Security Regression

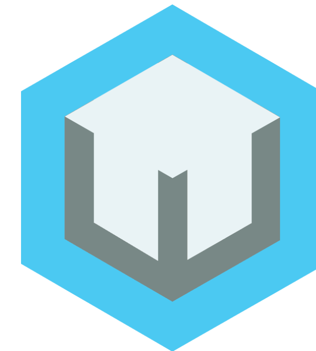
Addressing Security Regression by Unit Testing

Christopher Grayson
@_lavalamp

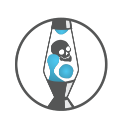
Introduction

WHOAMI

- ATL
- Web development
- Academic researcher
- Haxin' all the things
  - (but I rllly like networks)
- Founder
- Red team



@_lavalamp

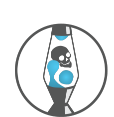


WHY'S DIS

- Security regression is a **huge** problem
- Lots of infrastructure built around regression testing already
- Let's leverage all of that existing infrastructure to improve application security posture at a minimal cost to development teams

Agenda

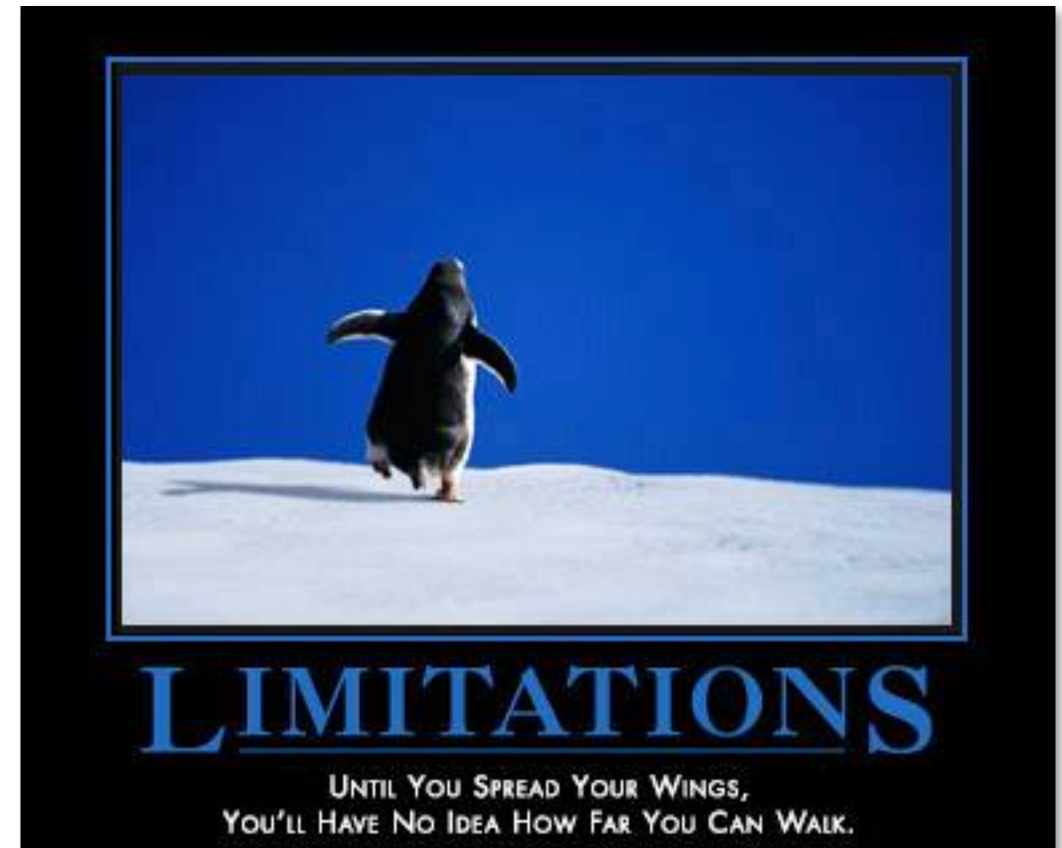
1. Background
2. Dynamic Security Test Generation
3. Non-dynamic Security Test Generation
4. Conclusion



