Pythia: Identifying Dangerous Data-flows in Django-based Applications

Linos Giannopoulos
Greek Research and Technology Network
lgian@noc.grnet.gr

Panayiotis Tsanakas
National Technical University of Athens
panag@cs.ntua.gr

Eirini Degkleri
Greek Research and Technology Network
degleri@noc.grnet.gr

Dimitris Mitropoulos
Greek Research and Technology Network
dimitro@grnet.gr

ABSTRACT
Web frameworks that allow developers to create applications based on design patterns such as the Model View Controller (MVC), provide by default a number of security checks. Nevertheless, by using specific constructs, developers may disable these checks thus re-introducing classic application vulnerabilities such as Cross-site Scripting (XSS) and Cross-Site Request Forgery (CSRF). Framework-specific elements including (1) the nature of these applications, (2) the different features that they involve (e.g., templates), and (3) the inheritance mechanisms that govern them, make the identification of such issues very difficult.

To tackle this problem, we have developed Pythia, a scheme that analyzes applications based on the Django framework. To identify potentially dangerous data flows that can lead to XSS and CSRF defects, Pythia takes into account all the aforementioned elements and employs ideas coming from standard data-flow analysis and taint tracking schemes. To the best of our knowledge, Pythia is the first mechanism to consider framework-specific elements in its analysis. We have evaluated our scheme with positive results. Specifically, we used Pythia to examine five open-source applications that are currently in production and have thousands of users including an e-voting service, and a web-based translation management system. In four cases we have identified dangerous paths that in turn led to vulnerabilities. Notably, in many cases the paths involved the particular features of Django-based applications e.g. templates.

CCS CONCEPTS
- Security and privacy → Web application security; • Software and its engineering → Software defect analysis.

KEYWORDS
Data-flow Analysis, Application Security, Templates, Unsanitized Output, Cross-site Scripting, Cross-Site Request Forgery, Django

1 INTRODUCTION
Development frameworks based on existing programming languages and architectural patterns, have become one of the most widespread vehicles to create web applications. Such frameworks include by default security features to assist developers write secure code without introducing vulnerabilities such as SQL injection, Cross-Site Scripting (XSS) and Cross-Site Request Forgery (CSRF). In this way, developers do not have to re-invent the wheel every time they needed to examine user input or sanitize the output of their application.

However, on many occasions developers need to disable such features to allow for specific functionalities. This can re-introduce the aforementioned vulnerabilities. In addition, the complex nature of the frameworks [22] and the various features that they offer can make the identification of such defects a difficult task.

Django is a popular Python-based web framework that follows the Model Template View (MTV) design pattern, an adaptation of the Model View Controller (MVC) architectural pattern [10, 13]. In a Django-based application there is an object-relational mapper (ORM) that mediates between data models and a relational database (model). There is also a system for processing HTTP requests with a web templating system (view), and a regular-expression-based URL dispatcher. Finally, both patterns provide several features that can be used to enhance an application such as templates and decorators.

A potential risk emerges when developers disable the security checks that the framework provides especially within templates and decorators. Such cases are difficult to identify because of the inheritance mechanisms supported by the frameworks, e.g. a template inherits another with disabled security checks. Notably, existing mechanisms are not designed to detect such issues.

To address such problems, we have developed Pythia, a mechanism that analyzes Django-based applications to identify potentially dangerous data paths. To do so, it checks if dangerous code constructs (i.e. the ones used to bypass the security checks provided by

1 In ancient Greece, Pythia was the name of the high priestess of the Temple of Apollo at Delphi who also served as the oracle.
the framework) are included in an application. Then, by performing data-flow analysis it examines all critical application parts to identify if any data that may incorporate user input reaches the constructs identified in the first step. Note that, a reported alert does not necessarily involve an existing hazard. However, Pythia’s alerts can be helpful to developers in the future as we further explain in our ‘Evaluation’ section (4).

