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

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

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

Panayiotis Tsanakas
National Technical University of Athens
panag@cs.ntua.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 complex nature of these applications, (2) the different features that they involve (e.g. templates), and (3) the inheritance mechanisms that governs 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 four 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 all 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

**ACM Reference Format:**
Linos Giannopoulos, Eirini Degkleri, Panayiotis Tsanakas, and Dimitris Mitropoulos. 2019. Pythia: Identifying Dangerous Data-flows in Django-based Applications. In *12th European Workshop on Systems Security (EuroSec '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3301417.3312497

## 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 [20] 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 [8, 11]. 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 these features. In addition, such issues are difficult to identify because of inheritance mechanisms supported by the

*EuroSec '19, March 25–28, 2019, Dresden, Germany*

ACM ISBN 978-1-4503-6274-0/19/03.
https://doi.org/10.1145/3301417.3312497

frameworks, e.g. a template inherits another one with disabled security checks. Notably, existing mechanisms are not designed to detect such dangers.

To address this issue, we have developed *Pythia*,[1] a mechanism that analyzes Django-based applications to identify potentially dangerous data paths. In particular, it checks if dangerous code constructs (i.e. the ones used to bypass the security checks provided by 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, Pythia searches for potentially dangerous data flows. That is, an alert does not necessarily involve an existing hazard. However, such alerts can be helpful to the developer 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 [10, 12, 14] and taint tracking schemes [21, 25, 27] 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* [22], an MVC-based framework used to create applications in PHP.

We have evaluated Pythia by examining four, large, open-source applications with thousands of users including an e-voting service [26], a cloud service, and a web-based translation management system [9]. In all cases we have identified corresponding vulnerabilities and notified the developers. 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.

Three of the applications were patched hence we provide further details about them. Following responsible disclosure practices, we do not reveal the name of the fourth application and we will do so once the issue is fixed.

## 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 [24]. 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*,[2] 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 [18]. In our case, a sink involves an invocation that bypasses the default security mechanisms of a Django

[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.

[2] Pythia will be available as open source code. Locations not shown for double-blind reviewing

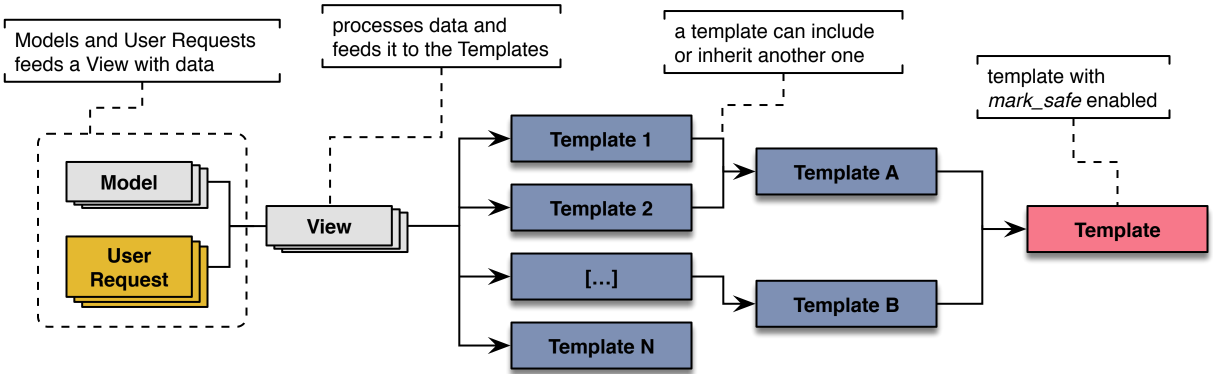
Models and User Requests feeds a View with data
processes data and feeds it to the Templates
a template can include or inherit another one
template with *mark_safe* enabled
Model
User Request
View
Template 1
Template 2
[...]
Template N
Template A
Template B
Template