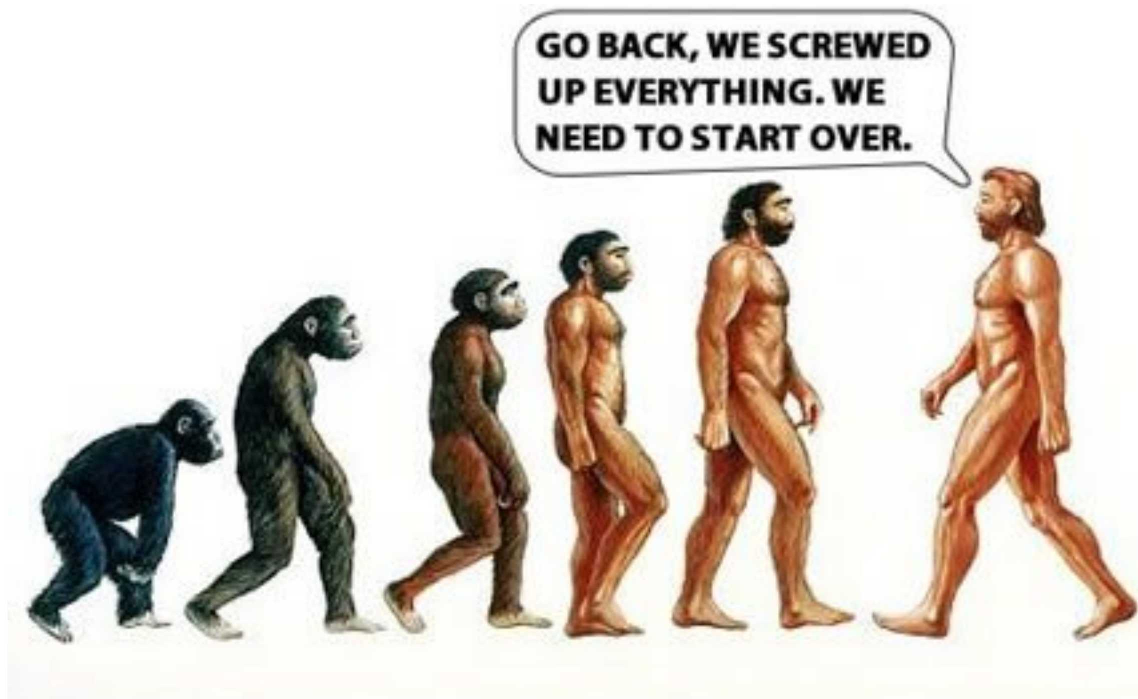
Background

A Bit More on Motivation…

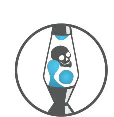
- I've always loved breaking into things, have been doing this professionally since 2012
- Go in, break app, help client with remediation, check that remediation worked – **great!**
- Come back 3-6 months later and test again, same vulns are back (commonly in the same places)
- Offensive testing is good at diagnosing - not solving



