Good to the last drop
Writing robust Flask and Django Apps

Clara McCreery | r2c
@r2cdev
Who are we?

me:
Clara McCreery, software engineer @ r2c
MS in Computer Science from Stanford University
Previously: researcher at MIT Lincoln Laboratory

r2c:
We’re a static analysis startup in San Francisco on a mission to profoundly improve software security and reliability
tl;dr

- Read docs for your framework & stay up to date (Flask Security Considerations or Django Security Docs)
- Favor secure-by-default frameworks (e.g. SQLAlchemy) and architectures (e.g. default True for TLS verification)
- **Check your code with code! In reality, you should automate this stuff and not have to be an expert.**

**Note:** We’ll use the open source tool Semgrep to explore security bugs during this talk. Search code with code that looks like it
Writing Secure Flask

1. Have a Security Philosophy
2. Flask Security Bugs (bug → mitigation → detection)
3. Scan Every Commit and Build
Here’s the Netflix account compromise Bugcrowd doesn’t want you to know about [Updated]

Weakness allows attackers to steal browser cookies used to authenticate Netflix users.

DAN GOODIN - MAR 23, 2020 10:37 PM UTC

How Patreon got hacked – Publicly exposed Werkzeug Debugger

October 2, 2015

What do they have in common?
Philosophy

Humans *will* make mistakes, eventually.

A short **complete** wall is better than an isolated tall gatehouse.

[https://www.reddit.com/r/pics/comments/8sayi8/a_polite_but_useless_gate/](https://www.reddit.com/r/pics/comments/8sayi8/a_polite_but_useless_gate/)
Humans will make mistakes, eventually. A short complete wall is better than an isolated tall gatehouse.

https://momentummag.com/tag/bike-lock/
YOU READY?
Writing Secure Flask

1. Have a Security Philosophy
2. **Flask Security Bugs** *(bug → mitigation → detection)*
3. Scan Every Commit and Build
# Stack/Security Overview

<table>
<thead>
<tr>
<th>Stack Layer</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpython</td>
<td>your responsibility</td>
</tr>
<tr>
<td>libc / musl / CRT</td>
<td></td>
</tr>
<tr>
<td>builtin / 3rd-party libs (Flask/Django)</td>
<td></td>
</tr>
<tr>
<td>operating system</td>
<td></td>
</tr>
</tbody>
</table>

**Ontologies:**
- [Open Web Application Security Project](https://owasp.org/www-project-top-10) (OWASP) Top 10
- [MITRE CWE](https://cwe.mitre.org) (extensive, used by governments, etc.)
Top 10 from Open Web Application Security Project

1. Injection
2. Broken authentication
3. Sensitive data exposure
4. XML XXE
5. Broken access control
6. Security misconfiguration
7. Cross-site scripting (XSS)
8. Insecure deserialization
9. Components with known vulnerabilities
10. Insufficient logging & monitoring

https://owasp.org/www-project-top-ten/
Top 10 from Open Web Application Security Project

1. Injection
2. Broken authentication
3. Sensitive data exposure
4. XML XXE
5. Broken access control
6. Security misconfiguration
7. Cross-site scripting (XSS)
8. Insecure deserialization
9. Components with known vulnerabilities
10. Insufficient logging & monitoring

https://owasp.org/www-project-top-ten/
#6: Security Misconfiguration

“Such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise.” - OWASP Top 10
#6: Security Misconfiguration

```python
from flask import Flask

if __name__ == "__main__":
    # ruleid: debug-enabled
    app.run("0.0.0.0", debug=True)
```

1. Ok to run this while developing
2. Opens up a debugging console on error

what's the problem?
How Patreon got hacked - Publicly exposed Werkzeug Debugger

October 2, 2015

How do we audit for this issue?

https://semgrep.dev/s/7g6e/

Solution: https://semgrep.dev/s/ErgL/
#6: Security Misconfiguration

@app.route('/login', methods=['POST'])
def login():
    ...
    session.clear()
    session['user_id'] = user.id
    session.permanent = True
    ...

