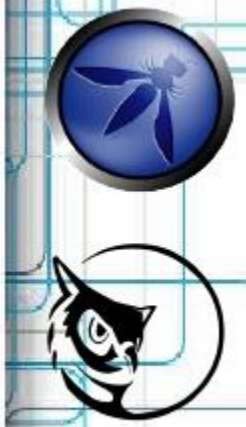
Secure infrastructure as code
How I built w3af.org

Andrés Riancho – April / March 2013
OWASP LATAM Tour



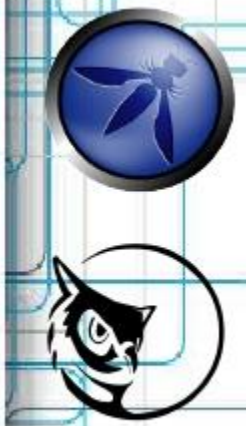
/me

- w3af project leader (open source web application security scanner)
- Software developer (Python)
- Web application security expert

@w3af

- I'm no infrastructure as source expert
- Use my advise with caution

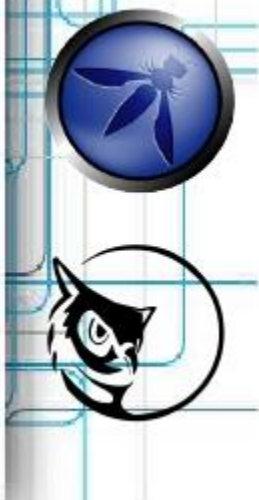
Glossary: clear terms before diving in



Source code

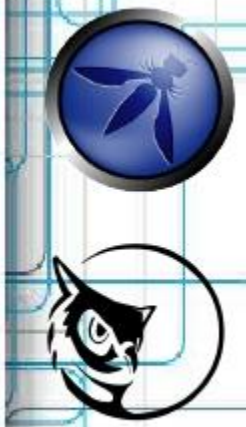
"a sequence of instructions written to perform a specific task with a computer"

```
def add5(x):
    return x+5

def dotwrite(ast):
    nodename = getNodename()
    label=symbol.sym_name.get(int(ast[0]),ast[0])
    print '    %s [label="%s"' % (nodename, label),
    if isinstance(ast[1], str):
        if ast[1].strip():
            print '= %s"];' % ast[1]
        else:
            print '"];'
    else:
        print '"];'
        children = []
        for  in n, childenumerate(ast[1:]):
            children.append(dotwrite(child))
        print ,'    %s -> (' % nodename
        for  in :namechildren
            print '%s' % name,
```



Test Driven Development

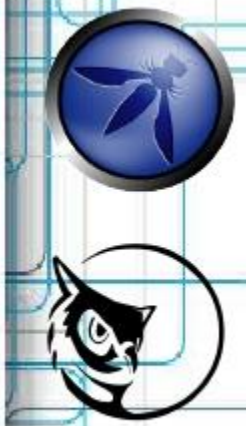
"Test-driven development (TDD) is a **software development process** that relies on the repetition of a very short development cycle: **first the developer writes an (initially failing) automated test case that defines a desired improvement** or new function, **then produces the minimum amount of code to pass that test**, and finally refactors the new code to acceptable standards."



TDD: Tests first

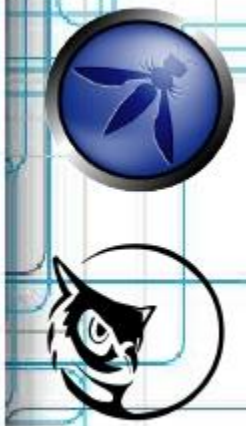
test_is_odd.py

```
import unittest
from mymodule import is_odd

class IsOddTests(unittest.TestCase):
    def test_one(self):
        self.assertTrue(is_odd(1))

    def test_two(self):
        self.assertFalse(is_odd(2))
```



