App Sec with Python

Austin Python Meetup

Jon Oberheide

CTO, Duo Security

jono@duo.com

Hacking the Planet

PhD Researcher

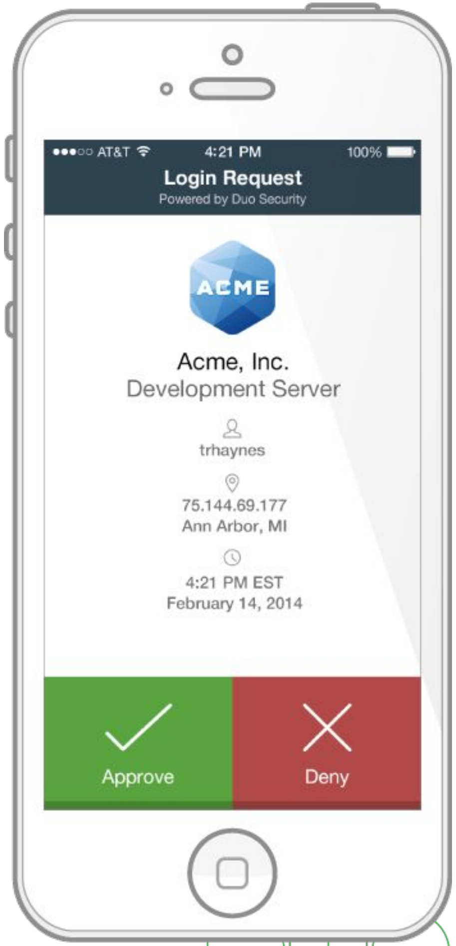
Co-Founder & CTO

Talking to you!

Hello Austin!

- We’re Duo!
  - Just opened an Austin office
  - Also based in A2, CA, UK

- Duo protect orgs against breach
  - Securing their users, their devices, and their access to corporate services



AUTODESK box Dropbox workday Atlassian zendesk slack infor yammer DOMO

FireEye paloalto NETWORKS QUALYS CYBERARK RAPID7 Palantir ARBOR NETWORKS OpenDNS rackspace DELL Secure

ViaSat xerox SYNOPSYS Lam RESEARCH ANIXTER Holcim BorgWarner GKN AEROSPACE stryker ASHLAN

CHS Community Health Systems Catholic Health Initiatives NYP BOSTON MEDICAL CENTER Stanford HEALTH CARE Centura Health San Francisco Department of Public Health BlueCross BlueShield NORTON HEALTHCARE Celgene

UNITED STATES COURTS Virginia.gov Michigan.gov Ohio.gov MICHIGAN HARVARD UNIVERSITY MIT Massachusetts Institute of Technology Stanford Yale

Forbes GH GRAHAM HOLDINGS TRIBUNE THOMSON REUTERS FOX MGM Disney PIXAR SONY NFL

facebook twitter YAHOO! mozilla King Etsy Pinterest amazon Lab126 NETFLIX hulu

Eventbrite yelp Expedia ORBITZ travelocity KAYAK priceline.com airbnb UBER lyft

organStanley Lincoln Financial Group intuit CNO FINANCIAL Quicken Loans LEGG MASON CARLYLE GROUP Arion banki Blackstone CARLY GROU

AVON e company for women GAP MEN'S WEARHOUSE HMS BURGER KING Chick-fil-A Denny's Cracker Barrel

Duo by the numbers

- 300 employees
- 7500+ customers
- 3x revenue growth the past 3 years
- 98% customer recommendation
- 67 NPS, < 4% churn
- Funded by Benchmark, Google Ventures, Redpoint, True Ventures

Hiring here in Austin

- Engineering
  - Software engineers
  - Engineering managers
- Product
  - Product management
  - Product marketing
- Security
  - App sec, corp sec
- Sales, marketing, and more!

duo.com/jobs

Application security with Python

- Duo is a big Python shop
  - Preaching to the choir
- App sec is critical
  - One XSS/SQLi = game over
- App sec with Python can be hard
  - Not a lot of great tooling/frameworks

Philosophy

... testing 'security' is not the same as testing 'functionality' ... If a door-knob opens a door, the door works. If a safe-lock opens when you dial the combination, it does not mean the safe works.