Our mechanism borrows some of the standard ideas (Abstract Syntax Tree (AST) analysis) and terms (sources and sinks) of other data-flow analysis [12, 14, 16] and taint tracking schemes [23, 28, 30] employed to identify web application defects. Nevertheless, it goes one step further and takes into account the complicated architecture and the various mechanisms (inheritance) and features (templates) of Django-based applications. To explore paths in such applications Pythia follows a novel way of analysis that can be applied in other similar contexts such as Laravel [24], an mvc-based framework used to create applications in PHP.

We have evaluated Pythia by examining five, large, open-source applications with thousands of users including an e-voting service [29], a cloud service, and a web-based translation management system [11]. In four cases we have identified corresponding vulnerabilities and notified the developers. In all cases the applications were patched [8]. Apart from the straightforward xss vulnerabilities (where user input may reach a sink in a view), Pythia identified many paths that involved the particular features of Django-based applications such as templates.

2 VULNERABILITY INHERITANCE IN DJANGO-BASED APPLICATIONS

Development frameworks enforce security mechanisms such as the sanitization of HTML output by default. In this way, certain characters (e.g. ‘<’ and ‘>’) are not interpreted as special control characters. However, the frameworks also allow programmers to bypass such mechanisms to support specific use cases which in turn, can lead to serious vulnerabilities including XSS and CSRF.

A serious issue emerges when this bypassing happens in *templates*, a feature that provides a way to generate HTML dynamically. Templates are supported by several frameworks including Django and Laravel. In general, templates contain static parts of the desired HTML output, together with some special syntax describing how the dynamic content will be inserted.

When developers insert HTML content into a template, they need to mark it as “safe”, so that the framework itself treats it as such, and avoid escaping HTML characters. In this case developers have to be sure about the origins of the content because if the content includes user input, attackers can perform an XSS attack.

To track such a vulnerability is not trivial because templates can either include or inherit other templates. Hence, it is difficult to find which views are affected by a potentially vulnerable template. Figure 1 highlights this issue, i.e. a template that marks content as safe is included by multiple other ones within the application. In turn, these templates are used by different views that process content either from user input, or loaded from the model.

Another issue occurs when developers use the @csrf_exempt decorator to disable CSRF protection in Django views [27]. Specifically, Django allows developers to inject a CSRF token for every form rendered and then expect the client to supply this token with the corresponding POST request. This mechanism is disabled when the @csrf_exempt feature is used.

To identify the aforementioned issues during a code review, security experts have to find all the occurrences of the different features and track which views are affected manually. Such a task could even be performed using grep command. However this would require an exhaustive and repetitive search from the security expert’s side.

Additionally, tracking the various intermediate entities that inherit such properties is not an easy task in modern, large applications. Thus, some level of automation is needed to cope with large code-bases. Our approach provides a way to automatically discover data flows when such features are employed and warn security engineers about potentially vulnerable paths.

3 APPROACH

This work describes Pythia, an approach that analyzes applications, to identify data-paths which involve dangerous constructs such as the ones described in Section 2. To do so, Pythia analyzes an application’s views and templates, leaving out models as they do not hold any relevant information. Figure 2 presents the basic steps performed by our approach: First, Pythia searches for specific constructs, which we call sinks, marking any affected templates and then, examines views to identify (1) if untrusted data can reach the elements identified in the first step, (2) if other sinks are being used by the views.

3.1 Sinks

A sink method depicts a coding construct where the hazard might take place [20]. In our case, a sink involves an invocation that bypasses the default security mechanisms of a Django Application. In such an application we can divide sinks into two categories, namely: in-view sinks, and template sinks. The first category only
2.1) analyze Views to find which Templates are used and 2.2) search for more "sinks" utilized directly by a View.

