Secure Coding Review

- Select a programming language and application to audit.
- Perform a code review to identify security vulnerabilities.
- Use tools like static analyzers or manual inspection methods.
- Provide recommendations and best practices for secure coding.
- Document findings and suggest remediation steps for safer code.

1. Scegli linguaggio e applicazione

Per iniziare facilmente, prendi in considerazione:

- Linguaggio: Python 3.x

- Framework / App: una micro-web app tipo Flask (to-do list, blog, contact form...)

Questa scelta ti dà un ecosistema di tool consolidati e una base di codice contenuta.

2. Prepara l'ambiente di audit

Ho controllato di poter utilizzare questo codice e scansionato mediante tool automatici.

git clone https://github.com/sreecodeslayer/todo-flask.git

A simple todo app to learn Python Web development micro framework and also CRUD operations, Login/Signup User account handling by sreecodeslayer

Topics

angularjs mongo todo sweetalert font-awesome mongoengine python-flask

```bash

# 1. Clona il repository

git clone https://github.com/sreecodeslayer/todo-flask.git

cd todo-flask

# 2. Crea e attiva il virtualenv
```

python3 -m venv venv

source venv/bin/activate

Installa strumenti di analisi statica:

- Bandit (`pip install bandit`)

Installa dipendenze

pip install -r requirements.txt

- Flake8 (`pip install flake8`)

- Safety o pip-audit (`pip install safety`)

3. Analisi automatica

Esegui subito una “prima passata” con gli scanner:

```bash
bandit -r .

flake8 .

safety check
```

Annota ogni alert:

- Hard-coded secrets

- Comandi di sistema da input utente

- Query SQL non parametrizzate

4. Esegui Bandit sull’intero progetto

-r: ricorsivo; -lll: mostra tutte le severità; -f json: output in JSON

bandit -r . -lll -f json > bandit-reportj.json

5. Analizza i risultati

- Apri bandit-report.json con un editor o:

jq '.' bandit-report.json # se hai jq installato

6. YML e Report Bandit

Apri bandit.yml e copia-incolla questo contenuto:

yaml

```
# bandit.yml
# ---------
# 1) Escludi directory/file dal controllo
exclude:
 - tests/       # cartella di test
 - migrations/    # migration files

# 2) Ignora specifiche regole (codici Bandit)
skip:
 B101:          # uso di assert
 B303:          # pickle insecure

# 3) Imposta la severità minima da segnalare
minimum_severity: MEDIUM

# 4) (Opzionale) Configura confidence minima
minimum_confidence: MEDIUM
```

Spiegazione:

- exclude: directory/file che non vuoi vengano analizzati
- skip: singole regole di Bandit (Bxxx) da ignorare
- minimum_severity: ti segnalerà solo issue di livello ≥ MEDIUM
- minimum_confidence: (opzionale) filtra risultati a bassa confidenza
-

Lancia Bandit usando il config file Nel terminale, sempre dalla root, esegui:

bandit -r . -c bandit.yml

Dove:

- -r . fa partire la scansione in modo ricorsivo
- -c bandit.yml indica di utilizzare il file di configurazione personalizzato

Code scanned:

Total lines of code: 171

Total lines skipped (#nosec): 0

Run metrics:

Total issues (by severity):

Undefined: 0

Low: 2

Medium: 1

High: 1

Total issues (by confidence):

Undefined: 0

Low: 0

Medium: 4

High: 0

Files skipped (0):

(Opzionale) Esporta il report in HTML o TXT Bandit non lo fa automaticamente via config, ma puoi aggiungere le opzioni a riga di comando:
bash

HTML report

bandit -r . -c bandit.yml -f html -o bandit-report.html

Consigli extra:

- Integra Bandit in CI (GitHub Actions, GitLab CI) per far girare l'analisi a ogni push.

- Combina con `flake8` e `safety check` per coprire stile, vulnerabilità note e dependency risks.

Per l'ispezione manuale:

Passa in rassegna le parti più critiche, usando questa checklist:

- Configurazioni sensibili: `DEBUG=True`, SECRET_KEY in chiaro

- Autenticazione/Autorizzazione: esistono rotte aperte a chiunque?

- Input validation: vengono sanitizzati tutti i campi?

- Protezione CSRF: i form POST includono token?

- SQL Injection: tutte le query usano parametri placeholder?

- XSS: escape di output in template Jinja2?