- John Tan
  Cyberspace Underwriters Laboratories

More philosophy

... if [all] users spent even a minute a day reading URLs to avoid phishing, the cost (in terms of user time) would be two orders of magnitude greater than all phishing losses.

- Cormac Herley
  [So Long, And No Thanks for the Externalities]

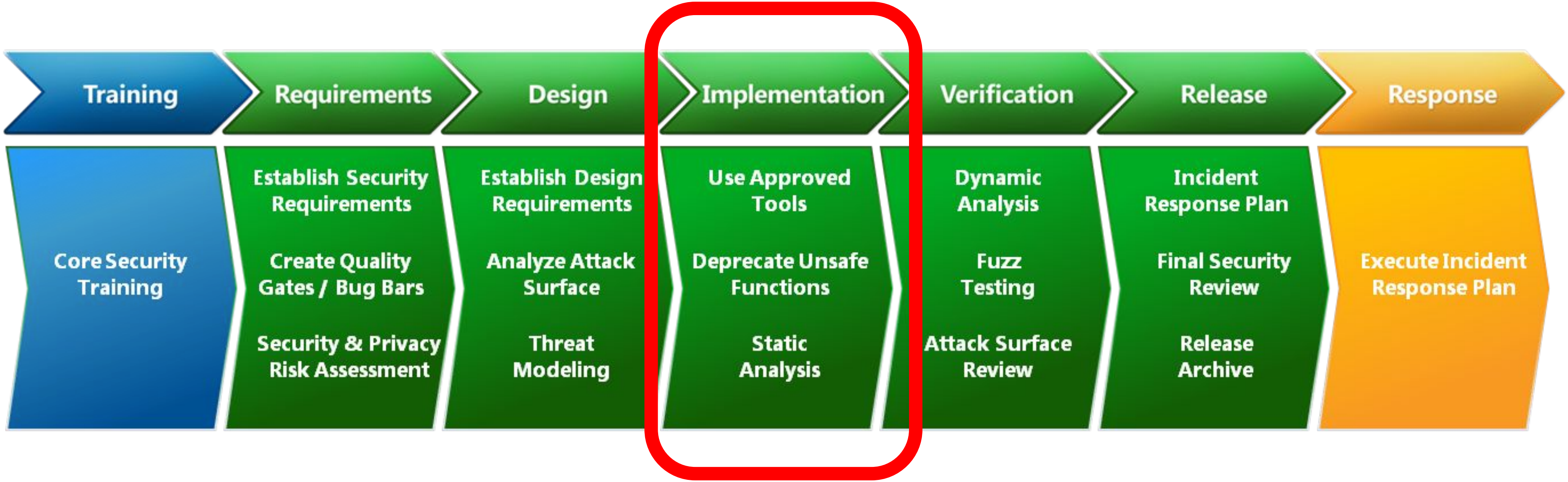
Last slide on philosophy, I promise!

It’s not enough to give developers the mere *opportunity* to write secure code.

We must build tools/frameworks that are *secure by default* and *cooperate* with lossy humans.

Ideally, they solve hard problems for us - but at a minimum, they convert subtle “security” bugs into obvious “functionality” bugs!

S[S]DL[C]



Training
Requirements
Design
Implementation
Verification
Release
Response
Core Security Training
Establish Security Requirements
Create Quality Gates / Bug Bars
Security & Privacy Risk Assessment
Establish Design Requirements
Analyze Attack Surface
Threat Modeling
Use Approved Tools
Deprecate Unsafe Functions
Static Analysis
Dynamic Analysis
Fuzz Testing
Attack Surface Review
Incident Response Plan
Final Security Review
Release Archive
Execute Incident Response Plan

OWASP top 10 risks

- Injection
- Broken Authentication and Session Management
- Cross Site Scripting
- Insecure Direct Object References
- Security Misconfiguration
- Sensitive Data Exposure
- Missing Function Level Access Control
- Cross-Site Request Forgery
- Using Components with Known Vulnerabilities
- Unvalidated Redirects and Forwards

