Athens Journal of Sciences- Volume 5, Issue 3 – Pages 261-280

An Approach for the Automated Detection of XSS Vulnerabilities in Web Templates

By Sebastian Stigler[*]
Gulshat Karzhaubekova[†]
Christoph Karg[‡]

Web sites are exposed to various threats such as injections or denial of service attacks. Hence, the protection of the web site and the underlying system components is of major importance in order to deploy a reliable and secure web application. The task of securing a web application is quite complex. It requires the hardening of the system components and installation of security patches on a regular basis. Furthermore, the web application should be checked against vulnerabilities by using penetration testing methods. From a developer's point of view, security aspects should be considered in the early stage of web application's development life cycle already. Of special interest is the web template which is used to generate the user interface of the web page. This component connects the web application to the outside world. Hence, it is the main entrance point to inject malicious things into the web application. In this paper, we present an approach to check web templates against cross site scripting (XSS) attacks in an automatic manner. The basic idea is inspired by the LLVM compiler which uses an internal representation to optimize the machine code independently from the used programming language. Instead of searching for vulnerabilities selectively in the template code itself, the web template is converted into an internal representation. This representation decouples the structure of the web template from the syntax of the template engine. The internal representation is analysed with respect to exploitable parts. The results of the analysis are potentially vulnerable code snippets. These snippets are coded back into the template format and are used within unit tests to check whether they can be exploited with actual attack vectors. The advantage of our approach is its universality; it works with any kind of web template engine.

Keywords: *Cross Site Scripting Attack, Web Security, Web Template Engines, Unit Testing.*

Introduction

Web development is nowadays usually not done by employees of the company for which the website is used but by external contractors. These contractors have to obey the requirements that their costumer set for the project which may include the actual web framework. Therefore a developer can be confronted with different frameworks for different costumers. One component of such a web framework is the Web Template Engine which may differ, too.

[*]Academic Staff, Aalen University of Applied Sciences, Germany.
[†]Academic Staff, Aalen University of Applied Sciences, Germany.
[‡]Professor, Aalen University of Applied Sciences, Germany.

https://doi.org/10.30958/ajs.5-3-4 doi=10.30958/ajs.5-3-4

Unfortunately (from the perspective of a possible attack vector) the templates, generated for these Web Template Engines, look often very much alike but behave not always equal. Due to the fact that software engineers learn in their technical training that they should use techniques such as design pattern, as often as possible. Hence most of them develop some own pattern to solve problems concerning the development of templates for web frameworks, too.

Combined, this two facts, lead to the problem that the developer uses a pattern for Web Template Engine A in a template for Web Template Engine B. This problem is even bigger if the developer is not a full stack web developer who works on the frontend and the backend of the website but just a frontend developer who does not know exactly which Web Template Engine is used in the project.

From the developers perspective it seems everything is fine, but from the security perspective it can lead to different gaps, which, at the development phase, one may not be noticed at all. In our approach, we show exactly where such a gaps can be found and tests it against the used web framework with a selenium[1] driven unit tests in order to give (mostly) the backend developer some hints which variables must be treated extra carefully and which not.

Cross site scripting (XSS) is the most serious vulnerability according to the Open Web Application Security Project (OWASP). "Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites"[2].

The usual scenario for XSS attack is when a user tries to load the malicious code (mostly JavaScript code) into the web page (for example in a comment field). This code can be stored in database and then (unintentional) be executed by rendering the template and load it in the browser.

We focus on exploiting XSS as well as we check web templates against XSS attacks in an automatic manner. We use automatic test case generation to determine which template variables (in the context of the surrounding HTML code) could be vulnerable to such attacks and then test them with selenium in the browser.

In our approach we develop a test tool, which not only analyze templates, detect vulnerabilities and generate a XSS attack for this template code, but also yields a report with a list of potential injection points in the source code. These results can be used by the developer to harden the front- and the backend against such attacks.

A very important aspect while designing our tool was, that it should be very easy to add a new Web Template Engine and the corresponding unit tests for the web framework to the tool.

[1]See https://www.seleniumhq.org/.
[2]See https://www.owasp.org/index.php/Cross-site_Scripting_(XSS).

Literature Review

There are various techniques to detect XSS attacks. Static analysis technology is widely developed for analysis and recognition of code bottlenecks. The static test examines the code through walkthroughs, syntax checking and code reviews, but the code is not running (Conrad et al., 2016). However, this approach to code testing has high false positive and negative rates. "A false positive rate is a reported vulnerability in a program that is not really a security problem. A false negative is a vulnerability in the code which is not detected by the tool" (Diaz and Bermejo, 2013). A research team from university of Spain used nine tools to detect performance of static analysis for detecting vulnerabilities in source code. They used SAMATA test suite 45 with 74 specific cases. Their research showed that by using nine tools with 74 specific cases the summary result has false positive rate ranging from 6.5 to 63 percent. They also studied each tool of all these vulnerabilities separately and reported that false positive rate for 20 CWEs in total (Common Weakness Enumeration) has 9.5 – 43.7 percent range (Diaz and Bermejo, 2013).

