



TD ANALYSE STATIQUE CODE

23/01/2024

Sadry FIEVET

sadry.fievet@univ-cotedazur.fr



PRÉSENTATION	2
Introduction	2
DAST, IAST, SAST	3
Fonctionnement d'un SAST	4
EXERCICES	5
BANDIT	5
Installation	5
Exercice 1	6
SEMGREP	7
Exercice 2	7
Exercice 3	8
Exercice 4	9

PRÉSENTATION

Cette séance de TP est une introduction aux techniques d'analyse statique de code. Si l'analyse statique de code permet de détecter de nombreux types d'erreurs (mauvaises pratiques de code, boucles infinies, fonctions et/ou import non utilisés, ...) nous allons nous concentrer sur la détection de failles de sécurité.

Introduction

C'est depuis la création des tous premiers programmes (1940) que l'idée de les vérifier existe. On peut cependant noter une sévère prise de conscience à la suite du tragique accident que subit la fusée Ariane le 4 juin 1996.



explosion Ariane 5 1996

L'étude des circonstances de cette explosion révèle que c'est la conversion d'un nombre flottant vers un entier non signé qui a généré une altitude négative dans l'ordinateur de bord.

Les pertes furent estimées à près de 700 millions d'euros.

DAST, IAST, SAST

Il existe 3 catégories d'outils employés dans la détection de vulnérabilités.

DAST (Dynamic Application Security Testing)

Principalement utilisés pour des audits en black box, ils permettent, à la différence des SAST, la détection de vulnérabilités pendant l'exécution de l'application (runtime). ZAP Proxy, BURP, SQLMap....

IAST (Interactive Application Security Testing)

Regroupant les avantages des SAST et DAST, les IAST proposent ainsi une vue plus complète de l'application. Ils sont néanmoins plus délicats à implémenter car ils peuvent affecter sérieusement les applications.

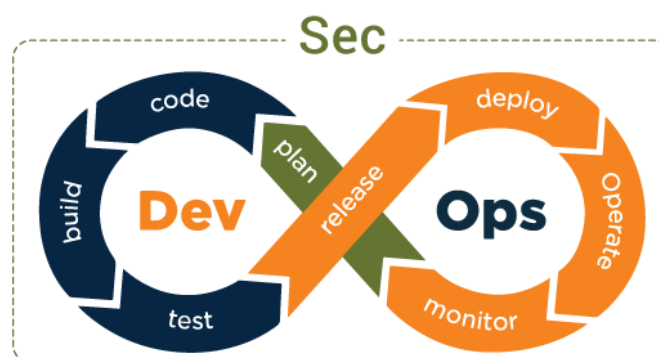
SAST (Static Application Security Testing)

L'analyse statique d'un code revient à évaluer son exécution sans pour autant l'exécuter. Pour prévoir la sortie d'un programme mais aussi savoir isoler les parties de codes qui pourraient être exploitées par un acteur malveillant il existe deux principales techniques :

- analyse 'manuelle'
- analyse avec outils

Si l'analyse manuelle permet un audit plus fin et de meilleurs résultats, les compétences et le temps qu'elle requiert ont permis aux scanners et autres outils automatiques de s'imposer comme la solution la plus utilisée dans les entreprises.

De nos jours, l'approche **DevSeCops** qui est adoptée par la plupart des entreprises du secteur informatique a automatisé la pratique de l'analyse statique de code en l'incluant dans son itération d'intégration continue.



En effet, les étapes de **code**, **build** et **test** sont souvent constituées d'analyses statiques réalisées par des outils spécifiques.

