INFOM227 : Analyse de programmes pour la cybersécurité

Travail individuel 3 : Fuzzing

Johan Rochet

Université de Namur
Faculté d'informatique
Année académique 2023-2024

1 Description du projet

La cybersécurité est un domaine essentiel de l'informatique moderne : mettre en place des systèmes d'informations sûre et résilient assure une bonne continuité des processus métiers pris en charge et assure la sécurité des données des utilisateurs. Plusieurs techniques existent et ont fait leur preuve pour tester la robustesse d'un système. Ici, nous nous concentrons sur le fuzzing : générer des inputs de manière pseudo-aléatoire afin de trouver un bug dans un programme, une fonction...

Dans ce rapport, nous présentons un fuzzer basé sur une grammaire et des mutations permettant de fuzzer une fonction python en particulier : *pandas.read_csv*. [7] Pour mettre en place une telle approche, le projet se base sur certains chapitres du Fuzzing Book [3] ainsi que sur les slides du cours : *INFOM227 : Analyse de programme pour la cybersécurité*. Le code final du fuzzer est présenté dans un notebook jupyter *final_fuzzer.ipynb* sur le repository ci-joint.

1.1 La grammaire

Mettre en place un fuzzer basé sur une grammaire, implique la construction d'une telle grammaire. La fonction *pandas.read_csv*. [7] prend en entrée une chaine de caractères représentant un CSV, de ce fait une grammaire permettant de représenter n'importe quel CSV valide a été mise en place sur base de différentes sources. [1] [2] [5]

```
list_char: List[Expansion] = srange("".join(list_ascii_printable))

CSV_GRAMMMAR: Grammar = {
    "<start>": ["<csv-file>"],
    "<csv-file>": ["<hdr>", "<rows>"],
    "<rows>": ["<row>", "<row><crlf><rows>", "<row><crlf>"],
    "<hdr>": ["<row>"],
    "<row>" : ["<fields>"],
    "<fields>": ["<field>", "<field><comma><fields>"],
    "<field>": ["<TEXT>", "<STRING>", ""],
    "<TEXT>": ["<character>", "<character><TEXT>"],
    "<STRING>": ["<dblquote><list_character><dblquote>", "<dblquote><dblquote>"],
    "<list_character>": [
        "<character>",
        "<character><list_character>",
        "<dblquote><dblquote><list_character>",
    ],
    "<character>": list_char,
    "<dblquote>": [chr(34)],
    "<comma>": [","],
    "<crlf>": ["\r\n"],
}
START_SYMBOL = "<start>"
```

Sémantiquement, un CSV est supposé avoir le même nombre de colonnes pour chacune de ses lignes [2]. C'est pourquoi, un solver ISla a été mis en place en se basant sur le chapitre du Fuzzing Book sur le fuzzing avec des contraintes [4] ainsi que la documentation d'ISla [6]. Le solver suivant permet de vérifier la contrainte exposée ci-dessus. À noter que le solver vient avec une méthode *parse* permettant de vérifier qu'une chaine de caractère est syntaxiquement et sémantiquement valide et qui renvoie des exceptions *SyntaxError* et *SemanticError* si ce n'est respectivement pas le cas.

```
solver = ISLaSolver(CSV_GRAMMMAR, # type: ignore
                    '''
                    exists int nb_comma :
                        exists <row> r :
                            (count(r, "<comma>", nb_comma)
                            and
                            forall <row> row in <rows>:
                                count(row, "<comma>", nb_comma))
                    '''
                    )
```

1.2 Démarche suivie

1.2.1 Génération des seeds

La première étape de la démarche est de pouvoir générer des seeds pour le fuzzer à base de mutation. Pour ce faire, un fuzzer basé sur la grammaire (uniquement sa syntaxe) et mettant en place la couverture de toutes les transitions possibles de la grammaire a été implémenté (GreyBoxGrammarFuzzing). Celui-ci permet de générer des inputs couvrant l'entièreté des règles de production de la grammaire. L'idée étant d'avoir un ensemble de seeds le plus représentatif possible. Toutes ces inputs sont ensuite passées au solver qui trie les inputs en deux ensembles : les inputs sémantiquement valides et celles uniquement syntaxiquement valides. Ce sont ces inputs sémantiquement valides qui seront utilisées comme seeds de notre fuzzer à base de mutation.

1.2.2 Génération des fuzz

Une fois les seeds générés, nous pouvons générer des nouveaux fuzz sur base d'un *Lang Fuzzer* avec des mutations par fragment et un schedule de type PowerSchedule. La mutation par fragment permet de gérer les mutations directement au niveau de l'arbre de dérivation et de manipuler ses nœuds (remplacement, suppressions, addition...). Les nouveaux inputs dit "mutant" sont ensuite triées par le solver en plusieurs ensembles ordonnées : les inputs valides, syntaxiquement invalides, sémantiquement invalides et les autres inputs (ensemble normalement vide). En parallèle, nous exécutons la fonction coverage sur les inputs afin de récupérer la couverture de code des inputs passées dans la fonction *pandas.read_csv*.

