Web Application Security

SECT.14.2026

- How can we anticpate threats to our application? What do we protect against?
- What are common threats & mitigations?
- What do we do when an incident occurs?

how to not get r00ted

~~SECT.14.2026~~

ohno, what happen?

you should have known better

what now?

Engineering Licensure

Mechanical, Structural, Civil Engineers are required to be licensed. This involves education, practical training, passing a licensing exams, and taking ethics courses.

Professional engineering and surveying licensure provides public protection.

The safety of buildings, bridges, roads, and other structures and technologies used in our everyday lives depends on licensed professional engineers and surveyors who have successfully demonstrated a level of competence through education, experience, and examination requirements.

-National Council of Examiners for Engineering and Surveying

Why are these professions licensed?

- https://en.wikipedia.org/wiki/Tacoma_Narrows_Bridge
- https://en.wikipedia.org/wiki/Space_Shuttle_Challenger_disaster
- https://en.wikipedia.org/wiki/Hyatt_Regency_walkway_collapse
- https://en.wikipedia.org/wiki/Bhopal_disaster

It’s just computers…

- https://en.wikipedia.org/wiki/Therac-25, 1985-1987. At least 6 deadly radiation overdoses due to race conditions. **(250x intended dose)**
- Boeing 737-MAX, 346 deaths
- 2003 Northeast Blackout: up to 4 days, 55 million affected, 100+ deaths

Today, most crtiical bugs come in the form of **security vulnerabilities**: Heartbleed (17% of internet), Log4Shell (countless corporate breaches), dozens of others with serious human & economic impact.

Yet *Security Engineer* is a job title most organizations "can’t afford".

This means all of us are responsible for ensuring the systems we build and deploy are secure.

Threat Modeling

You can’t protect against every threat.

Important to understand probability of different kinds of threats, likelihood in your system, and scale of impact. Which do we decide to spend energy on?

- 5% chance a DoS could bring the site down.
- 0.5% chance of exposing user medical records to public.
- 0.001% chance of giving patients too much radiation.

Must take into account likelihood as well as impact.

STRIDE

- **Spoofing** - log in as someone you are not with stolen or guessed credentials
- **Tampering** - sneaking malicious software onto the host (via supply chain or direct compromise)
- **Repudation** - lack of audit trails; people making unauthorized changes to a system they are allowed to access
- **Information Disclosure** - user data being exposed or breached
- **Denial of Service** - making service unavailable (overwhelmed with traffic)
- **Elevation of privilege** - An authorized user gains admin privileges due to flaws in system.

See Also: OWASP Top 10 security threats to web applications.

Attack Surfaces

- **Human** - "This is IT, we need to log in to update your system, what is your password?"
- **Physical** - "Here's a free USB drive."
- **Supply Chain** - *xz/liblzma* in 2024, typo-squatting (*jellyfish*)
- **Network** - open ports, servers with known CVEs
- **Cloud** - "AWS IAM is absurdly hard, just give this role access to everything and then narrow it *later*"
- **Application** - Unsantitized inputs, privilege escalations, verbose error messages.

Trust Boundaries

When receiving data from another system, must be careful to sanitize inputs.

- User Uploads
- APIs
- Client-side vs. Server-side validation

```
/* client side */
if(!validUser(form)) {
    turnFieldRed('from');
    disableSubmit();
}
if(!validMessage(form)) {
    turnFieldRed('message');
    disableSubmit();
}
```

```
def new_message(request):
    Post.objects.create(request.POST["user"], request.POST["message"])
    return HttpRedirect("/messages/")
```

no match for our l33t hacker skills

```
httpx.post("/new-message/", {
 "user": "admin",
 "message": "<a href='https://malicious/'>hi</a>"
})
```

my new hacking technique is unstoppable

Can we trust the backend?

```
{# server side messages from database #}
{# OR client side messages from server #}
{% for message in messages %}
   <div>
   {{ message }}
   </div>
{% endfor %}
```

Your template language does this for you, but don’t disable without reason!

```
{{ message|mark_safe }}
<a href='malicious'>click me</a>
```

What are the common application-level vulnerabilities?