**Figure 1: A potentially vulnerable Django `template` is included by various templates which in turn are utilized by Django `Views`**

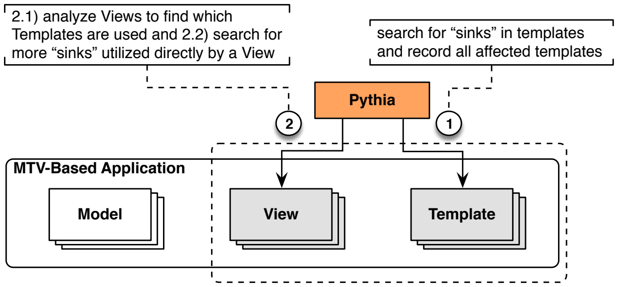
2.1) analyze Views to find which Templates are used and 2.2) search for more "sinks" utilized directly by a View
search for "sinks" in templates and record all affected templates
Pythia
2
1
MTV-Based Application
Model
View
Template

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

Application. In such an application we can divide sinks into two categories, namely: *in-view sinks*, and *template sinks*. The first category only affects a particular `View`, while the second on involves templates that may affect all the views that use them.

*Template Sinks* involve Django filters such as `safe` and `safeseq`, and tags such as `autoescape`. The `autoescape` tag, controls the auto-escaping effect of a code block. When disabled (set to `off`), it is equivalent to marking all variables inside that code block as "safe". Escaping concerns dangerous characters such as '<', which in turn is converted to '`&lt;`' and more. Furthermore, filters including `safeseq` and `safe` dictate that a variable does not require further HTML escaping before it becomes part of the output.

*In-view Sinks* include *decorators* such as `@csrf_exempt` and functions invocations that can lead to XSS defects when used in an improper manner (e.g. `mark_safe`).

To trace dangerous data flows, Pythia expects as input two lists which in turn include the constructs of each category. The lists, can easily be expanded with more methods depending on the needs of the developer, thus providing flexibility to track other application / project-specific filters or decorators.

When Pythia identifies a sink, it then examines the application for potential sources, i.e. variables that hold data that

**Algorithm 1** Searching for Dangerous Flows

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

can reach the sink. Notably, such data can come from a `Model` or a user request (e.g. `request.GET.get()`). In the upcoming section we discuss how Pythia examines the various data paths of an application.

## 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 potentially 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 being used and the data that each template requires to render the page. Finally, the results are being cross-examined in order to find unsafe data paths.

**Algorithm 2** Exploring Paths

```
INPUT $TS$: template sinks
INPUT $paths[]$: a list holding all potentially tainted paths
INPUT $node$: current AST node
INPUT $origin$: template ancestry
function WALK($TS$, $paths[]$, $node$, $origin$)
    for all $subnode \in node.nodelist$ do
        if $subnode.isVariable()$ then
            if $subnode.invokes(TS)$ then
                $paths[] \leftarrow subnode, origin;$
        else if $subnode.isExtendsNode()$ then
            $walk(TS, paths, subnode, subnode.extended\_template)$
        else if $subnode.isIncludeNode()$ then
            $walk(TS, paths, subnode, subnode.included\_template)$
```

## 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. In addition, 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. [3] 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. 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

As we discussed earlier 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

In the scope of Django applications that Pythia analyzes, this section provides a number of details regarding the tool's implementation and its internals.

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 requirements, 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. In four cases we found vulnerabilities. We have reported the defects to the developers of each project. In all cases except for one patches were introduced, hence we name only the applications that were fixed. 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

[3] The term context is used to describe all variable aliasing / assignments within the current scope, until a dangerous filter invocation is reached.

did not report any defects for Misago [1], a forum application written in Python.