Other related work explored the ability of static code analysis to detect vulnerabilities. The process of analysis was based on the benchmark test suite, called Juliet (Goseva-Popstojanova and Perhinschi, 2015). Juliet test suit has a large number of different types of vulnerabilities. It allows to fix probability of false alarm, accuracy, as well as to find probability of code vulnerabilities, etc. Their research was experimentally proved. The results showed that the use of static analysis tools has a high false alarm rates to identify security vulnerabilities. The authors confirmed that qualitative analysis of vulnerability cannot only be limited by statistical analysis (Goseva-Popstojanova & Perhinschi, 2015).

There are other methods of detecting vulnerabilities. One of them is black box testing which does not discover the internal code, but it can use date inputs and outputs (Conrad et al., 2016). This means that the tester can not affect the source code. The application can only be tested from the outside-in. This process will indicate that the tester has similar behavior to the real attacker ones. The black-box testing was analyzed by scholars from the University of Canada on black-box web application scanners effectiveness in detection of stored SQL and XSS injection. The research team used three tools and defined a complex series of steps to detect XSS vulnerabilities for the tester. For example, one of the steps that the tester should do is to load the malicious JavaScript code and submit it in the comment field. In the frame of the research there were registered 32 users. The system detected the stored XSS attack in 29 cases. The black-box testing approach proved its efficiency in detecting XSS attacks. However, it cannot deal with any XSS attacks. If XSS attack is loaded in the text filed, then the system perceives any other XSS attack as a stored XSS attack. That's why, the scanner must "improve use right attack in right situation" (Parvez et al., 2015). The researchers also recommend adding interactive multi-step options to the black-box scanner. It will help to understand interaction between pages as well as identify the place of a potential vector attack. The experiments showed that black box scanners fail to detect XSS vulnerabilities without multi-step options (Parvez et al., 2015).

There is another related approach called unit testing to detect XSS vulnerability when source code is still under development cycle (Mohammadi et al., 2017). Unit testing searches potential vulnerabilities in source code. The research team developed a JSP code analyzer that is aimed at taking all untrusted statements in the source codes. The JSP code analyzer converts statements and variables from source code to equivalent Java statements and host variables. A typical JSP file contains not only all elements of HTML and JavaScript but also Java statements and host variables. The unit testing focuses on finding input points (so called "injection points") which are a variable or a part of the code, where one can load the attack. If any attack is found in the line of the source code, then the researchers can identify the same line number in JSP file. The team uses context free grammar (CFG) rules (Mohammadi et al., 2017) to determine the type of attack strings. These attack strings are loaded at the input points via JavaScript-interpreter. The test is complete if the entered attack is launched in the original code. This approach presents low false positive rate and detects vulnerabilities which cannot be found in black box fuzzing system. The use of CFG helps to generate a wide range of attacks strings which aren't used in other systems (Mohammadi et al., 2017).

A group of researchers from Institute for Software Technology, Austria studied and compared the accuracy of automatic testing with a state-of-the-art manual testing tool (Bozic et al., 2015). They use combinatorial grammar for XSS attack vectors and presented it in BNF form. To improve the quality and reduce input space, they add constraints to XSS attack grammar. This allowed them to get the best result when comparing a fuzz method with a combinatorial one. They confirm that combinatorial testing gives higher exploitation rates of web applications on testing XSS attack (Bozic et al., 2015).

The authors demonstrate their research on two cases. The first case is about comparing attack pattern based on combinatorial testing with manual testing tool. The second case is the comparison of attack pattern results based on combinatorial testing with fuzz testing. In order to compare these two cases, they define "exploration rate" as the number of XSS attacks used divided by the total number of tested attacks. First the OWASP Broken Application Project and the Exploit Database Project were used to test two cases and then the researchers added to them such applications as WebGoat, Gruyere and Bitweaver. The result of testing for the first case showed that combinatorial testing can get better exploration rate "by applying constraints upon input generation" and "with greater combinatorial interaction strengths". They state that testing with constraints can have seven times more exploration rate than testing without constraint. The result of testing for the second case indicate that combinatorial testing can get better result in some tests runs only if it uses quality attack vectors and size of attack strings (Bozic et al., 2015).