Regression Testing



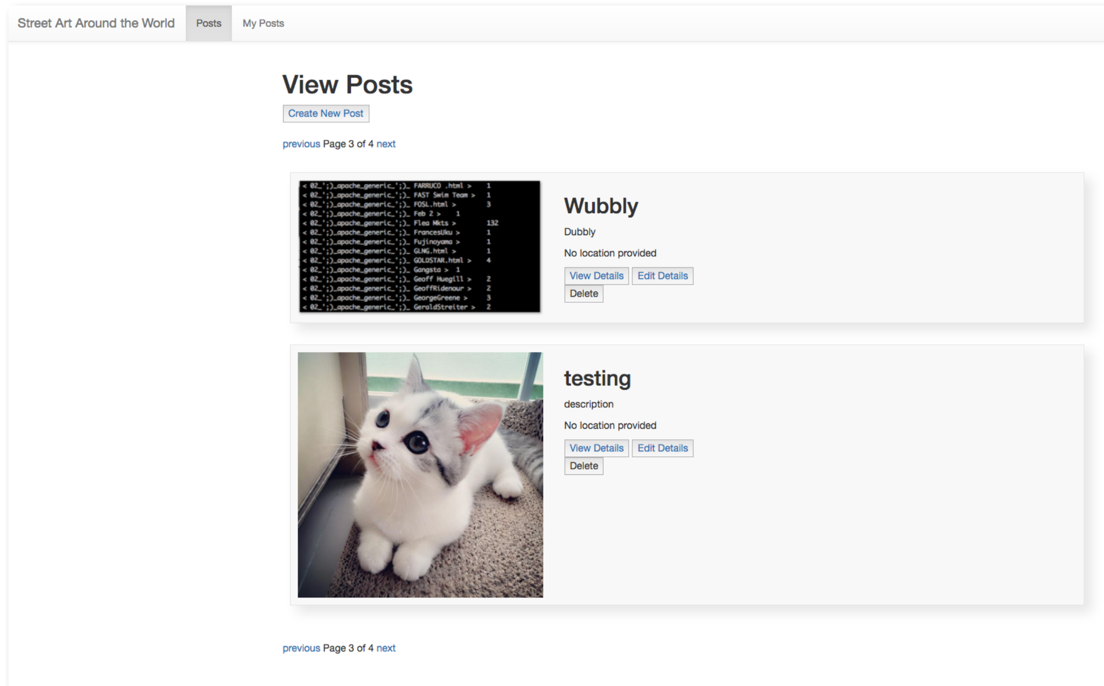
- Standard tool in any development team's toolbox
- Unit tests to ensure code does not regress to a prior state of instability
- Lots of great tools (especially in the CI/CD chain) for ensuring tests are passing before deployment



Putting it All Together

Why not take the problem of security regression and use all of the tools already built for regression testing to improve the security posture of tested applications?

The Demo Application



https://github.com/lavalamp-/security-unit-testing

- Street Art Around the World!
- Written in Django (standard framework, no API, full post-back)
- Same techniques work for any programming language and framework that support introspection
- These examples require a framework that has explicit URL mapping

Dynamic Generation

Django Registered Routes

- Django requires users to write views and then explicitly map these views to URL routes where they are served from
- Views come from a set of pre-defined base classes that support default functionality (UpdateView, DeleteView, DetailView, FormView, etc)

```
urlpatterns = [

    # Admin

    url(r"^admin/", admin.site.urls),

    # Posts

    url(r"^$", views.PostListView.as_view(), name="post-list"),
    url(r"^my-posts/?$", views.MyPostsListView.as_view(), name="my-posts"),
    url(r"^new-post/?$", views.CreatePostView.as_view(), name="new-post"),
    url(r"^view-post/(?P<pk>[-\w]+)/?", views.PostDetailView.as_view(), name="view-post"),
    url(r"^edit-post/(?P<pk>[-\w]+)/?", views.EditPostView.as_view(), name="edit-post"),
    url(r"^delete-post/(?P<pk>[-\w]+)/?", views.DeletePostView.as_view(), name="delete-post
    url(r"^post-successful/(?P<pk>[-\w]+)/?", views.SuccessfulPostDetailView.as_view(), nan

    # Authentication

    url(r"^login/?$", auth_views.login, {"template_name": "pages/login.html"}, name="login"
    url(r"^logout/?$", auth_views.logout, {"template_name": "pages/logout.html"}, name="log
    url(r"^register/?$", views.CreateUserView.as_view(), name="register"),
    url(r"^register-success/?$", views.CreateUserSuccessView.as_view(), name="register-suc

    # Error Handling

    url(r"^error-details/?$", views.ErrorDetailsView.as_view(), name="error-info"),

]
```