1.2.3 Tests des fuzz

Une fois, les inputs du fuzzer triés dans chaque ensemble ordonné, nous effectuons une dernière fois l'appel à la fonction *pandas.read_csv* pour chaque input de chaque ensemble. L'objectif ici est de non seulement vérifier que les inputs corrects sont acceptées, mais aussi que les inputs incorrects sont rejetées par la fonction. On répertorie ainsi toutes les inputs ayant mené à un comportement inattendu dans des listes. Ces inputs ne sont pas bien gérées par la fonction et méritent d'être plus amplement analysées.

2 Erreurs détectées

Comme résultat de cet analyse, nous repérons, **pour une exécution du fuzzer** sur la fonction *pandas.read_csv*, que :

— 7 inputs valides ont été rejetées.
— 2 inputs syntaxiquement invalides n'ont pas été rejetées.
— 229 inputs sémantiquement invalides n'ont pas été rejetées.

Ces nombres d'inputs ont peu d'importance (dépend d'une exécution du fuzzer à une autre), ce qui importe surtout, c'est de comprendre pourquoi la fonction à un comportement inattendu face à certains inputs. Pour expliquer cela, une analyse manuelle des fuzz a dû être mise en place. Il en ressort que les 7 inputs valides rejetées sont en fait des cas où la fonction considèrent que le CSV est vide et renvoie une erreur de type EmptyDataError. Pour les inputs syntaxiquement invalides, mais acceptées, il semblerait que la présence de ″ à la fin d'une ligne du CSV ne soit pas repéré comme une erreur. Finalement, tous les fuzz sémantiquement invalides ont été acceptés par la fonction, ce qui pourrait indiquer que la fonction ne prend pas en compte la contrainte sémantique d'égalité du nombre de colonnes pour chaque ligne du CSV.

3 Liens avec la cybersécurité

Nous avons ainsi détecté certains comportements inattendus de la fonction *pandas.read_csv*. Ces dysfonctionnements présentent un risque pour la sécurité de tous les programmes qui utilisent cette fonction. En effet, si un programme suppose que la fonction n'accepte que des entrées valides, alors il supposera que le résultat de la fonction (dataframe) respecte aussi un certain format. Si des entrées invalides s'avèrent pouvoir passer au travers, les erreurs seront propagées dans la suite du processus. Le fait que la fonction n'accepte pas certaines entrées valides est moins embêtant pour la sécurité logiciel, bien que cela puisse venir perturber son bon fonctionnement.

4 Exécuter le fuzzer

Pour exécuter le fuzzer dans un environnement python, il faut d'abord exécuter la commande suivante afin de télécharger les modules pythons utilisés par le code source :

```
pip install -r requirements.txt
```

Ensuite, pour exécuter le code, deux possibilités sont proposées :

1. Exécuter l'une après les autres les cellules du notebook *final_fuzzer.ipynb*
2. Exécuter la commande :

   ```
   python .\final_fuzzer.py
   ```

   dans le terminal afin de lancer le fuzzing.

L'opération entière de fuzzing prend quelques minutes à cause de la génération de seeds préliminaire basée sur la couverture de code (5-6 min.) ainsi que la génération des inputs mutants(1-2 min).

5 Conclusion

Le fuzzer implémenté utilise plusieurs notions du fuzzing Book [3] et du cours afin de mettre en place une démarche dans le but de pouvoir générer des inputs mutants les plus variés possibles. Cela a permis la détection d'erreur dans la fonction *pandas.read_csv*. Elle accepte en effet certains inputs syntaxiquement et/ou sémantiquement invalides et rejettent certains inputs valides. Comme le fuzzer agit de manière black box, il n'y a pas moyen de préciser exactement d'où viennent les erreurs repérées. Toutefois, l'analyse des fuzz incorrectement gérés par la fonction permet déjà de donner l'intuition sur la provenance de l'erreur en question.

Références

[1] Comma-separated values (csv) parser tutorial. URL https://textx.github.io/Arpeggio/2.0/tutorials/csv/.

[2] Common format and mime type for comma-separated values (csv) files. URL https://datatracker.ietf.org/doc/html/rfc4180#section-2.

[3] The fuzzing book. URL https://www.fuzzingbook.org/.

[4] Fuzzing with constraints. URL https://www.fuzzingbook.org/html/FuzzingWithConstraints.html.

[5] Github repository : Antlr v4 grammar : Csv. URL https://github.com/antlr/grammars-v4/tree/master/csv.

[6] The isla language specification. URL https://rindphi.github.io/isla/islaspec/.

[7] Pandas documentation : pandas.read_csv. URL https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html.