Web framework security checklist

- What do you use for a Python web framework?
- How does it handle...
  - XSS
  - XSRF
  - SQL injection
  - Session fixation
  - Secure cookies
  - Safe redirects
  - XXE

Ex: XSRF

1. Alice logs into https://mybank.com, and gets back a session cookie:

200 OK
Set-Cookie: session-id=123-456789; path=/; domain=.mybank.com; Secure; HttpOnly;

2. Alice is tricked into opening https://evilsite.com, whose JavaScript code sends a POST to mybank.com:

POST /transfer_funds
Cookie: session-id=123-456789...
destination=evil_account_number&amount=100000&currency=USD

XSRF tokens

1. https://mybank.com sends back another cookie with an “xsrf token”:

200 OK
Set-Cookie: session-id=123-456789; path=/; domain=.mybank.com; Secure; HttpOnly;
Set-Cookie: _xsrf=SOMESECRETVALUE; path=/; domain=.mybank.com; Secure; HttpOnly;

2. On any page with a form, https://mybank.com includes the same token in an input field to be POST-ed:

…
<input type='hidden' name='_xsrf' value='SOMESECRETVALUE'>
…

3. https://mybank.com rejects any POST that without an XSRF token, or in which the token doesn’t match the Cookie

XSRF automation

- Ideally your web framework does something like:

```
token = (self.get_argument(self.xsrf_cookie_name, None) or
         self.request.headers.get("X-Xsrftoken") or
         self.request.headers.get("X-Csrftoken"))
if not token:
    raise HTTPError(403, "'_xsrf' argument missing from POST")
if self.xsrf_token != token:
    raise HTTPError(403, "XSRF cookie does not match POST argument")
```

- Turning a security risk into apparent functionality issue
- If not, can use static analysis on HTML templates

Ex: XSS

```
<html>
  <body>
    <h1>Posts</h1>
    {% for row in rows %}
    <hr>
    <p>
      {{ row.content }}
    </p>
    {% end %}
  </body>
</html>
```

XSS - Threats

- Annoy Users (e.g. "<script>alert('hi')</script>")
- Steal any data in the DOM
- Defeat XSRF protections
- Phish users' credentials, *even if this wasn't a login page*

```
<html>
  <body>
    <h1>Your Notes</h1>
    {% for row in rows %}
    <hr>
    <p>
      {{ enc_html(row.content) }}
    </p>
    {% end %}
  </body>
</html>
```

Why not just Auto-Escape?

```
<html>
  <head>
    <title>Hello, World</title>
    <script>
      var qux = '{{ enc_js(qux) }}';
    </script>
  </head>
  <body>
    <input type="hidden" name="foo" value="{{ enc_attr(foo) }}" />
    <a href="/{{ enc_url(bar) }}">{{ enc_html(baz) }}</a>
  </body>
</head>
```

Analyzing templates

- Uses a modified version of our template engine to render a template with placeholder values
  - With control flow statements no-op'ed out

- Runs an HTML parser on the output to ensure
  - Escaping is _always_ used
  - Proper escaping is used in the right context (js vs html)

Something like...

```
<html>
  <head>
    <title>Hello, World</title>
    <script>
      var qux = '{{ enc_js }}';
    </script>
  </head>
  <body>
    <input type="hidden" name="foo" value="{{ enc_attr }}" />
    <a href="/{{ enc_url }}">{{ enc_html }}</a>
  </body>
</head>
```

Mitigation: Content-Security-Policy

HTTP Header that will tell the browser from what sources it’s allowed to load (and in the case of scripts, execute) content.

- **Content-Security-Policy: default-src 'self'**
  load scripts/images/etc. only from the same domain (and do not run inline scripts or process inline CSS!)
- **Content-Security-Policy: default-src 'self'; img-src ***
  same, except allow loading images from any host

For more, see: http://cspisawesome.com

Mitigation: Content-Security-Policy

- Turns security vulnerabilities back into “ordinary bugs”…
  - *(… if your users are using supported browsers!)*
- Eliminating inline scripts usually requires some restructuring
  - *but separating code, data, and presentation is a good pattern anyway, right? :)*

"Injection" in general

"[Vulnerabilities like this] occur when data in grammar A is interpreted as being in grammar B."