Testing Registered Routes

```
class BaseRequestor(object):
    """
    This is a base class for all requestor classes used by the Street
    """

    # Class Members

    requires_auth = False
    supported_verbs = []

    # Instantiation

    # Static Methods

    # Class Methods

    # Public Methods

    def get_delete_data(self, user="user_1"):...

    def get_get_data(self, user="user_1"):...

    def get_patch_data(self, user="user_1"):...
```

- We can use introspection to enumerate all of the views registered within an application
- Now that we know the views, how can we support testing functionality that issues requests to all of the view functionality?
- Enter the **Requestor** class

Requestor Registry Architecture

- Requestors mapped to views they are meant to send requests to via Python decorators
- Singleton registry contains mapping of views to requestors
- Importing all of the views automatically establishes all of the mappings

```
@requested_by("streetart.tests.requestors.pages.CreatePostViewRequestor")
class CreatePostView(BaseFormView):
    """
    This is a view for creating new street art posts.
    """

    template_name = "pages/new_streetart_post.html"
    form_class = NewStreetArtPostForm

    def form_valid(self, form):
        """
        Handle the processing of the uploaded image to S3 and add the expected fields to th
        StreetArtPost object.
        :param form: The form that was submitted.
        :return: The HTTP redirect response from super.form_valid.
        """
        try:
            image_processor = ImageProcessingHelper.from_in_memory_uploaded_file(form.files
        except InvalidImageFileException as e:
            raise ValidationError(e.message)
        latitude, longitude = image_processor.coordinates
        post_uuid = str(uuid4())
        s3_helper = S3Helper.instance()
        response = s3_helper.upload_file_to_bucket(
            local_file_path=form.files["image"].temporary_file_path(),
            key=post_uuid,
            bucket=settings.AWS_S3_BUCKET,
        )
        if response["ResponseMetadata"]["HTTPStatusCode"] != 200:
            raise ValueError("Unable to upload image file to Amazon S3.")
        user = self.request.user if self.request.user.is_authenticated else None
        self._create_object(
            latitude=latitude,
            longitude=longitude,
```

Dynamic Test Generation

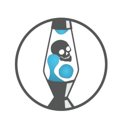
```python
def __get_csrf_enforcement_tests(self):
    """
    Get a list of test cases that check to make sure that CSRF checks are being correctly
    enforced.
    :return: A list of test cases that check to make sure that CSRF checks are being corre
    enforced.
    """
    to_return = []
    csrf_verbs = [x.lower() for x in self.CSRF_VERBS]
    for _, _, callback in self.url_patterns:
        view, requestor = self.__get_view_and_requestor_from_callback(callback)
        supported_verbs = [x.lower() for x in requestor.supported_verbs]
        supported_csrf_verbs = filter(lambda x: x in csrf_verbs, supported_verbs)
        for supported_csrf_verb in supported_csrf_verbs:

            class AnonTestCase1(CsrfEnforcementTestCase):
                pass

            to_return.append(AnonTestCase1(view=view, verb=supported_csrf_verb))
    return to_return

def __get_dos_class_tests(self):
    """
    Get a list of test cases that will test to ensure that all of the configured URL route
    return successful HTTP status codes.
    :return: A list of test cases that will test to ensure that all of the configured URL
    return successful HTTP status codes.
    """
    to_return = []
    for _, _, callback in self.url_patterns:
        view, requestor = self.__get_view_and_requestor_from_callback(callback)
        for supported_verb in requestor.supported_verbs:
```

- We now can enumerate all of the views and access classes that are designed to submit requests to the views
- With this capability we can dynamically generate test cases for all of the views in an application
- Test cases take view classes and HTTP verbs as arguments to constructors