TDD: Test fails

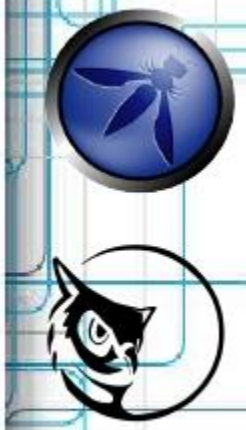
```
pablo@eulogia:/tmp$ nosetests test_is_odd.py
E
======================================================
ERROR: Failure: ImportError (No module named
------------------------------------------------------
Traceback (most recent call last):
  File "/usr/local/lib/python2.7/dist-package
    addr.filename, addr.module)
  File "/usr/local/lib/python2.7/dist-package
    return self.importFromDir(dir_path, fqnam
  File "/usr/local/lib/python2.7/dist-package
    mod = load_module(part_fqname, fh, filena
  File "/tmp/test_is_odd.py", line 2, in <mod
    from mymodule import is_odd
ImportError: No module named mymodule

------------------------------------------------------
Ran 1 test in 0.002s
```



TDD: Write the code

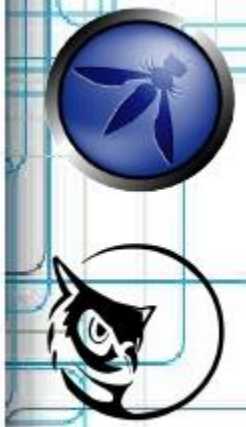
mymodule.py

```
def is_odd(n):
    return n % 2 == 1
```