- Ross Anderson, *Security Engineering*

SQL Injection - Review

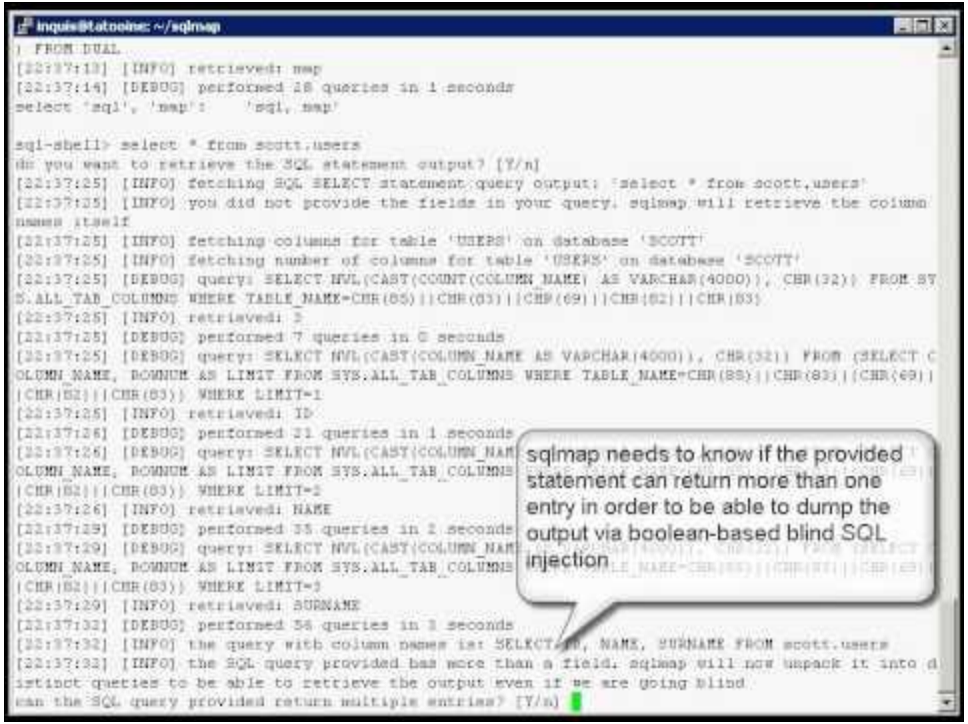
```
@defer.inlineCallbacks
def post(self):
    ukey = self.get_argument('ukey')
    rows = yield self.application.db.runQuery(
        "SELECT * FROM users WHERE ukey='%s'" % [ukey])
    self.render('user.html', rows=rows)
```

SQL Injection - Review

Fun ‘ukey’ values:

- foo' OR '1' = '1
- foo'; DROP TABLE users; SELECT 'bar
- ...

Automated tools - sqlmap



https://www.youtube.com/watch?v=whSDF8KOtK4

Parameterized Queries

```
@defer.inlineCallbacks
def post(self):
    ukey = self.get_argument('ukey')
    rows = yield self.application.db.runQuery(
        "SELECT * FROM users WHERE ukey=?", [ukey])
    self.render('user.html', rows=rows)
```

Can you see the difference?

What if, instead...

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String

Base = declarative_base()
class User(Base):
    __tablename__ = 'users'

    uid = Column(Integer, primary_key=True)
    ukey = Column(String)
...

def post(self):
    ukey = self.get_argument('ukey')
    users = self.session.query(User).filter(User.ukey==ukey)
    self.render('user.html', users=users)
```

Or...