Testing for Requestors

If we are relying on requestor classes being defined for all views, then let’s test for it!

```
def runTest(self):
    """
    Tests that the given view has a requestor mapped to it.
    :return: None
    """
    registry = TestRequestorRegistry.instance()
    self.assertTrue(
        registry.does_view_have_mapping(self.view),
        "No requestor found for view %s." % self.view,
    )
```



Testing for Denial of Service

We've got the ability to test every known HTTP verb of every registered view, so let's test for successful HTTP responses.

```python
def runTest(self):
    """
    Tests that the given view returns a successful HTTP response for the given verb.
    :return: None
    """
    requestor = self._get_requestor_for_view(self.view)
    response = requestor.send_request_by_verb(self.verb, user_string="user_1")
    self._assert_response_successful(
        response,
        "%s did not return a successful response for %s verb (%s status, regular user)"
        % (self.view, self.verb, response.status_code)
    )
```

Testing for Unknown Methods

Test to ensure that the methods supported by requestors match the methods returned by OPTIONS request.

```
def runTest(self):
    """
    Tests that the HTTP verbs returned by an OPTIONS request match the expected values.
    :return: None
    """
    requestor = self._get_requestor_for_view(self.view)
    response = requestor.send_options(user_string="user_1")
    allowed_verbs = response._headers.get("allow", None)
    if not allowed_verbs:
        raise ValueError("No allow header returned by view %s." % self.view)
    allowed_verbs = [x.strip().lower() for x in allowed_verbs[1].split(",")]
    supported_verbs = [x.lower() for x in requestor.supported_verbs]
    self.assertTrue(
        all([x.lower() in supported_verbs for x in allowed_verbs]),
        "Unexpected verbs found for view %s. Expected %s, got %s."
        % (self.view, [x.upper() for x in supported_verbs], [x.upper() for x in allowed_verbs])
    )
```

Testing for Auth Enforcement

- Tell the requestors whether or not the tested view requires authentication
  - Can improve upon this demo by checking for inheritance of the **LoginRequiredMixin**
    - Check that unauthenticated request is denied

```
def runTest(self):
    """
    Tests that the given view returns the expected HTTP response value when an unauthenticated
    request is submitted to it.
    :return: None
    """
    requestor = self._get_requestor_for_view(self.view)
    response = requestor.send_request_by_verb(self.verb, do_auth=False)
    self._assert_response_redirect(
        response,
        "Response from unauthenticated %s request to view %s was %s. Expected %s."
        % (self.verb, self.view, response.status_code, [301, 302])
    )
```

Response Header Inclusion

```
def runTest(self):
    """
    Tests that the HTTP response received from the view contains a header corresponding to
    self.header_key.
    :return: None
    """
    requestor = self._get_requestor_for_view(self.view)
    response = requestor.send_request_by_verb(self.verb, user_string="user_1")
    self._assert_response_has_header_key(
        response=response,
        header_key=self.header_key,
        message="Response from view %s with verb %s did not contain header key of %s. Keys were %s."
        % (self.view, self.verb, self.header_key, response._headers.keys())
    )
```

```
def runTest(self):
    """
    Tests that the HTTP response received from the view contains a header key and value corresponding
    to self.header_key and self.header_value.
    :return: None
    """
    requestor = self._get_requestor_for_view(self.view)
    response = requestor.send_request_by_verb(self.verb, user_string="user_1")
    self._assert_response_has_header_value(
        response=response,
        header_key=self.header_key,
        header_value=self.header_value,
        message="Response from view %s with verb %s did not contain expected header value of %s: %s. "
                "Headers were %s."
        % (self.view, self.verb, self.header_key, self.header_value, response._headers)
    )
```

Testing for OPTIONS Accuracy

We already built out requestors based on the OPTIONS response, so now let's make sure that the OPTIONS response included the correct HTTP verbs.

```
def runTest(self):
    """
    Tests that self.verb does not work against self.view.
    :return: None
    """
    requestor = self._get_requestor_for_view(self.view)
    response = requestor.send_request_by_verb(self.verb, user_string="user_1")
    self._assert_response_not_allowed(
        response,
        "HTTP verb %s returned %s status code when it should have been 405 (regular user)."
        % (self.verb, response.status_code)
    )
```