1. Cookie could be requested over HTTP connection, not HTTPS
2. Cross-site request forgery (CSRF)
3. Cookie could be read via Javascript (stealing user sessions if XSS happens)

what's the problem?
#6: Security Misconfiguration

[Link to article](https://arstechnica.com/information-technology/2020/03/bugcrowd-tries-to-muzzle-hacker-who-found-netflix-account-compromise-weakness/)
Fix

1. Follow Flask session cookie recommendations:

   ```python
   app.config.update(
       SESSION_COOKIE_SECURE=True,
       SESSION_COOKIE_HTTPONLY=True,
       SESSION_COOKIE_SAMESITE='Lax',
   )
   ```

2. Do this for all cookies you create:

   ```python
   resp.set_cookie('user', 'flask', secure=True, httponly=True, samesite='Lax')
   ```

3. Prevent merges to master with `secure` and `httponly` set to False
How do we audit for this issue?

https://semgrep.live/s/OZR

Solution: https://semgrep.live/s/vWX
Top 10 from Open Web Application Security Project

1. Injection
2. Broken authentication
3. Sensitive data exposure
4. XML XXE
5. Broken access control
6. Security misconfiguration
7. Cross-site scripting (XSS)
8. Insecure deserialization
9. Components with known vulnerabilities
10. Insufficient logging & monitoring

https://owasp.org/www-project-top-ten/
#1: Injection

“Injection can result in data loss, corruption, or disclosure to unauthorized parties, loss of accountability, or denial of access. Injection can sometimes lead to complete host takeover. Injection vulnerabilities are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, expression languages, and ORM queries.” - OWASP Top 10

Types of injection:

- SQL injection
- Command injection
- Code injection
- Header injection
- Template injection
#1: Injection, SQL Injection

def search(request):
    user = request.GET['q']
    pw = request.GET['p']
    sql = f"SELECT * FROM users WHERE name=' %{user}% ' and password=' %{pw}% ';"

    cursor = db.cursor()
    cursor.execute(sql)
    ...

what's the problem?
#1: Injection, SQL Injection

```python
def search(request):
    user = request.GET['q']
    pw = request.GET['p']
    sql = f"SELECT * FROM users WHERE name='{user}' and password='{pw}';""'

    cursor = db.cursor()
    cursor.execute(sql)
    ...
```

user=' OR 1=1;"--"

Database sees:

"SELECT * FROM users WHERE name=' OR 1=1;"--"' and ...
#1: Injection, SQL Injection

```python
def search(request):
    user = request.GET['q']
    pw = request.GET['p']
    sql = f"SELECT * FROM users WHERE name='{user}' AND password='{pw}';"
    cursor = db.cursor()
    cursor.execute(sql)
    ...
```

user="' OR 1=1;'--"

Database sees:

"SELECT * FROM users WHERE name=''' OR 1=1;"
Fix

1. Don’t use SQL (Use an ORM)

```python
user = User.get(request.query['id'])
```

2. Maintain separation between code and data — parameterized queries

```python
user = q.exec('SELECT * WHERE id=?', request.query['id'])
```

3. Sanitize all user input (hard to remember to do it everywhere) (DO NOT USE)

```python
user = q.exec('SELECT * WHERE id=%s' % (strip_bad_chars(request.query['id'])))
```

4. Prevent merges to master that side-step any of the above (semgrep example).
Top 10 from Open Web Application Security Project

1. **Injection**
2. Broken authentication
3. Sensitive data exposure
4. XML XXE
5. Broken access control
6. **Security misconfiguration**
7. Cross-site scripting (XSS)
8. Insecure deserialization
9. Components with known vulnerabilities
10. Insufficient logging & monitoring

https://owasp.org/www-project-top-ten/
But wait, there's more!

**OWASP Top 10 for webapps**

1. Injection
2. Broken authentication
3. Sensitive data exposure
4. XML XXE
5. Broken access control
6. Security misconfiguration
7. Cross-site scripting (XSS)
8. Insecure deserialization
9. Components with known vulnerabilities
10. Insufficient logging & monitoring

**OWASP Top 10 for APIs**

1. BOLA (Broken Object Level Authorization)
2. Broken Authentication
3. Excessive Data Exposure
4. Lack of Resources & Rate Limiting
5. BFLA (Broken Function Level Authorization)
6. Mass Assignment
7. Security Misconfiguration
8. Injection
9. Improper Assets Management
10. Insufficient Logging & Monitoring
security
Writing Secure Flask

1. Have a Security Philosophy
2. Flask Security Bugs (bug → mitigation → detection)
3. Scan Every Commit and Build
What kinds of tools?

- Secure-by-design languages and frameworks
- **Static analysis**
- Formal methods
- Dynamic analysis (fuzzing, etc.)
- Let people poke it (bug bounties!)
Frameworks

Good frameworks and good defaults

- Avoid SQL injection: SQLAlchemy (not foolproof)
- Avoid XSS: React.js
- Avoiding XML vulnerabilities: defusedxml
- Avoid some CSRF with authentication via JWT (JSON Web Tokens) instead of cookies
Frameworks

What makes some better than others?