```
from sqlalchemy.sql import select
...

def post(self):
    s = select([users]).where(users.c.ukey == ukey)
    rows = self.conn.execute(s)
    self.render('user.html', rows=rows)
```

Magic!

- Bad news: Sometimes ORMs have vulns

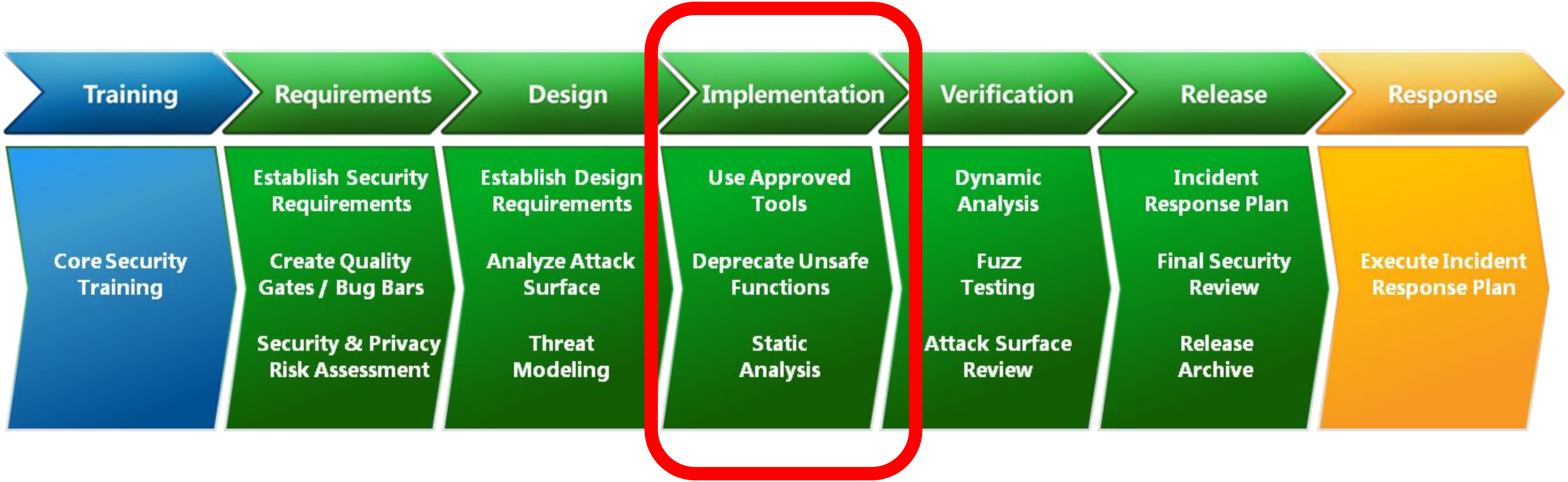
SQLAlchemy 'limit' and 'offset' Parameters SQL Injection Vulnerabilities

SQLAlchemy is prone to multiple SQL-injection vulnerabilities because it fails to sufficiently sanitize user-supplied data before using it in an SQL query.

Exploiting these issues could allow an attacker to compromise the application, access or modify data, or exploit latent vulnerabilities in the underlying database.

- But generally, use an ORM to have to worry _less_ about SQLi
  - You will likely still have raw queries in deep dark corners

S[S]DL[C]


Training
Requirements
Design
Implementation
Verification
Release
Response
Core Security Training
Establish Security Requirements
Create Quality Gates / Bug Bars
Security & Privacy Risk Assessment
Establish Design Requirements
Analyze Attack Surface
Threat Modeling
Use Approved Tools
Deprecate Unsafe Functions
Static Analysis
Dynamic Analysis
Fuzz Testing
Attack Surface Review
Incident Response Plan
Final Security Review
Release Archive
Execute Incident Response Plan

Static analysis

- Not all app sec problems can solved by a framework…
- We’re big fans of static analysis
  - More flexible to solve unique problems

Static program analysis is the analysis of computer software that is performed without actually executing programs (analysis performed on executing programs is known as dynamic analysis). In most cases the analysis is performed on some version of the source code, and in the other cases, some form of the object code.

- Basically, analyzing your code...with code!

Static analysis

- Why?
  - Lots of ways to do security engineering
  - Code review, testing, QA, attack monitoring, etc
- Automation
  - Humans are lossy and make mistakes
- Scale
  - Don’t let security get in the way of productivity
- Cost
  - Much cheaper to catch bugs in dev vs prod

Static analysis

... but Python is dynamic!

This doesn't work *in theory*, but within some constraints, it can be useful *in practice*.

Static analysis

- Commercial tools
  - Powerful, but extremely expensive
  - Veracode
  - Coverity
  - Fortify
  - Limited Python capabilities
- Is it hard to roll our own?
  - Mostly no, sometimes yes ;-)

Homegrown hacks

Example: make sure that we only ever use Python’s “SystemRandom” class to generate random values

v1: Basically, grep for instances of:

- ‘random\.\w+’ (other than ‘random.SystemRandom)
- ‘from random import .*’ (other than ‘from random import SystemRandom)

v2: Use the python AST

Abstract Syntax Tree