Testing for CSRF Enforcement

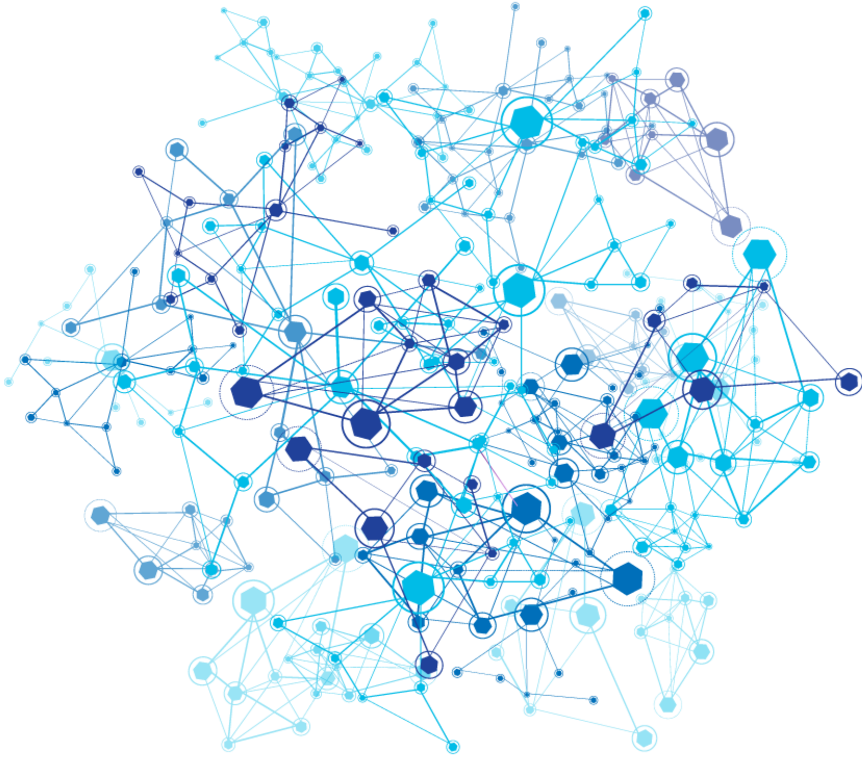
Test to ensure that CSRF tokens are required for function invocation on non-idempotent view functionality.

```
def runTest(self):
    """
    Test to ensure that that the CSRF token is properly enforced on the referenced view.
    :return: None
    """
    requestor = self._get_requestor_for_view(self.view)
    response = requestor.send_request_by_verb(
        self.verb,
        user_string="user_1",
        enforce_csrf_checks=True,
    )
    self._assert_response_permission_denied(
        response,
        "Response from %s indicated that CSRF protection was not enabled (verb was %s, status %s)."
        % (self.view, self.verb, response.status_code)
    )
```

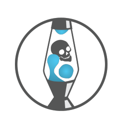
What Have We Gained?

- We now have guarantees that
  - Our app contains no hidden functionality
  - All of our views are working as intended given expected input
  - Authentication is being properly enforced
  - Security headers are present
  - CSRF is properly protected against

Why Dynamic Generation?



- Those guarantees are great and all, but can’t we just write individual unit tests to test for them?
- In a development team we have multiple people contributing code all the time
- Through dynamic generation, these tests will automatically be applied to all new views, providing the same guarantees to code that **hasn’t even been written yet**



Where Can We Go?