In conclusion, there are different approaches for determining vulnerability and in particular XSS. The most effective approach is to detect the vulnerabilities in development of source code cycles. Context free grammar and constraints for determining XSS attacks have low false positive rate and better exploration rate. Therefore, using approaches in source coding mentioned above give more

effective result. In our approach we use a parser to analyze the raw document. The parser reads the information from the document and translates it into an abstract syntax tree (AST). Abstract syntax tree is an abstract data structure, which represent the structure of source code (Hermerschmidt et al., 2015). We use ASTs in our compiler not only to define logic and structure, but also to find vulnerabilities in the source code. We generate our XSS attack string with constraints and look for the part of code where we can load our XSS attack.

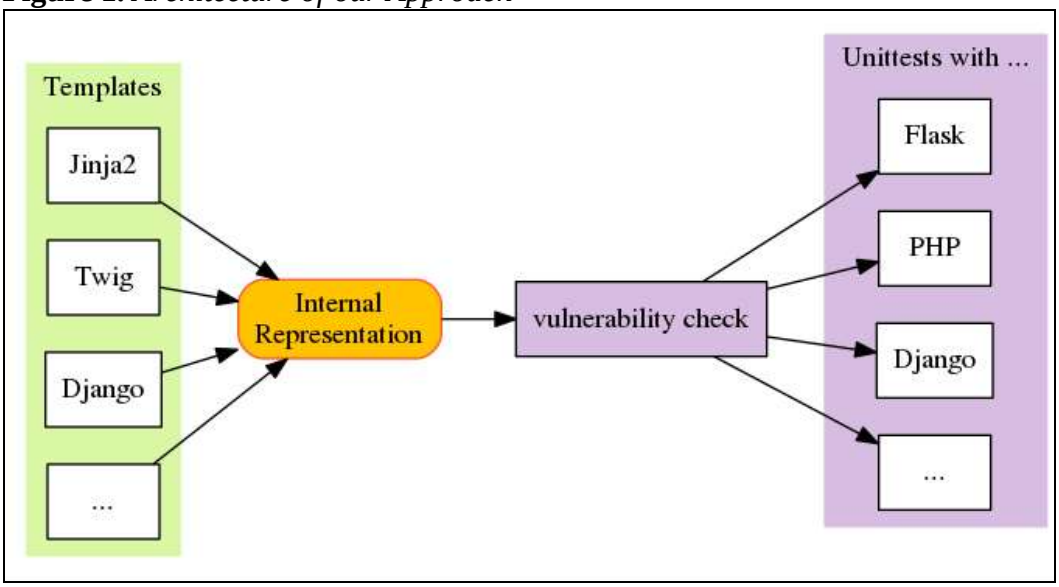
Methodology

Our architecture is based on the idea of the LLVM compiler (Lattner, 2012) to make the frontend and the backend of the compiler exchangeable and use a common optimizer code which works on an internal representation (IR). The frontend has to convert the programming language to this IR and the backend has to produce machine code from it.

We do something quite similar as can be seen in (Figure 1). We parse the template to an internal representation (IR). On this IR we perform the checks for potential vulnerable code fragments and create from these fragments an unit test that we can run to verify if it is indeed a vulnerability for this web framework or not.

Before we go any deeper in the explanation of our approach we have to define the scope of term "Web Template Engine".

Figure 1. *Architecture of our Approach*



What do we consider a Web Template Engine and what not

The main purpose of a Web Template Engine is to separate the business logic from the view. Therefore, the Web Template Engines we consider obeys the following rules:

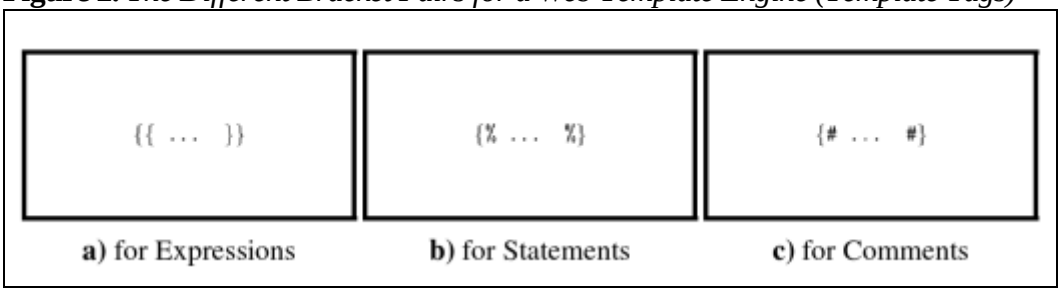
1. There is only a restricted set of control structures:
   - Loop i.e. *for*, *foreach* or *while*.
   - Condition i.e. *if*, *elif* and *else*.
   - Filter[3] i.e. *{{ variable|filter }}*
   - Setting of variables.
   - Printing of a variable.
2. There may be mechanisms to include other templates, to use inheritance of templates[4]or to use macros, written only in the restricted instructions from above.
3. There is no way to write pure code in the language that is used for the backend (i.e. PHP, Python or Java) within the template.