Ci-après une liste non exhaustive d'outils SAST :

[Synk Code](#)
[Veracode Static Analysis](#)
[Fortify Static Code Analyzer \(SCA\)](#)
[NodeJsScan](#)
[Checkmarx SAST](#)
[Semgrep](#)
[Bandit](#)
[Bearer](#)
[Contrast Scan](#)
[Coverity](#)
[HCL AppScan](#)
[Kiuwan](#)
[Klocwork](#)
[Reshift](#)

Fonctionnement d'un SAST

Scan

La première étape (scan) pour un compiler qui tente de comprendre de code va consister à le découper en petits morceaux appelés jetons. On peut lister les différents tokens utilisés dans Python grâce à la librairie tokenize

```
import io
import tokenize

code = b"nom = input('Entrez votre nom: ')"

for token in tokenize.tokenize(io.BytesIO(code).readline):
    print(token)
```

```
(sadry@localhost)-[~/FAC/sophia_2024/caspar/static_analysis]
$ python3 import\ io\ import\ tokenize.py
TokenInfo(type=63 (ENCODING), string='utf-8', start=(0, 0), end=(0, 0), line='')
TokenInfo(type=1 (NAME), string='nom', start=(1, 0), end=(1, 3), line="nom = input('Entrez votre nom: ')")
TokenInfo(type=54 (OP), string='=', start=(1, 4), end=(1, 5), line="nom = input('Entrez votre nom: ')")
TokenInfo(type=1 (NAME), string='input', start=(1, 6), end=(1, 11), line="nom = input('Entrez votre nom: ')")
TokenInfo(type=54 (OP), string='(', start=(1, 11), end=(1, 12), line="nom = input('Entrez votre nom: ')")
TokenInfo(type=3 (STRING), string="'Entrez votre nom: '", start=(1, 12), end=(1, 32), line="nom = input('Entrez votre nom: ')")
TokenInfo(type=54 (OP), string=')', start=(1, 32), end=(1, 33), line="nom = input('Entrez votre nom: ')")
TokenInfo(type=4 (NEWLINE), string='', start=(1, 33), end=(1, 34), line='')
TokenInfo(type=0 (ENDMARKER), string='', start=(2, 0), end=(2, 0), line='')

```

Parse

Les jetons en eux-mêmes ne reflètent rien de la grammaire de la langue. C'est là que le parser entre en jeu. Il prend ces jetons, valide que la séquence dans laquelle ils apparaissent est conforme à la grammaire et les organise dans une structure arborescente, représentant une structure de haut niveau du programme. C'est ce qu'on appelle un arbre de syntaxe abstraite (**AST**).

Analyse des AST

Le SAST va maintenant parcourir les différents AST produits dans l'étape précédente en appliquant les règles de détection qui lui sont propres.

EXERCICES

Nous allons maintenant passer à la pratique et utiliser 2 SAST de la liste ci-dessus :

- Bandit
- Semgrep

BANDIT

SAST pour analyser les programmes Python, Bandit passe en revue tous les fichiers et construit les AST résultants. Il lance ensuite une série de tests sur les nœuds puis génère un rapport. On peut l'utiliser en CI car il permet de scanner chaque merge request/commit avec une configuration modulaire des tests à effectuer.

En effet, c'est dans le fichier de configuration que l'on détermine les modules qui seront lancés.

Installation

```
$ pip3 install bandit
Defaulting to user installation because normal
DEPRECATION: Loading egg at /usr/local/lib/p
4.3 will enforce this behaviour change. A po
be found at https://github.com/pypa/pip/iss
Requirement already satisfied: bandit in /us
```

Vérifiez ensuite que tout est ok

```
(sadry@localhost)-[~]
$ bandit -v
usage: bandit [-h] [-r] [-a {file,vuln}] [-n CONTEXT_LINES] [-c CONFIG_FILE] [-p PROFILE] [-t TESTS] [-s SKIPS]
              [-l | --severity-level {all,low,medium,high}] [-i | --confidence-level {all,low,medium,high}]
              [-f {csv,custom,html,json,screen,txt,xml,yaml}] [--msg-template MSG_TEMPLATE] [-o [OUTPUT_FILE]] [-v]
              [-d] [-q] [--ignore-nosec] [-x EXCLUDED_PATHS] [-b BASELINE] [--ini INI_PATH] [--exit-zero]
              [--version]
              [targets ...]
```