TDD: Tests PASS

```
pablo@eulogia:/tmp$ nosetests test_is_odd.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```



"Servers* you can rent by the hour"

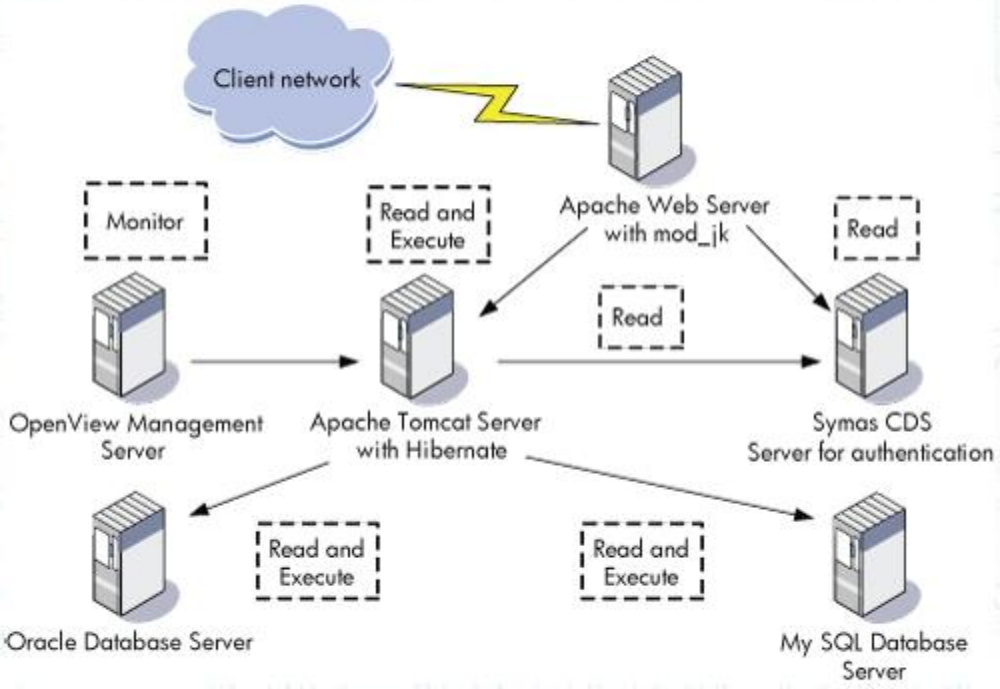
* Not only servers as in an Ubuntu 12.04 ec2 instance, also services like managed databases, ready to use email servers, Queues, etc.

- In most cases, the user manages his own resources by **creating and shutting down servers** as required by network load.

- Since you "pay for what you use", there is no need to buy expensive hardware up-front.

Infrastructure

- Can be described as “daemons and services running on an operating system which are all configured to provide one or more services to users”

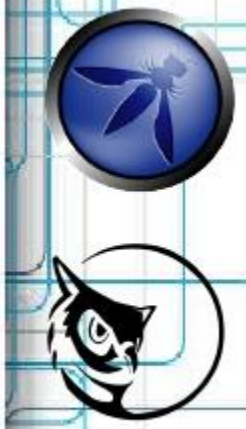


Classic infrastructure

The good

Classic infrastructure: The good

- We've been doing this for 20+ years
- Every good sysadmin knows how to configure a server. **It's on his job description.**
- Works “well” in most scenarios

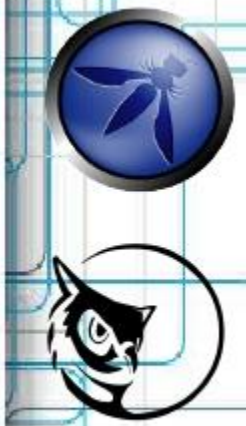


Classic infrastructure

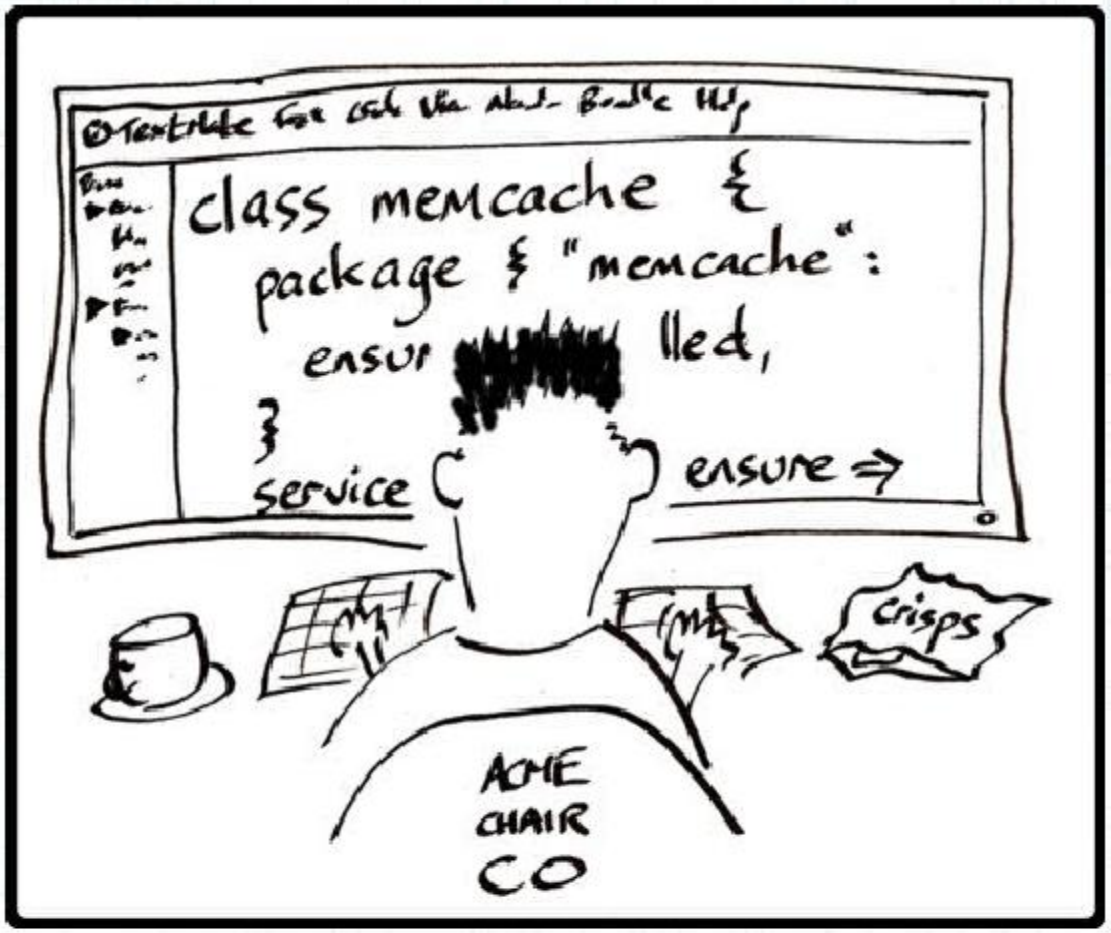
The bad

Classic infrastructure: The bad

- **Poor change control**: *"Who changed X, which broke feature Y?"*
- Hard to create **dev/QA/staging servers** which are identical to production ones. Leads to *"Works in my environment"*
- **Doesn't scale** if our application gets popular, how do we handle 1M users? What about 10M? One sysadmin can't configure 1k servers over the weekend.

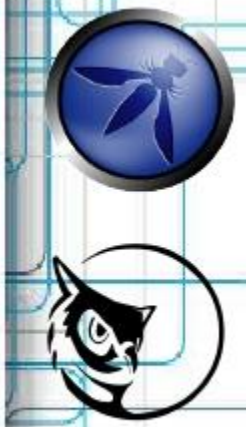


Infrastructure as Code



Code + HW = Running infrastructure

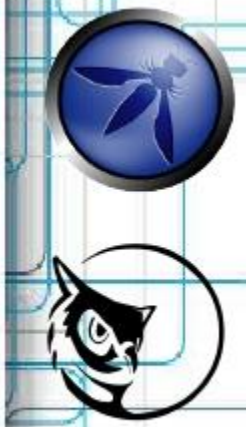
- **All your infrastructure is defined by custom made software** and stored in a repository
- Run this software any number of times and you'll get a **clone of your infrastructure**, all in an automated way
- Building and maintaining a modern server begins to look a lot like **managing a software project**



Problems solved!

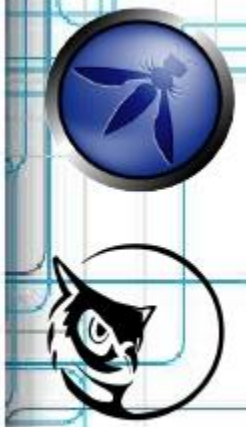
- **Scalability**: deploy N servers, all equal.
- **Change control**: “git log” to view the latest changes to a server
- **No regressions**: Apply TDD your infrastructure development process and you'll know when a new change adds a regression