## 4.1 Okeanos

We have analyzed an IaaS (Infrastracture as a Service) application with thousands of users. In this case, 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`:

```
template_name = '/projects/project_application_form.html'
```

At some point, the `View` utilizes the template to render a form that incorporates data stored in the database. However, the data are initially provided by users through another form. In turn, the template above includes another one named `form_field.html`:

```
{% for field in form %}
    {% with filter_fields=details_fields %}
        {% include "/projects/form_field.html" %}
    {% endwith %}
{% endfor %}
```

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

```
1  {% if field.name in filter_fields %}
2  <div class="form-row"
3      {% if field.errors|length %}with-errors{% endif %}
4      {% if field.is_hidden %}with-hidden{% endif %}">
5        {{field.errors}}
6        <p class="clearfix
7           {% if field.blank %}required{% endif %}">
8           {{field.label_tag}}
9           {{field|safe}}
10          <span class="extra-img">&nbsp;</span>
11             {% if field.help_text %}
12                 <span class="info">
13                     <em>more info</em>
14                     <span>{{field.help_text|safe }}</span>
15                 </span>
16             {% endif %}
17          </p>
18     </div>
19 {% endif %}
```

The above can lead to a stored XSS attack [18], 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.

**Table 1: Evaluation Results**

| Application | Version | XSS | | CSRF | |
|---|---|---|---|---|---|
| | | # | Impacts | # | Impacts |
| Okeanos | - | 3 | Privilege Escalation | 0 | - |
| Zeus | - | 1 | Privilege Escalation | 2 | Password Reset & DOS |
| ViMa | 2.2 | 0 | - | 9 | VM Manipulation |
| Weblate | 3.3 | 1 | Privilege Escalation | 0 | - |

## 4.2 Zeus

*Zeus* [7, 26] 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 [15].

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 to 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 ID (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) [6], 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 a malicious user 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:

```
@login_required
@csrf_exempt
@check_instance_auth
@check_admin_lock
def reboot(request, cluster_slug, instance):
    cluster = get_object_or_404(Cluster, slug=cluster_slug)
    [...]
```

### 4.4 Weblate

*Weblate* [9] 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* [5].

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. [4] 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 the "Okeanos". Specifically, the path involved the terms and conditions of the "Okeanos" 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 [16, 18]. 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. [14] 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* [12] 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 [10] 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. [27] 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. [25] use taint tracking 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. WASC [19] and PHP *Aspis* [21] apply further checks when it is established that tainted data have reached a sink. In 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* [23], and *Django Lint* [13] 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 false alarms. *Python Taint* (PYT) [17] extends the functionality of Bandit and performs data-flow analysis thus providing more precise results. Nevertheless, PYT does not take into account elements such as templates and mechanisms such as inheritance. Hence, it cannot address vulnerabilities like the ones described in Section 2. In addition, it does not resolve URLs to views to provide a detailed path. Contrary to the above techniques, Pythia takes into account all the novel features of a modern, Django-based application including templates and decorators. In this way it identifies vulnerabilities that other tools fail to do so.

## 6 CONCLUSIONS

Modern web frameworks intent 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

[4] https://github.com/WeblateOrg/weblate/blob/d84dfdbdfaba0d31588fd3bc9253e6d75d03e644/weblate/trans/views/basic.py#L91-L129

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.

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 (see also Section 2).

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 similar to WASC [19] and PHP Aspis [21] (see Section 5).

## 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

[1] [n. d.]. Misago Project Forums. https://misago-project.org/. [Online; accessed 1-January-2019].

[2] [n. d.]. phpMyAdmin: Bringing MySQL to the web. https://www.phpmyadmin.net/. [Online; accessed 1-January-2019].

[3] [n. d.]. Pylint: Code Analysis for Python. https://www.pylint.org/. [Online; accessed 20-December-2018].

[4] [n. d.]. Python AST-based static analyzer from OpenStack Security Group . https://github.com/openstack/bandit. [Online; accessed 20-December-2018].

[5] [n. d.]. The Debian Administrator's Handbook. https://debian-handbook.info/. [Online; accessed 1-January-2019].