```
>>> import ast
>>> m = ast.parse("from random import SystemRandom")
>>> ast.dump(m)
"Module(body=[ImportFrom(module='random', names=[alias(name='SystemRandom', asname=None)], level=0)])"
>>> m.body[0].module
‘random'

>>> m2 = ast.parse("self.db.execute('SELECT * FROM users WHERE uname=%s' % (uname))")
>>> ast.dump(m2)
"Module(body=[Expr(value=Call(func=Attribute(value=Attribute(value=Name(id='self', ctx=Load()), attr='db',
ctx=Load()), attr='execute', ctx=Load()), args=[BinOp(left=Str(s='SELECT * FROM users WHERE uname=%s'),
op=Mod(), right=Name(id='uname', ctx=Load()))], keywords=[], starargs=None,
kwargs=None))])"
```

Checking SystemRandom

```
class RandomVisitor(ast.NodeVisitor):
    def visit_Attribute(self, node):
        if (isinstance(node.value, ast.Name) and node.value.id == 'random'
            and node.attr != 'SystemRandom'):
            raise BadRandomGenerator(node.lineno)

    def visit_ImportFrom(self, node):
        if (node.module == 'random'
            and any(alias.name != 'SystemRandom' for alias in node.names)):
            raise BadRandomGenerator(node.lineno)

with open(some_python_module, 'r') as fp:
    m = ast.parse(fp.read())
    RandomVisitor().visit(m)
```

Common anti-patterns

- Bad stuff
  - Pickle, subprocess/os.system/etc, basically any XML parsing, etc…
- Any time you ever want to say/enforce:
  - “Don’t ever call that module, function, method, whatever!”
- Hook into your build/testing/CI framework

AST-based frameworks

- Not much for python
  - Most AST framework are linters! Pylint, pyflakes, etc
  - Checking for code quality/etc vs security issues
- Bandit is a great tool though
  - Can write easier checks than our SystemRandom example
  - https://github.com/openstack/bandit

Bandit framework