Algorithm 1

```
1. INPUT TS: template sinks
2. INPUT IS: in-view sinks
3. INPUT url_conf: the url pattern terms object containing all URLs
4. function ANALYZE(TS, IS, url_conf)
   5. S ← set();
   6. T ← getTemplates();
   7. paths[];
   8. for all t ∈ T do
      9. t.walk(TS, paths[], t.root, t.name);
   10. V ← getAllViews(url_conf);  
      11. for all v ∈ V do
         12. if v.invokes(IS) then
            13. S.add(v);
         14. for all p ∈ paths[] do
            15. if v.renders(p) then
               16. S.add(v, p);
```

Figure 2: In the Model Template View (MTV) Context, Pythia focuses on Views and Templates.

3.2 Application Analysis

Algorithms 1 and 2 illustrate how Pythia examines a Django application. First, our approach retrieves all project templates and generates their corresponding Abstract Syntax Trees (ASTs). Then, starting from the root node of each AST, it recursively traverses all children nodes searching for variables that reach a sink method. This process starts in line 9 of Alg. 1 and continues in Alg. 2. Specifically, when Pythia identifies a sink, it marks the template as potentially dangerous (Alg. 2, line 9). If the current template extends or includes another one, the approach goes on to examine the other template too (Alg. 2, lines 10-13). In this way it creates a path that is recorded when a sink is found. When a dangerous path is found, the current node and the template ancestry are kept in a key-value store in order to be cross-examined in the second stage.

When the first part finishes, Pythia analyzes the views of the application and extracts all decorators. Then, it traverses the AST of each View to identify the templates that are using the data that each template requires to render the page. Finally, the results are being cross-examined in order to find unsafe data paths.

3.3 Resolution of Paths

The output of Pythia is based on the category of the sink identified. In the case of a decorator (i.e., an in-view sink such as csrf_exempt), the output contains: (1) the URL under which the potentially vulnerable view resides, (2) the module where the view is defined (the last component of the module is the view’s name), and (3) which decorators were used on it.

If a dangerous function is identified within a view, Pythia returns (1) the view’s corresponding URL (e.g., /users/<id>/), (2) the module where the view is defined alongside with the line under where the function was invoked, (3) the function’s name, and (4) the variable it was ran upon.

For template sinks, Pythia’s output includes all views that render, directly or indirectly, a template that uses the list of unsafe filters and tags. Also, it contains the module and the line where the view is defined, the ancestry path to reach the dangerous template, the filter that was applied, the variable itself, and the variable’s context.

Note that the context information is needed to track the data provided from the view to the template until they reach the sink.

Furthermore, in case a specific template containing sinks does not resolve to a view, the path to the sinks is still part of the output.

Algorithm 2 Exploring Paths

```
1. INPUT TS: template sinks
2. INPUT paths[]: a list holding all potentially tainted paths
3. INPUT node: current AST node
4. INPUT origin: template ancestry
5. function WALK(TS, paths[], node, origin)
   6. for all subnode ∈ node.nodeList do
      7. if subnode.isVariable() then
         8. if subnode.invokes(TS) then
            9. paths[] ← subnode.origin;
         10. else if subnode.isExtendsNode() then
            11. walk(TS, paths, subnode, subnode.extended_tmplt);
         12. else if subnode.isIncludeNode() then
            13. walk(TS, paths, subnode, subnode.included_tmplt)
```

The term context is used to describe all variable aliasing / assignments within the current scope, until a dangerous filter invocation is reached.
In this manner, a security expert can manually resolve the given template until the source of the data is found. An example where the full path is not resolved is when a template is used for the rendering of a form inside a template. Currently, Pythia does not go through form definitions to find template invocations.

3.4 Output Filtering

Pythia outputs a number of potentially dangerous data paths. This means that a part of a path may appear in other several paths. In addition, a template that marks output as safe may be included by multiple templates. In such occasions the output of the tool may include recurring information with duplicated elements.