- Gestione errori: non esporre stack trace in produzione

- Criptografia: hashing password con bcrypt/Argon2, non MD5/SHA1

Documenta i risultati

Metrics:

Total lines of code: 171
Total lines skipped (#nosec): 0

Test ID	Vulnerability	Severity	Line number	Description
B201	CWE-94	HIGH	170	A Flask app appears to be run with debug=True, which exposes the Werkzeug debugger and allows the execution of arbitrary code.
B104	CWE-605	MEDIUM	170	Possible binding to all interfaces.
B105	CWE-259	LOW	12	Possible hardcoded password: ''
B105	CWE-259	LOW	13	Possible hardcoded password: 'Am I being w@tched? Damn yes!'

Raccomandazioni e best practice

- Shift-left: integra scansioni Bandit/Flake8 in CI (GitHub Actions, GitLab CI...)

- Dependency management: esegui regolarmente `safety check` o `pip-audit`

- Secure Defaults: disabilita DEBUG in produzione; imposta cookie Secure e HttpOnly

- Peer review: aggiungi rubriche OWASP Top 10 nelle code review di team

- Logging & Monitoring: non loggare PII, usa WAF/IDS in produzione

Report finale

Executive summary: obiettivi, scope, tool usati

Obiettivi:

- identificare vulnerabilità nel codice, in modo da renderlo più sicuro. Si tratta di un lavoro di prevenzione.
- comprendere il funzionamento di una Secure Coding Review
- tool: Kali Linux, Bandit, Git, GitHub

```
┌──(venv)─(mina㉿kali)-[~/todo-flask]
└─$ jq '' bandit-report.json
{
  "errors": [],
  "generated_at": "2025-06-29T13:01:30Z",
  "metrics": {
    "./app.py": {
      "CONFIDENCE.HIGH": 0,
      "CONFIDENCE.LOW": 0,
      "CONFIDENCE.MEDIUM": 2,
      "CONFIDENCE.UNDEFINED": 0,
      "SEVERITY.HIGH": 1,
      "SEVERITY.LOW": 0,
      "SEVERITY.MEDIUM": 1,
      "SEVERITY.UNDEFINED": 0,
      "loc": 123,
      "nosec": 0,
      "skipped_tests": 0
    },
    "./models.py": {
      "CONFIDENCE.HIGH": 0,
      "CONFIDENCE.LOW": 0,
      "CONFIDENCE.MEDIUM": 0,
      "CONFIDENCE.UNDEFINED": 0,
      "SEVERITY.HIGH": 0,
      "SEVERITY.LOW": 0,
```

"SEVERITY.MEDIUM": 0,

"SEVERITY.UNDEFINED": 0,

"loc": 35,

"nosec": 0,

"skipped_tests": 0

},

"./settings.py": {

"CONFIDENCE.HIGH": 0,

"CONFIDENCE.LOW": 0,

"CONFIDENCE.MEDIUM": 2,

"CONFIDENCE.UNDEFINED": 0,

"SEVERITY.HIGH": 0,

"SEVERITY.LOW": 2,

"SEVERITY.MEDIUM": 0,

"SEVERITY.UNDEFINED": 0,

"loc": 13,

"nosec": 0,

"skipped_tests": 0

},

"_totals": {

"CONFIDENCE.HIGH": 0,

"CONFIDENCE.LOW": 0,

"CONFIDENCE.MEDIUM": 4,

"CONFIDENCE.UNDEFINED": 0,

"SEVERITY.HIGH": 1,

"SEVERITY.LOW": 2,

"SEVERITY.MEDIUM": 1,

```
            "SEVERITY.UNDEFINED": 0,

            "loc": 171,

            "nosec": 0,

            "skipped_tests": 0

            }

 },

 "results": [

            {

            "code": "169 if __name__ == '__main__':\n170 \tapp.run(debug=True, threaded=True,
host='0.0.0.0')\n",

            "col_offset": 1,

            "end_col_offset": 51,

            "filename": "./app.py",

            "issue_confidence": "MEDIUM",

            "issue_cwe": {

            "id": 94,

            "link": "https://cwe.mitre.org/data/definitions/94.html"

            },

            "issue_severity": "HIGH",

            "issue_text": "A Flask app appears to be run with debug=True, which exposes the Werkzeug
debugger and allows the execution of arbitrary code.",

            "line_number": 170,

            "line_range": [

            170

            ],

            "more_info": "https://bandit.readthedocs.io/en/1.8.5/plugins/b201_flask_debug_true.html",


            "test_id": "B201",
```

```
        "test_name": "flask_debug_true"
        }
 ]
}
```