SQL Injection / RCE

```
def login(username, password):
    hashed = pwhash(password)
    sql = "SELECT * FROM user WHERE username=? AND password=?"
    # some method provided by your library
    safe_sql = combine_query_with_params(sql, username, password)
```

Broken Auth

```
def delete_image(request):
    check_credentials( ... ) # returns False if not authorized
    Image.objects.delete( ... )
```

Better

```
# avoids mistakes
@login_required
def delete_image(request):
    Image.objects.delete( ... )

# Tip: simple tests for all login-required views that try without auth
```

Best (for apps with mostly logged-in behavior)

```
# even better: security as default
MIDDLEWARE += ["django.contrib.auth.LoginRequiredMiddleware"]

@login_exempt
def view_image( ... ):  ...
```

Insecure Deserialization

YAML !!

```
!!python/object/apply:os.system
args: ['rm -rf /everything']
```

Pickle

```
data = pickle.loads(pickled)
data.name
```

```
class Attack:
  @property
  def name(self):
      import random
      print("gotcha!", random.randint(10, 100))

>>> pickled = pickle.dumps(Attack())
>>> print(pickled)
b'\x80\x05\x95\x1a\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__\x94\x8c\x06Attack\x94\x93\x94)\x81\x94.'
>>> data = pickle.loads(pickled)
>>> data.name
gotcha! 42
```

Denial of Service (DoS)

Often focuses on finding slow behavior and making thousands of requests to quickly overwhelm server. Some libraries make this easier:

```
a: &a x
b: &b [*a,*a,*a,*a,*a,*a,*a,*a,*a,*a]
c: &c [*b,*b,*b,*b,*b,*b,*b,*b,*b,*b]
d: [*c,*c,*c,*c,*c,*c,*c,*c,*c,*c]
```

expands to 10000 strings, baloons memory / takes seconds to parse

```
import re
import time
N = 20
start = time.time()
re.match(r'(a+)+$', 'a' * N + '!')
print(time.time() - start)
```

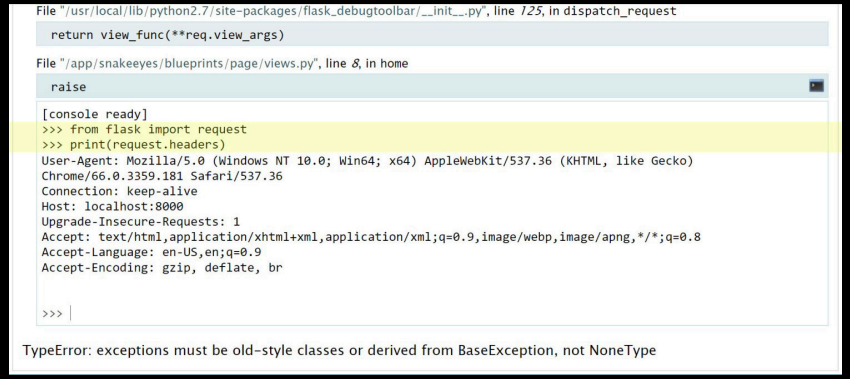
- N=20: 0.5s
- N=25: 2 seconds (already too slow!)
- N=30: 58 seconds
- N=35: 34 minutes

Debug Information

Django DEBUG=True

All URLs, possibly secrets, some code exposed!