Il faut maintenant générer le fichier de configuration.

```
(sadry@localhost)-[~/.../sophia_2024/caspar/static_analysis/bandit]
$ bandit-config-generator -o config.yml
[ INFO]: Successfully wrote profile: config.yml
```

On obtient un fichier qui devrait ressembler à celui ci-après

```
(sadry@localhost)-[~/.../sophia_2024/caspar/static_analysis/bandit]
$ cat config.yml

### Bandit config file generated from:
# '/usr/local/bin/bandit-config-generator -o config.yml'

### This config may optionally select a subset of tests to run or skip by
### filling out the 'tests' and 'skips' lists given below. If no tests are
### specified for inclusion then it is assumed all tests are desired. The skips
### set will remove specific tests from the include set. This can be controlled
### using the -t/-s CLI options. Note that the same test ID should not appear
### in both 'tests' and 'skips', this would be nonsensical and is detected by
### Bandit at runtime.

# Available tests:
# B101 : assert_used
# B102 : exec_used
# B103 : set_bad_file_permissions
# B104 : hardcoded_bind_all_interfaces
```

Exercice 1

Vous devez analyser les 5 scripts python contenus dans le dossier 'ex_1' de la manière suivante :

- Commencez par faire un scan Bandit sans préciser de fichier de configuration et sauvegardez ce premier rapport aux formats html et text.
- Etudiez ce rapport et lorsque vous pensez avoir trouvé quelle vulnérabilité est contenue dans chacun des 5 scripts, générez un nouveau fichier de configuration ne contenant que le module associé à cette vulnérabilité. Sauvez votre rapport aux format html et text

SEMGREP

Exercice 2

Nous allons maintenant utiliser Semgrep. Pour ce faire, vous devez dans un premier temps vous inscrire avec votre compte Github. Une fois votre inscription effectuée, installez SemGrep en suivant ces instructions :

<https://semgrep.dev/docs/getting-started/quickstart/>

Vérifiez ensuite votre installation en scannant le dossier 'semgrep_example'.

Vous devriez avoir un résultat proche de celui ci-après :

```

$ semgrep --config p/default
METRICS: Using configs from the Registry (like --config=p/ci) reports pseudonymous rule metrics.
To disable Registry rule metrics, use "--metrics=off".
Using configs only from local files (like --config=xyz.yml) does not enable metrics.
More information: https://semgrep.dev/docs/metrics

Scan Status
Scanning 9 files (only git-tracked) with 1707 Code rules:

CODE RULES

```

Language	Rules	Files	Origin	Rules
<multilang>	55	18	Community	1101
js	241	3	Pro rules	606
php	62	2		
java	231	1		
ruby	106	1		
csharp	67	1		

```

SUPPLY CHAIN RULES
Run `semgrep ci` to find dependency vulnerabilities and advanced cross-file findings.

PROGRESS
100% 0:00:05

16 Code Findings

```

Vous allez maintenant utiliser Semgrep afin d'auditer les différentes applications du dossier 'ex_2'.

Pour chacun des dossiers vous devez :

- réaliser un scan complet de l'application
- identifier les vulnérabilités High
- corriger ces vulnérabilités
- réaliser à nouveau un scan afin de vérifier votre correction

Vous devez illustrer et commenter chacune de ces étapes en fournissant des captures d'écrans, des fichiers, des pdf, etc...