To reduce the volume of such instances we apply filters that make the output more comprehensive and easy to use. For instance, we enumerate the views that end up using a template including safe tags and let the user explore their names as a second step.

3.5 Implementation Details

As we described earlier, Pythia requires the ast of a template. To generate this ast, Django requires a running environment. This is because Django needs to resolve user-defined components (e.g. custom filters) and its own defaults (e.g. for loops, variable aliasing). Other operations such as the analysis of a view’s module are realized statically using Python’s AST package.

This design decision, to divide the implementation between purely static and processes that require Django’s run-time to be setup has certain assets and liabilities. Using a purely static approach, one can easily use the tool on many projects without having to configure their dependencies and even run it during a project’s continuous integration as another security test. However, with the hybrid approach one can gain the advantage of having the ability to use certain aspects of Python’s and Django’s run-time and processes that require Django’s run-time to be configured. One can easily use the tool on many projects without having to configure their dependencies and even run it during a project’s continuous integration as another security test. However, with the hybrid approach one can gain the advantage of having the ability to use certain aspects of Python’s and Django’s run-time and exploring paths that otherwise would not be possible to explore.

Regarding further implementation details, Pythia supports both Python 2.x and 3.x versions and Django 1.8.x until 1.11.x. Versions before 1.7.x are not supported as they have a vastly different internal representation of the AST node classes.

4 EVALUATION

We have applied Pythia on five different open-source, Django projects. Note that many Django projects involve Jinja [26] templates and Pythia does not currently analyze such templates. Hence, we focused on projects that only employ Django templates.

In four projects we found vulnerabilities. We have reported the defects to the developers of each project and in all cases patches were introduced. Notably, all applications have thousands of users.

Table 1 presents the results of our evaluation. Specifically, we show the number of each vulnerability that Pythia identified for each application, and provide a high-level description of the impacts. In the following, we provide further details regarding the applications and the defects we found. Pythia did not report any defects for Misago [1], a forum application written in Python.

4.1 ~okeanos

~okeanos [5] is an an IaaS (Infrastructure as a Service) that has been in production since early 2012. It offers cloud computing services to the Greek and the European research and academic community, and involves thousands of users.

In the case of ~okeanos, Pythia identified an xss vulnerability very similar to the one described in Section 2. Specifically, one of the application’s views includes a template named project_application_form.html:

```python
{% for field in form %}
  (% if field.name in filter_fields %)
    <div class="form-row">
      (% if field.errors | length %)with-errors(% endif %)
      (% if field.is_hidden %)with-hidden(% endif %)
    (field.errors)
    <p class="clearfix">
      (% if field.blank %)required(% endif %)"
        (field.label_tag)
    (field|safe))
    <div class="form-group">
      (% if field.errors | length %)with-errors(% endif %)
      {% if field.errors %}
        <div class="invalid-feedback">
          (% if field.errors | length %)with-errors(% endif %)
          <strong>Error</strong>: (% for error in field.errors %)
            (% error %)
          (% endfor %)
        </div>
        </div>
      (% endif %)
    </div>
  (% endif %)
{% endfor %}

The sink occurs in form_field.html where every field of a form is marked as safe for output (line 14):

```python
{% if field.name in filter_fields %}
  <div class="form-row">
    (% if field.errors | length %)with-errors(% endif %)
    (% if field.is_hidden %)with-hidden(% endif %)
  (field.errors)
  <p class="clearfix">
    (% if field.blank %)required(% endif %)"
      (field.label_tag)
  (field|safe))
  <span class="form-control">
    (% if field.errors | length %)with-errors(% endif %)
    {% if field.errors %}
      <div class="invalid-feedback">
        (% if field.errors | length %)with-errors(% endif %)
        <strong>Error</strong>: (% for error in field.errors %)
          (% error %)
        (% endfor %)
      </div>
      </div>
    (% endif %)
  </div>
{% endif %}

The above can lead to an xss vulnerability [20], i.e. attackers can inject malicious scripts that are first stored on the database of the application and then run in the browser of other users (including administrators), thus stealing their cookies.