- **Easily move to a previous version:**

  “git checkout <revision>; fab deploy”



New challenges

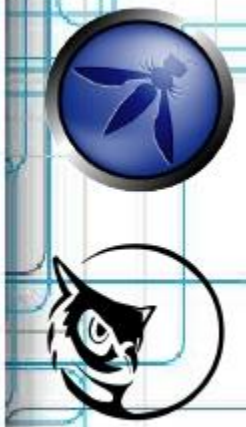
- The sysadmin needs to learn developer skills such as:
  - SDLC applied to the infrastructure
  - Concepts like classes, refactoring, coding standards, etc.
  - Test Driven Development *(optional)*



New features

- It's code, **share it**. If one team finds a bug in the SSH, he can share the new configuration with other teams.

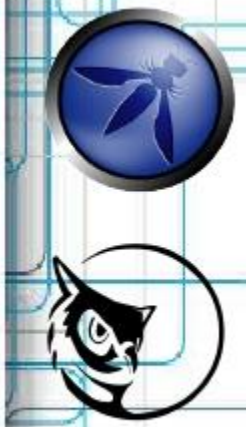
- It's code, **re-use it**. All teams can contribute on basic OS configuration, specific teams on DB, Web, etc.



Example scenario: w3af.org

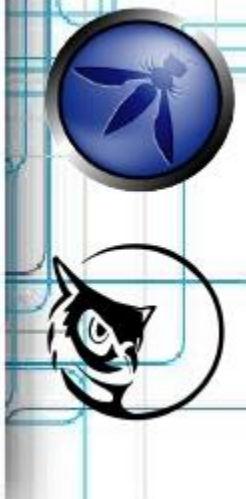
The requirements were simple:

- Secure
- Fast load speeds
- Easy to add new pages and blog posts
- Well documented
- Easy to develop new features and test them locally
- Fail gracefully (if hacked, DoS'ed, etc.)
- ***Learn something in the process***



The tools: Fabric for python fans

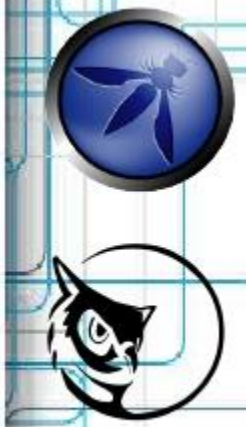
- Puppet, Chef, Fabric, etc. since **Fabric is Python**, I decided to use that for my deployments.
- “import unittest” for writing the tests
- Boto for interacting with the ec2 API



The code

- 19 unittests
- 850 lines of Python code
- 15 configuration files for Apache, Varnish, etc.

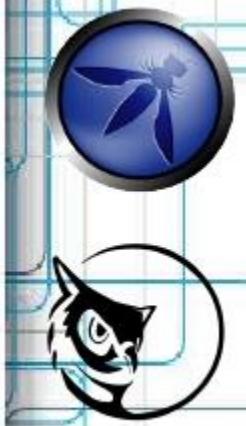
- Interesting code snippets:
  - fabfile.py
  - utils.ec2 , create new ec2 instance
  - utils.apache , configure apache



Finally: Focus on security

- TDD helps developers define clear requirements and **make sure the code they write covers them**

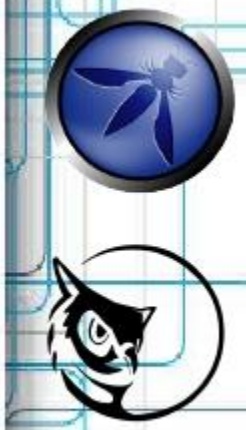
- With infrastructure as code we can **create security requirements** to make sure the OS and application are secure



TDD, nmap, infrastructure as code

- **Requirement**: *"Web servers should only be accessible via port 80 and 22"*

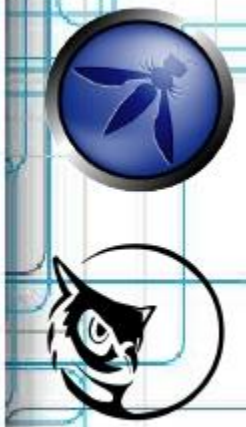
test_port80.py



Test wordpress improved security

- **Requirement**: *"Digest authentication needs to secure /wp-admin/"*
- ***Requirement:*** *"/wp-includes/ shouldn't be accessible using a browser"*

