, buffer overflows, exceptions, et autres comportements inattendues)**
5. Corriger les erreurs à partir du rapport généré et relancer la campagne de fuzzing.

Source: `https://github.com/resources/articles/security/what-is-fuzz-testing`

Outils de fuzzing

- **AFL / AFL++** — fuzzer basé sur la couverture.
- **Jazzer** — fuzzer basé sur la couverture pour la JVM / applications Java et Kotlin.
- **OSS-Fuzz** — service de fuzzing continu (Google).
- **Radamsa** — outil de *mutation* simple.
- **FFUF (Fuzz Faster U Fool)** — fuzzer léger orienté applications web / URL discovery (Go). Laboratoire en ligne `ffuf.me`

Source: `https://github.com/secfigo/Awesome-Fuzzing?tab=readme-ov-filetools`

Exemple : FFUF + Radamsa

```
# Generer 1000 mutations avec Radamsa
$ echo '{"user":"admin","email":"test@example.com"}' | \
  radamsa --seed 42 -n 1000 > payloads.txt

# Fuzzer l'endpoint avec FFUF
$ ffuf -u https://httpbin.org/post \
       -X POST \
       -H "Content-Type: application/json" \
       -d FUZZ \
       -w payloads.txt \
       -mc all \
       -t 10 \
       -maxtime 60
```

Bénéfices et limites

- Découverte de bugs inattendus (même dans du code déjà testé)
- Améliore la robustesse et la sécurité logicielle
- Complément aux tests unitaires et analyses statiques
- Efficace par rapport aux tests manuels et humains.

Bénéfices et limites

- Découverte de bugs inattendus (même dans du code déjà testé)
- Améliore la robustesse et la sécurité logicielle
- Complément aux tests unitaires et analyses statiques
- Efficace par rapport aux tests manuels et humains.

Limites

- Ne couvre pas toutes les cas possibles
- Bugs parfois difficiles à analyser ou reproduire
- Peut être non-déterministe
- Coût élevé en ressources et temps

Source: `https://github.com/resources/articles/security/what-is-fuzz-testing`

Conclusion

- Important que bien de connaître le système testé
- Complémentaire aux autres tests et aux bonnes pratiques

Sources principales

GitHub Security Lab.

What is fuzz testing ?

https://github.com/resources/articles/security/what-is-fuzz-testing (consulté le 1 octobre 2025).

Zeller, A. et al.

The Fuzzing Book.

https://www.fuzzingbook.org/ (consulté le 20 septembre 2025).

Helin, A.

Radamsa Fuzzer.

https://gitlab.com/akihe/radamsa (consulté le 20 septembre 2025).

Antithesis.

SDTalk Blog Post.

https://antithesis.com/blog/sdtalk/ (consulté le 29 septembre 2025).

Google.

OSS-Fuzz.

https://oss-fuzz.com (consulté le 29 septembre 2025).