Ci-après quelques exemples de captures d'écrans attendus :

Dashboard

Code

- High severity: 28
- Open findings: 105
- PR/MR fix rate: 0%

Supply Chain

- Reachable: 0
- Unreachable: 0
- Undetermined: 0

Most findings

Project name	Open findings	High severity	Fix rate
Demo project - OWASP Juice Shop	76	21	0%
vulpy	27	7	0%
Vulnerable-Code-Snippets	2	0	0%
-	-	-	-
-	-	-	-

Rules summary

Most fired | Most ignored | Most fixed

Rule	Ignored	Fix rate	Total
Reliance on Uncontrolled Component in github.com/notaryproject/notation			
Reliance on Uncontrolled Component in @keep-network/tbtc-v2			

Most vulnerabilities

Project name	Reachable	Unreachable	Undetermined
No projects found Try adjusting your filters			

New advisories

- Reliance on Uncontrolled Component in github.com/notaryproject/notation - 3 days ago
- Reliance on Uncontrolled Component in @keep-network/tbtc-v2

Findings

Group by Rule | All time

Projects: All projects

Status: Open (105) | Ignored

Category: All categories

Severity: High | Medium | Low

Confidence

105 Open Findings

Analyze (0) | Triage (0)

angular-bypasssecuritytrust (Security, Medium, Typescript)

Detected the use of `$TRUST`. This can introduce a Cross-Site-Scripting (XSS) vulnerability if this comes from user-provided input. If you have to use `$TRUST`, ensure it does not come from user-input or use the appropriate

Show more

23m	frontend/src/app/search-result/search-result.component.ts:152	IP master	Details
23m	frontend/src/app/search-result/search-result.component.ts:126	IP master	Details
23m	frontend/src/app/score-board/score-board.component.ts:162	IP master	Details
23m	frontend/src/app/data-export/data-export.component.ts:45	IP master	Details
23m	frontend/src/app/administration/administration.component.ts:66	IP master	Details

Show 7 more findings

Findings > #52656142

frontend/src/app/.../search-result.component.ts:152

23m | dev-semgrep-app[bot] | Demo project - OWASP Juice Shop | IP master | 2c757a9

angular-bypasssecuritytrust (Medium severity, Medium confidence, Monitor)

Detected the use of `bypassSecurityTrustHtml`. This can introduce a Cross-Site-Scripting (XSS) vulnerability if this comes from user-provided input. If you have to use `bypassSecurityTrustHtml`, ensure it does not come from user-input or use the appropriate prevention mechanism e.g. input validation or sanitization depending on the context.

REFERENCES

- https://angular.io/api/platform-browser/DomSanitizer
- https://cheatsheetseries.owasp.org/cheatsheets...

Activity

- New note
- Seen on IP master
- Tue, 23 Jan 2024 08:22:05 GMT
- 23 minutes ago by dev-semgrep-app[bot] via GitHub

Pattern | Metadata

Open in Editor

```
1 pattern-sources:
2   - patterns:
3     - pattern-either:
4       - pattern-inside: |
5         function ...({..., $X: string, ...}) { ... }
6       - pattern-inside: |
```

Example code

```
1 import { DomSanitizer, SecurityContext } from '@angular/platform-browser'
2 import DOMPurify from 'dompurify'
3
4 class SomeClass {
5   constructor(private sanitizer: DomSanitizer){}
6 }
```

Run locally

Exercice 3

Auditez l'application contenue dans le dossier 'ex_3', corrigez les vulnérabilités High détectées et vérifiez vos corrections avec un scan de contrôle.

Exercice 4

Les SAST comme Semgrep permettent d'identifier rapidement un certain nombre de vulnérabilités dans un code. Cependant, certaines failles leur échappent du fait qu'elles ne peuvent s'identifier qu'au runtime avec des DAST.

Pour ce dernier exercice, vous devez fournir un code contenant une vulnérabilité de votre choix qui n'est détectable qu'avec une analyse dynamique, réalisée avec un DAST.

Les documents attendus sont :

- le code vulnérable ainsi que tous les fichiers nécessaires pour lancer l'application.
- le rapport de SemGrep
- des captures d'écran de la détection de la vulnérabilité par un DAST (Burp, Zap...)
- Une courte explication de la vulnérabilité ainsi que les raisons de sa non détection par un SAST

FIN DU DOCUMENT