```
@bandit.checks('Call')
def prohibit_unsafe_deserialization(context):
    if 'unsafe_load' in context.call_function_name_qual:
        return bandit.Issue(
            severity=bandit.HIGH,
            confidence=bandit.HIGH,
            text="Unsafe deserialization detected."
        )
```

B312	telnetlib
B313	xml_bad_cElementTree
B314	xml_bad_ElementTree
B315	xml_bad_expatreader
B316	xml_bad_expatbuilder
B317	xml_bad_sax
B318	xml_bad_minidom
B319	xml_bad_pulldom
B320	xml_bad_etree
B321	ftplib
B401	import_telnetlib
B402	import_ftplib
B403	import_pickle
B404	import_subprocess
B405	import_xml_etree
B406	import_xml_sax
B407	import_xml_expat
B408	import_xml_minidom

Beyond ASTs

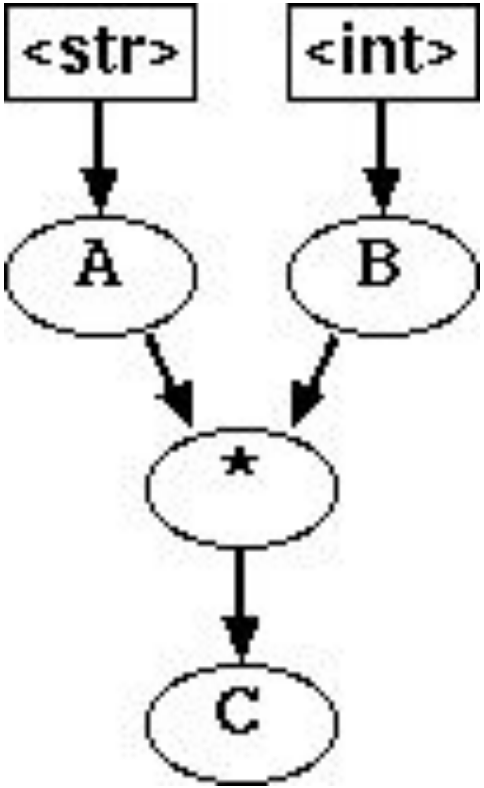
- Not all badness can be detected via AST
- Ex: SQL injection
  - db.execute(query, parameters)
  - How do I ensure the query variable does not contain external attacker-influenced input???
  - Python is dynamic!
- More advanced static analysis
  - Control flow, data flow, type flow, taint analysis

Taint analysis, hugely simplified

1. Parse the code into an Abstract Syntax Tree
2. Build a "program dependence graph"
3. Find a "node" you want to consider, and backtrack through the graph

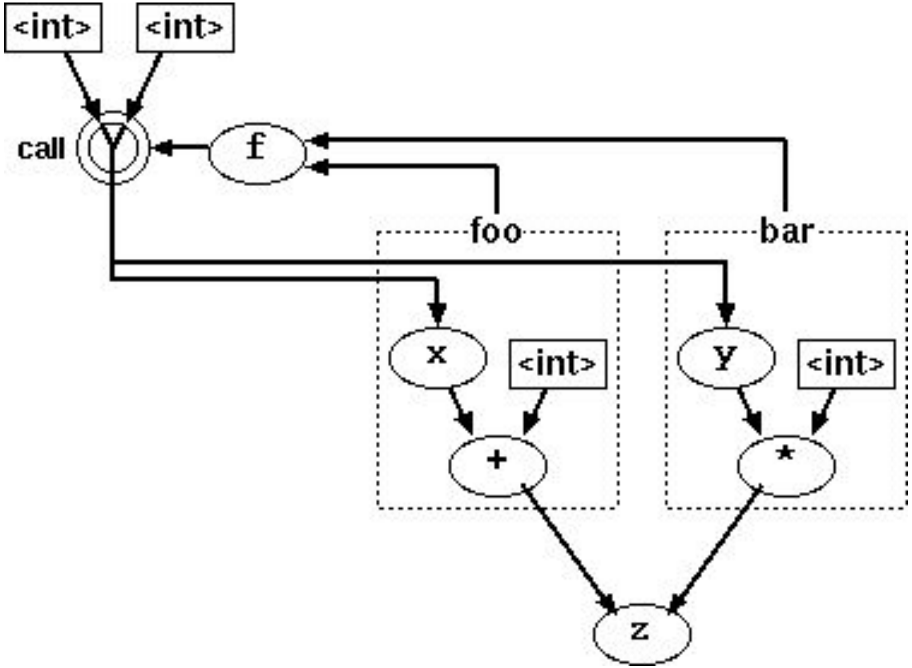
Type flow is simple, right?

A = 'xyz'
B = 2
C = a*b



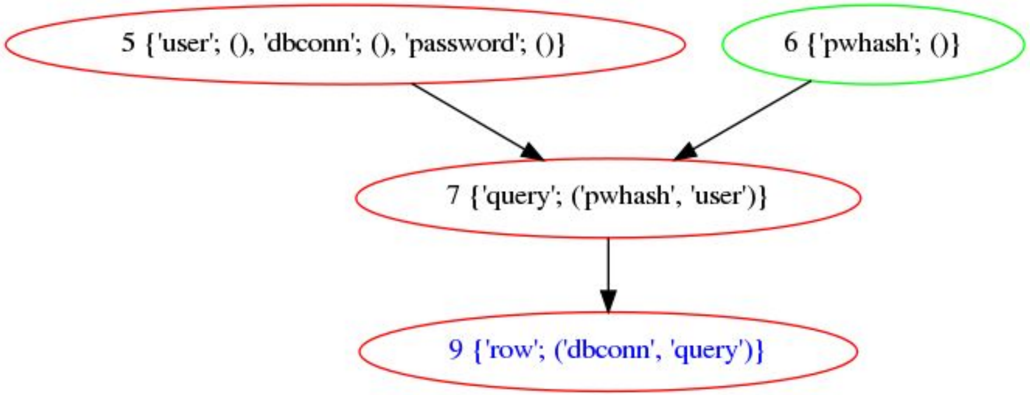
A bit more more complicated