test_wordpress_htaccess.py



PHP Eggs are disabled

- **Requirement**: *"PHP is configured to hide PHP eggs (expose_php = Off)"*

test_php_config.py

Nikto output is harmless

- **Requirement**: *"Nikto only identifies false positives and very low risk information"*

test_nikto.py

HTTP headers, the secure way

- **Requirement**: *"The web application sends the HTTP headers required to avoid ClickJacking, information gathering and XSS attacks"*

test_security_headers.py

Handling (security) bugs

- Bug is reported
- (if not yet available) deploy a development server in a VM and manually reproduce the bug
- Write unittest to reproduce it
- Change configuration / application code to fix it
- Run test to verify fix
- Run all tests to verify there are no regressions
- Commit/Push changes to Git
- Apply changes to production environment using Fabric

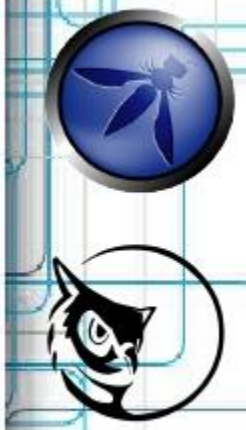
Handling (security) bugs

- **Relax:** knowing that everything works AND it won't be broken in the future if you follow the procedure

Monitoring using unittests

Since unittests verify that the server behaves the way we want, and is secure according to our tests, it's a good idea to run them periodically (once every X hours)

Remember: unittests need to be **idempotent**.

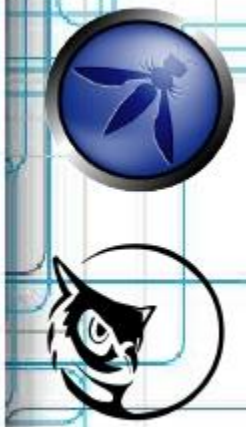


Enforcing policies with unittests

Usually a **security policy is a Word document** (that nobody reads) and states: *"All passwords need to be X chars long"*

With infrastructure as code we can make sure this is actually enforced:

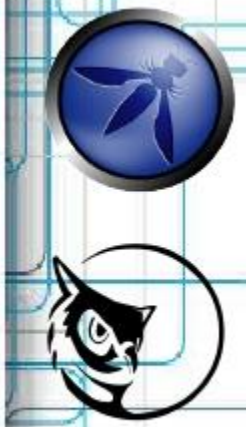
- Write a **unittest that verifies the configured password length**
- Make it **mandatory to run** on all servers
- Our unittest could also run **john the ripper** to verify that passwords are strong enough



Conclusions

- If properly implemented, infrastructure as code can **reduce the number of infrastructure vulnerabilities, bugs, and increase uptime.**
- Using TDD in your infrastructure code reduces regressions

- Requires skilled sysadmins
- Migration to infrastructure as code is time-consuming



Questions?

Thanks!

@w3af

andres.riancho@gmail.com