4.2 Zeus

Zeus [9, 29] is an e-voting application that has been in production since late 2012 and has been used in more than 500 real-world elections involving more than 55,000 voters [17].

Table 1: Evaluation Results

<table>
<thead>
<tr>
<th>Application</th>
<th>Version</th>
<th>XSS Impacts</th>
<th>CSRF Impacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>~okeanos</td>
<td>-</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Zeus</td>
<td>-</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>ViMa</td>
<td>2.2</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>Weblate</td>
<td>3.3</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
The XSS vulnerability that was discovered in Zeus is very similar to the one described earlier. An interesting observation was that the view that included the template with the sink, was not accessible through the application’s GUI (Graphical User Interface). However, after examining the code of the application, attackers could reach the view through a specific URL. Hence, a vulnerability scanner that examines a running instance of the application would not discover the vulnerability. Contrary to such scanners, Pythia’s ability to map URLs to views (as described in Subsection 3.5) led to the identification of the dangerous path.

Apart from the XSS defect, Pythia identified two CSRF vulnerabilities: one that can be employed to reset passwords and another that can be a denial of service attack vector. Specifically, in the application version that we examined, the password reset functionality was accessed through a GET request from a specific URL which incorporated the to (i.e. a number) of a user as a parameter. A valid attack scenario in this case, would be to host a website that sends AJAX requests to the application to reset every password. Given that the users’ IDs are stored in a sequential way in the database this is easy to accomplish. This would cause organizational mess during an election since the administrator or anyone else would not be able to log-in.

Missing CSRF protection on another POST form could lead to a Denial of Service (DoS) attack through a CSRF attack. In particular, attackers could host a website, lure victims to it, and submit the form multiple times through the victim’s browser. The above would lead to resource drain due to the required computing resources needed by the attacked endpoint.

4.3 ViMa

ViMa (Virtual Machines) [7], provides to the Greek academic community access to shared computing and network resources that can be used for production purposes. Currently, it is used by a hundreds of researchers and practitioners.

Pythia identified numerous @csrf_exempt decorators that can lead to 9 types of CSRF attacks. Through these attacks an attacker can perform different actions including: rebooting a virtual machine of another user, shutting it down and destroying it. This attack requires that the victim is lured to browse to a malicious website that performs these actions using the victim’s credentials, leveraging the lack of CSRF protection. For instance, the following decorator allows an attacker to reboot a virtual machine of another user:

```python
@check_admin_lock
@login_required
@csrf_exempt
@check_instance_auth
def reboot(request, cluster_slug, instance):
    cluster = get_obj_or_404(Cluster, slug=cluster_slug)
    [...]
```

4.4 Weblate

Weblate [11] is a web-based translation management system. It is used for translating many free software project as well as commercial ones, including PHPMyAdmin [2] and Debian Handbook [6].

In the case of Weblate, user data stored in the database were being marked as safe even before being sent to the view’s corresponding template (in-view sink). In particular, the name of a project is provided by a user. Then, this information is marked as safe by the application and is included on the output. This issue led to stored XSS attacks that could be used to elevate any user’s privileges to the ones of an administrator.

4.5 False Positives and Negatives

A potentially dangerous path may involve a source that will not include user data e.g. a variable that is instantiated by retrieving data from the database and in turn the data are auto-generated by the application. Pythia will report the path above producing a false alarm. An alarm like this was produced when we examined ~oceanos. Specifically, the path involved the terms and conditions of ~oceanos which are marked as safe by the template when they are displayed to the user. However, such information is not useless. After seeing such a path, a developer can note it down and be careful in the future regarding the identified sink.

In general, false positives in related mechanisms (either data-flow analysis or taint tracking techniques) is a known issue [18, 20]. Pythia though should be seen as an assistant for developers who want to have a thorough security perspective of their application, and not as a vulnerability detector.