Flask Werkezug Terminal



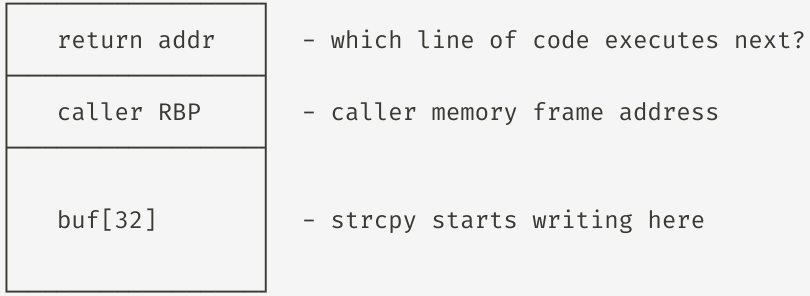
…all that plus, Remote Code Execution

Buffer Overflow

Not a problem in Python directly, but most common category of memory vuln. in non mem-safe languages:

```
void vuln(char *input) {
    char username[32];
    strcpy(username, input);  // copy all bytes from input to username
}
```

Call stack:



60 bytes of input:
[AAAAAAAAAAAAAAAAAAAAAAAAAAAA][BBBBBBBB][0×deadbeef]

How can we protect something running on the web?

SSL

Handles most common network-level issues, no change to app layer.

- HTTPS request -> forward proxy decrypts -> HTTP/WSGI request to app server

If you are handling sensitive information, **you need to use SSL.**

If users log in, **you need to use SSL.**

If you don’t want every page you read shared with your ISP/government, **you need to use SSL.**

Remember, SSL is free and simple to configure in 2026.

Hashing Passwords

Nobody other than the user should ever know a password.

Send me my password vs. *Reset my password*

^close my account

In Practice

```
user = User.objects.get(username=username)
if hashfunc(input, SALT) == user.hashed_and_salted_password:
    drake(True)

#   .————.
#  / (^‿^) \    👉
# |  \___/ |
#  \ '---' /
#   '-...-'
```

Secret Management

- Consider filesystem as insecure (another trust boundary, who can modify this FS?)
- Beware environment variable leakage:
  - config files
  - command line
  - Git commits
  - debug output
  - RCE (but likely bigger problems)

Only communicate secrets across encrypted channels.

Ideally: only store secrets in encrypted stores (Hashicorp Vault, AWS SSM, etc.)

Software Updates

Don’t update your dependencies often? Vulnerable to known CVEs.

Most common *random* breaches: scanning/exploiting old CVEs

0day: a vulnerability on the day of disclosure without a patch

too valuable to be burned on randoms, often bought by ransomware companies or state actors

Update too often? Vulnerable.

Supply chain attack: compromise a package, wait for people to update.

```
dependencies = [
    "httpx ⩾ 0.27.0",
    "pandas ⩾ 3.0.2",
]
```

Emerging norm: *exclude-newer = "7 days"*

Cooldown period where *brand new* releases aren’t adopted immediately.

Makes this *someone else’s problem* ideally, but not foolproof.

Don’t Roll Your Own

Cryptography code is complex math with all the classic edge cases (buffer overflows, etc.)