- Other things that we could write dynamic tests for
  - Rate-limiting
  - Fuzzing of all input values to POST/PUT/PATCH/DELETE (introspection into forms used to power the views)
  - Proper updating, creation, and deletion of new models based on input data

Testing Other Vulns

Testing for Cross-site Scripting

Test for proper encoding of output data!

```
ENCODE_REGEX = re.compile("\[SHOULDENCODE\](.*?)\[/SHOULDENCODE\]")
XSS_ENCODE_CHARS = "<>'\"&"
XSS_ENCODED_OUTPUT = "&lt;&gt;&#39;&quot;&amp;"

def runTest(self):
    """
    Test to ensure that the error details page is not vulnerable to cross-site scripting via its
    error query string parameter.
    :return: None
    """
    data = {
        "error": "[SHOULDENCODE]%s[/SHOULDENCODE]" % (self.XSS_ENCODE_CHARS,),
    }
    response = self.client.get("/error-details/", data=data)
    encoded_data = self.ENCODE_REGEX.findall(response.content)[0]
    self.assertEqual(
        encoded_data,
        self.XSS_ENCODED_OUTPUT,
        "XSS encoding characters (%s) were not properly encoded in response (%s)."
        % (self.XSS_ENCODE_CHARS, encoded_data)
    )
```

Testing for SQL Injection

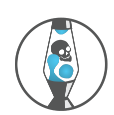
Submit two requests to the server, one making the SQL query match none and another making the SQL query match all, test to see if the results match the **none** and **all** expected responses

```
POST_COUNT_REGEX = re.compile("title-row")

def runTest(self):
    """
    Tests to ensure that the SQL injection vulnerability found in the GetPostsByTitleView view is not
    present.
    :return: None
    """
    post = StreetArtPost.objects.first()
    total_posts = StreetArtPost.objects.count()
    none_url_path = "/get-posts-by-title/?title=%s' and 1=2--" % post.title
    all_url_path = "/get-posts-by-title/?title=%s' or 1=1--" % post.title
    none_response = self.client.get(none_url_path)
    all_response = self.client.get(all_url_path)
    none_count = len(self.POST_COUNT_REGEX.findall(none_response.content))
    all_count = len(self.POST_COUNT_REGEX.findall(all_response.content))
    self.assertFalse(
        all([none_count == 0, all_count == total_posts]),
        msg="Response from GetPostsByView indicates SQL injection present. %s entries returned for "
            "match none, and %s entries returned for match all."
        % (none_count, all_count)
    )
```

Testing for Open Redirects

Submit malicious input and see if HTTP redirect response redirects to full URL

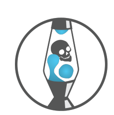
```python
def runTest(self):
    """
    Tests to ensure that the RedirectView view is not vulnerable to open redirects.
    :return: None
    """
    target_url = "http://www.google.com"
    response = self.client.get("/redirect/?redirect=%s" % target_url)
    self.assertNotEqual(
        response["location"],
        target_url,
        msg="Redirect allowed for open redirect to target URL of %s."
        % (target_url,)
    )
```



Conclusion

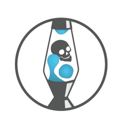
Benefits of Dynamic Generation

- Initial overhead is greater than writing individual unit tests, but new views added to the application also benefit from the tests
- Provide us with strong guarantees about known application functionality and basic HTTP-based security controls



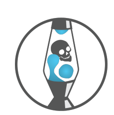
Benefits of Sec. Unit Testing

- Security guarantees now enforced by CI/CD integration
- Test Driven Development? **Great** – have your security testers write failing unit tests that you then incorporate into your test suite
- A new interface for how security and development teams can work together in harmony



Recap

- Security regression is a big problem
- We can use the development paradigm of regression testing to address security regression
- Dynamic test generation can take us a long way
- Individual tests for individual cases further augment dynamic test generation capabilities



Resources

- Security Unit Testing Project
  https://github.com/lavalamp-/security-unit-testing
- Lavalamp's Personal Blog
  https://l.avala.mp/
- Django Web Framework
  https://www.djangoproject.com/

THANK YOU!

@_lavalamp
chris [AT] websight [DOT] io
github.com/lavalamp-