**OK:** "Do not use this in a production environment"

**BETTER:** "Do not use this in a production environment, or you will get hacked"

**BEST:** "This is production-ready by default, but you are running with  
―DANGEROUS_ALLOW_ARBITRARY_CODE_FROM_USERS, so we're letting  
you do something scary, but don't you forget about it!"

**Bad example:** render_template_string vs render_template in Flask  
**Good example:** dangerouslySetInnerHTML in React
Static Analysis Tools

- **Free:**
  - `grep` (!?)
  - Bandit (64 security rules) or Dlint (38 security rules, newer)
  - Flake8 and associated plugins
  - semgrep has >150 security checks for python
    (it's free, open-source, and made by r2c!)

- **Proprietary/closed license/SaaS:**
  - SonarPython (36 security rules)
  - Commercial tools like Coverity, Fortify, Veracode, Checkmarx ($$$)
Create new branch

$ git commit

Pull Request
Builds & Review

Merge
tl;dr

- Read docs for your framework & stay up to date (Flask Security Considerations)
- Favor secure-by-default frameworks (e.g. SQLAlchemy) and architectures (e.g. default True for TLS verification)
- Check your code with code!
  - Existing toolkits like Flake8, Bandit, Dlint
  - "Custom static analysis" with Semgrep

I’d love your feedback on this presentation! https://r2c.dev/survey
Want to learn more about Semgrep?

Learn how to write custom rules:
- Tutorial

See more rules and rulesets:
- Explore

Run on the command line:
- Docs

For more questions and to stay in touch, join our Community Slack!
Bonus content...
Top 10 from Open Web Application Security Project

1. Injection
2. Broken authentication
3. Sensitive data exposure
4. XML XXE
5. Broken access control
6. Security misconfiguration
7. Cross-site scripting (XSS)
8. Insecure deserialization
9. Components with known vulnerabilities
10. Insufficient logging & monitoring

https://owasp.org/www-project-top-ten/
The impact of XSS is moderate for reflected and DOM XSS, and severe for stored XSS, with remote code execution on the victim’s browser, such as stealing credentials, sessions, or delivering malware to the victim.

XSS is the second most prevalent issue in the OWASP Top 10, and is found in around two thirds of all applications.” – OWASP Top 10
#7: Cross-Site Scripting (XSS)

what's the problem?
#7: Cross-Site Scripting (XSS)

```html
<body>
  <h1>welcome to my website {{ app_name }}</h1>
  <li class="my-favorite-cities">
    {{ value }}
  </li>
</body>
```

value = "onmouseover=alert('xss:' + document.cookie)"

what's the problem?
#7: Cross-Site Scripting (XSS)

*Straight from the* [docs](#):

Flask configures Jinja2 to automatically escape all values unless explicitly told otherwise. This should rule out all XSS problems caused in templates, but there are still other places where you have to be careful:

- generating HTML without the help of Jinja2
- calling Markup on data submitted by users
- sending out HTML from uploaded files, never do that, use the Content-Disposition: attachment header to prevent that problem.
- sending out textfiles from uploaded files. Some browsers are using content-type guessing based on the first few bytes so users could trick a browser to execute HTML.

Another thing that is very important are unquoted attributes.
#7: Cross-Site Scripting (XSS)

Flask is configured to autoescape most views.

Some gotchas:

- Template files must have the `.html` extension
- HTML attributes **must be quoted**
- Using template variables for `href` in anchor tags is **unsafe**
#7: Cross-Site Scripting (XSS)

Flask will autoescape in this case:

```python
@app.route("/safe")
def safe():
    return render_template("safe.html", hello=request.args.get("hello"))
```

But, Flask will not autoescape in this case:

```python
@app.route("/unsafe")
def unsafe():
    return render_template("unsafe.txt", hello=request.args.get("hello"))
```

[Example output]

```html
<script>alert("hello");</script>
```

[Output]

```
localhost:5000/unsafe?hello=<script>alert("hello");</script>
```

[Confirmation]

```
localhost:5000 says
hello
```

[OK]
```
Fix

1. Use a framework like React.js which is less vulnerable to XSS

```html
<script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin>
```

2. Always use .html extension for templates

```python
render_template("index.html", context)
```

3. Set the Content Security Policy for your application

```python
response.headers['Content-Security-Policy'] = "script-src https:"
```

4. Use Flask-Talisman by Google

```python
from flask_talisman import Talisman; app = Flask(...); Talisman(app)
```

5. Prevent merges to master that violate any of the above.