[6] [n. d.]. ViMa, Virtual Machines. https://vima.grnet.gr. [Online; accessed 1-January-2019].

[7] [n. d.]. Zeus E-voting Service. https://zeus.grnet.gr/zeus/. [Online; accessed 1-January-2019].

[8] Ronan Barrett and Sarah Jane Delany. 2004. OpenMVC: A Non-proprietry Component-based Framework for Web Applications. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers &Amp; Posters (WWW Alt. '04)*. ACM, New York, NY, USA, 464–465.

[9] Michal Cihar. [n. d.]. Weblate: Bring translators closer to development. https://weblate.org/en/. [Online; accessed 1-January-2019].

[10] Johannes Dahse and Thorsten Holz. 2014. Static Detection of Second-order Vulnerabilities in Web Applications. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 989–1003.

[11] Luo GuangChun, WangYanhua Lu, and Xianliang Hanhong. 2003. A Novel Web Application Frame Developed by MVC. *SIGSOFT Softw. Eng. Notes* 28, 2 (March 2003), 7–.

[12] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 258–263.

[13] Chris Lamb. [n. d.]. Django-Lint: Static analysis tool for Django projects. https://chris-lamb.co.uk/projects/django-lint. [Online; accessed 20-December-2018].

[14] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 18–18.

[15] Panos Louridas, Georgios Tsoukalas, and Dimitris Mitropoulos. [n. d.]. Requirements and User Interface Design. https://panoramix-project.eu/wp-content/uploads/2016/10/D5.1.pdf. [Online; accessed 20-December-2018].

[16] Stephen McCamant and Michael D. Ernst. 2007. A Simulation-based Proof Technique for Dynamic Information Flow. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security (PLAS '07)*. ACM, New York, NY, USA, 41–46.

[17] Stefan Micheelsen and Bruno Thalmann. 2016. PyT - A Static Analysis Tool for Detecting Security Vulnerabilities in Python Web Applications. https://github.com/python-security/pyt

[18] Dimitris Mitropoulos, Panos Louridas, Michalis Polychronakis, and Angelos D. Keromytis. 2017. Defending against Web application attacks: approaches, challenges and implications. *IEEE Transactions on Dependable and Secure Computing* PP (2017).

[19] Susanta Nanda, Lap-Chung Lam, and Tzi-cker Chiueh. 2007. Dynamic Multi-process Information Flow Tracking for Web Application Security. In *Proceedings of the 2007 International Conference on Middleware Companion*. ACM, Article 19, 20 pages.

[20] Frolin S. Ocariza, Jr., Karthik Pattabiraman, and Ali Mesbah. 2015. Detecting Inconsistencies in JavaScript MVC Applications. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 325–335.

[21] Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. 2011. PHP Aspis: Using Partial Taint Tracking to Protect Against Injection Attacks. In *Proceedings of the 2Nd USENIX Conference on Web Application Development*. 2–2.

[22] PHP Laravel [n. d.]. Laravel - The PHP Framework For Web Artisans. https://laravel.com/. [Online; accessed 20-December-2018].

[23] PyCQA [n. d.]. *Flake8: Your Tool For Style Guide Enforcement*. PyCQA. [Online; accessed 20-December-2018].

[24] Stack Overflow Django CSRF [n. d.]. How to exempt CSRF Protection on direct_to_template. https://stackoverflow.com/questions/11610306/how-to-exempt-csrf-protection-on-direct-to-template. [Online; accessed 20-December-2018].

[25] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *23rd USENIX Security*. 655–670.

[26] Georgios Tsoukalas, Kostas Papadimitriou, Panos Louridas, and Panayiotis Tsanakas. 2013. From Helios to Zeus. In *2013 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '13, Washington, D.C., USA, August 12-13*. https://www.usenix.org/conference/evtwote13/workshop-program/presentation/tsoukalas

[27] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. 2007. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS '07*.