```
def foo(x):
    return x+1
def bar(y):
    return y*2
f = foo
z = f(1)
f = bar
z = f(2)
```

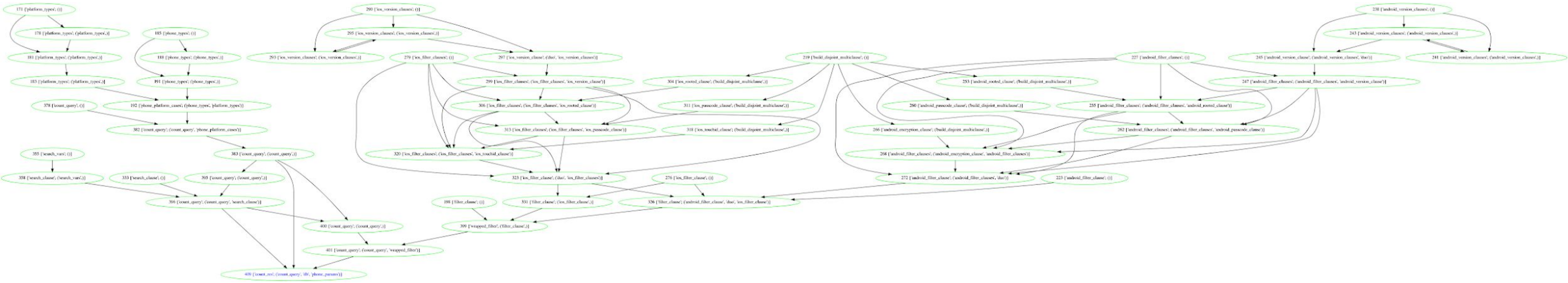


A pseudo-realistic example

```
@defer.inlineCallbacks
def login_user(dbconn, user, password):
    pwhash = hashlib.sha1(password).hexdigest(); ← note: don’t do this, just an toy example
    query = "SELECT uid FROM users WHERE uname='%s' AND password='%s'" % (user, pwhash)
    row = yield dbconn.runQuery(query)
    defer.returnValue(row.uid if row else None)
```



Real world look more complicated



The SQLi use case

- SQLi example
  - Find db.execute() node and query variable
  - Backtrack ancestry of query variable
  - Ensure that roots of query var are string/int literals
- It actually works!
  - For some values of “works”
  - Minimize FNs, escalate FPs
  - If tool can’t understand the code, provide suggestions to the developer on how to restructure

We’re not program analysis experts!

- v1: based on an old, unmaintained project called "pyntch"
- v2: we contracted logilab (developers of pylint) to build us a python dataflow / taint analysis framework based on astroid
  - https://www.astroid.org/
- To OSS’ed Real Soon Now™...

Wrap-up

- Use frameworks and tools that prevent entire classes of bugs by default
  - Either by intentionally mitigating vulnerabilities or simply by encapsulating dangerous code so you don’t have to deal with it.
- If you see an anti-pattern, write a script to enforce it!
  - Can be quite basic, especially if you pair it with peer code reviews and consistent coding norms
- Don’t forget about the rest of the SDLC!

Thanks!

Questions?

@jonoberheide

jono@duo.com

https://duo.com