- Login/Password Hashing
- OAuth
- Secure Cookies
- JWT

USE AUDITED CODE

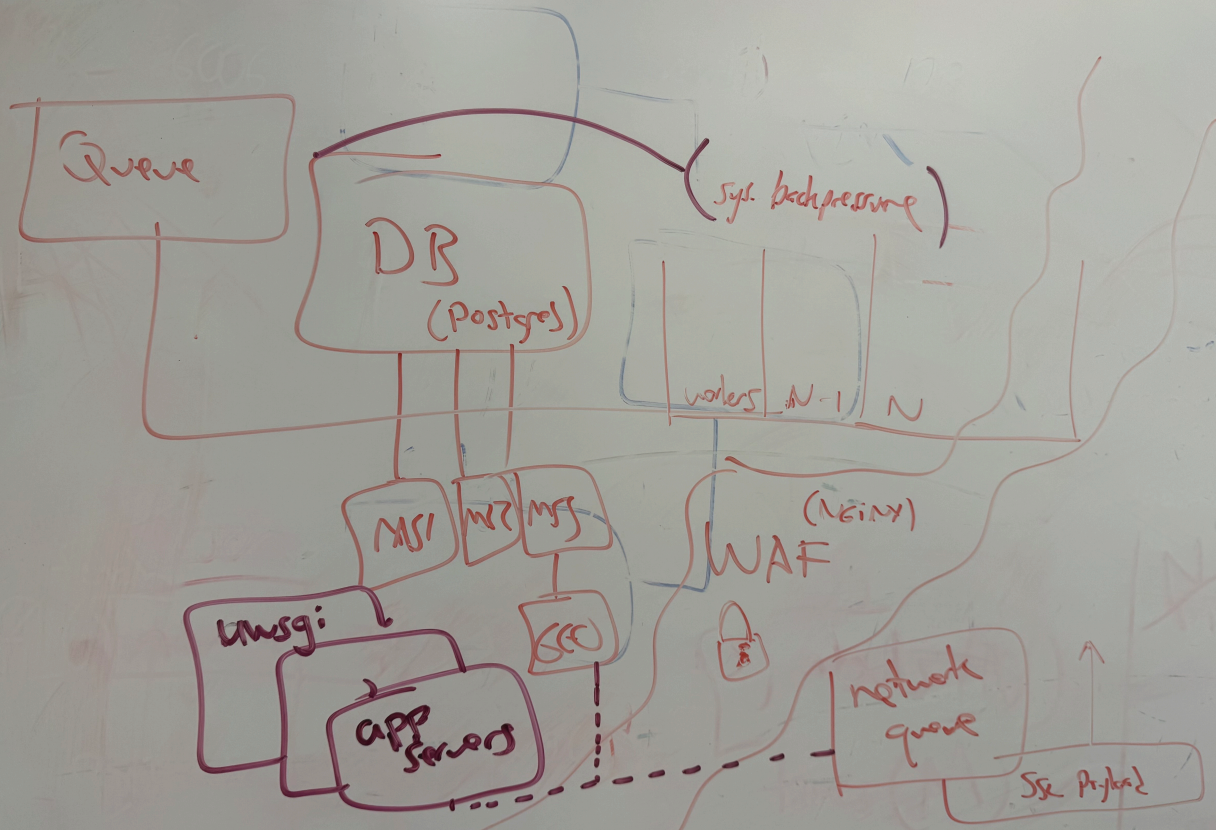
Do not write custom cryptographic code unless that is what you do for a living!

Staying Secure

- Consider what kinds of threats are *likely*.
  - Random vuln scans vs. targeted attacks vs. inside actors
- Audit vulnerable points regularly.
  - This includes humans!
- Have policies in place, panicking often pours fuel on the fire.
- Larger projects: consider red teams/bounties/etc.

secure

Wifi: little-red-riding-hood



A breach happened, what now?


DON'T
PANIC

Before the Breach (hopefully)

Good server monitoring and logging is essential. A proper response requires knowing *what* the attackers did not just that they got in.

Logs pushed to another (hopefully uncompromised) machine are the standard here.

Mitigation

Priority #1: Isolate the breached machine(s).

Take the machine offline. Rotate credentials. Lock down access.

Once isolated, removing the exploit can be done with some patience.

Document

Get to the bottom of incident:

- what happened?
- what policies were ignored?
- what new policies need to be instituted?

blameless postmortem

If people need to lie to save their jobs, they will. Accidents happen.

Flagrant/habitual disregard is a reason to remove someone from a team, making a mistake *any of us could have made* is not.

Disclose

Breaches damage trust, but we're all used to them.

Clear & accurate communication about what went wrong, who was affected, what is being done to mitigate.

Hiding breach:

- Increased legal liability.
- Attacker *gains confidence* you aren't disclosing—attacks can continue.
- Depending on domain/scale may need to notify law enforcement/insurance/etc.
- Destroys trust/credibility when discovered.
  - *adam.smith+ticketmaster@gmail.com* or *ticketmaster@jpt.sh*

Root Cause Analysis

Goes beyond technical: *What led to this failure?*

1. Identify/describe problem (Document)
2. Establish timeline where attack vector begins.
3. Distinguish root cause from other factors.
4. Establish causal graph from root cause to outcome.

Also used for non-breach issues: outages, etc.

"*5 whys*" / Fault-tree analysis.