A false negative may appear in the following way: consider a mark_safe construct that exists in a function invoked by a view. Pythia will not report this case because it does not examine the code of the invoked functions. However, we can perform a more in-depth analysis and include such cases too.

5 RELATED WORK

Pythia is related to various approaches that have been developed to identify web application vulnerabilities either through static data-flow analysis or dynamic taint tracking. We describe how each one operates, and outline its main characteristics.

Livshits et al. [16] statically analyzes the CFG (Control Flow Graph) of a Java application to identify variables that carry user input and check if they reach a sink method. In this way they detect possible SQL injection and XSS defects. Pixy [14] is an open source tool that examines PHP scripts in a similar manner. In Pixy, rules can appear in the source of the program in the form of annotations. Dahse and Holz [12] have introduced a refined type of data-flow analysis to identify second-order vulnerabilities. Note that vulnerabilities of this kind occur when an attack payload is first stored by the web application on the server-side and then later on used in a security-critical operation.

Vogt et al. [30] have developed a taint tracking scheme which operate on the server-side at runtime. Their mechanism tracks sensitive information at the client-side to ensure that a script can send sensitive user data only to the site from which it came from. In this way they prevent XSS attacks that may attempt to send the user’s cookie to a malicious site. Stock et al. [28] use tainting to prevent DOM-based XSS attacks. Specifically, it employs a taint-enhanced JavaScript engine that tracks the flow of malicious data. To prevent an attack, the scheme uses HTML and JavaScript parsers that can identify the generation of code which in turn comes from tainted data. WASP [21] and PHP Aspis [23] apply further checks when it is established that tainted data have reached a sink. In

---

1. https://github.com/WeblateOrg/weblate/blob/d84dfbdfab0f3d158f5d39293e6d75903e444/ weblate/trans/views/basic.py#L91-L129
particular, WASC analyzes HTML responses to examine if there is any data that contains malicious scripts to prevent XSS attacks. PHP Aspis, associates tainted data with metadata such as the propagation path to prevent SQL injection and XSS attacks in PHP applications.

There is a number of static analysis mechanisms that are used to analyze Python programs and applications that are also related to Pythia. Libraries such as PyLint [3], Flake8 [25], and Django Lint [15] are popular quality checkers that report common programming errors such as code smells and provide simple refactoring suggestions. Security wise, the most widely used python static analyzer is Bandit [4], which is being actively maintained. However, it does not provide great insights when analyzing large Django projects. This is because it analyzes ASTs searching for dangerous code constructs without any further analysis. As a result, it generates numerous visualizations based on the tool’s output and the application of developers and security experts to identify such issues within large code bases and in an easy manner. Furthermore, our evaluation results indicate that Pythia can be used to examine applications in production and with thousands of users.

Pythia is the first mechanism that takes into account features such as templates and their inheritance. Notably, the approach behind Pythia can be applied to different contexts such as the Laravel framework which runs on MVC-based applications written in PHP.

Future work on our mechanism involves the creation of visualizations based on the tool’s output and the application of further checks when a dangerous path is identified in a way similar to WASC [21] and PHP Aspis [23] (see Section 5).

6 CONCLUSIONS

Modern web frameworks intend to assist developers by enabling security features by default. Nevertheless, the negligent bypassing of these measures can compromise the security of an application. Furthermore, the complex architecture of MVC-based applications makes it hard to identify such hazards. Pythia, aims to assist developers and security experts to identify such issues within large code bases and in an easy manner. Furthermore, our evaluation results indicate that Pythia can be used to examine applications in production and with thousands of users.

Acknowledgements. This work was supported by the European Union’s Connecting Europe Facility (CEF) Work Program 2016 under grant agreement INEA/CEF/ICT/A2016/1332498 (project CERTCOOP).

REFERENCES


L. Giannopoulos et al.