Hence things like "old school" PHP intermixed with HTML or JSP which can contain (nearly) arbitrary Java code are explicitly excluded from our consideration.

Structure of a Template for a Web Template Engine

A template is a text document where HTML code is intermixed with control structures of the Web Template Engine. These control structures are separated from the HTML code via a combination of brackets and some other characters which may differ for the miscellaneous Web Template Engines.

Figure 2. *The Different Bracket Pairs for a Web Template Engine (Template Tags)*



In Figure 2 you see the three different bracket pairs used by Jinja2[5]. The Expression bracket pair is used to print the value of a variable ("print something"). The Statement bracket pair is used for commands like loops, conditions or setting of variables. You can see them as a "do something" entry. Finally the Comment bracket pair is used for comments.

Different Web Template Engines may differ in which bracket combinations are used for these three groups or if some of this groups uses the same bracket combinations. I.e. the Web Template Engines Jinja2, Twig[6] and Django[7] are

[3]Filters are a restricted set of predefined functions that can be applied to variables or literals via a postfix notation.

[4]You can for example have a base template where some parts are marked in some special way, so that you can, in a derived template, override this marked part or extend it.

[5]A Web Template Engine for Python. See *http://jinja.pocoo.org/docs/2.10/templates/*.

[6]A Web Template Engine for PHP that is designed with Jinja2 in mind. See *https://twig.symfony.com/doc/2.x/*.

using the same bracket types while for example Smarty[8] uses only a single pair of curly braces for the Statement and the Expression type with no Comment type at all. This is an important point that has to be considered in the design of the parser for the IR later on. An example of a (Jinja2) template can be seen in Figure 3.

Figure 3. *A Simple Template with a for Loop and an Output Variable*

```
<!DOCTYPE html>
<html>
    <body>
        <h1>Hello world</h1>
      {% for name in usernames %}
        <p>Hello {{ name }}</p>
      {% endfor %}
    </body>
</html>
```

The Internal Representation

The internal representation is represented by a two-staged intertwined abstract syntax tree (AST).

The First Stage Ast

For the first stage we extended a HTML-parser[9] with the ability to detect the three bracket pairs from Figure 2. As HTML itself has a tree structure and we do not need to understand the structure of the document in a deep way, we can directly, create an AST from the output of the parser. The following classes represent the nodes in the first stage AST:

- The different characteristics of a tag (without template tags within): *StartTag*, *EndTag* and *StartEndTag*.
- The attribute of a tag (without template tags within): *Attribute*.
- The different kind of entries between tags (without template tags within): *CharacterReference*, *Comment*, *Data*, *Declaration*, *EntityReference* and *ProcessingInstruction*.
- The three types of template tags: *TemplateExpression*, *TemplateStatement* and *TemplateComment.*
- An attribute with an Expression template[10] within: *TemplateAttribute.*
- A Tag with an Expression template within *TemplateStartTag* and *TemplateStartEndTag.*

[7]The Web Template Engine for a web framework of the same name for Python. See *https://docs.djangoproject.com/en/2.0/topics/templates/#the-django-template-language*.
[8]Another Web Template Engine for PHP. See *https://www.smarty.net/docs/en/smarty.for.designers.tpl*.
[9]See *https://github.com/python/cpython/blob/3.5/Lib/html/parser.py*.
[10]Another template tag type makes no sense within a HTML tag or its attribute.

The *TemplateStartTag* and the *TemplateStartEndTag* classes are use to find nested template tags within HTML tags. If we parse the HTML document in Figure 3 we will get the following first stage AST (Figure 4).

Figure 4. *The First Stage AST*[11] *of Figure 3*

```
Declaration(raw="<!DOCTYPE html>")
StartTag(tag="html", attributes=[])
  StartTag(tag="body", attributes=[])
    StartTag(tag="h1", attributes=[])
      Data(raw="Hello world")
    EndTag(tag="h1")
    TemplateStatement(entry="for name in usernames")
    StartTag(tag="p", attributes=[])
      Data(raw="Hello ")
      TemplateExpression(entry="name")
    EndTag(tag="p")
    TemplateStatement(entry="endfor")
  EndTag(tag="body")
EndTag(tag="html")
```

In order to be as flexible as possible we parameterized the parser in such a way, that we can load the definition of character strings for the various bracket types via a YAML[12] -file. Therefore we can parse arbitrary template files into the first stage AST (Figure 5).

Figure 5. *Parameter File for First Stage AST for a Jinja2-like Web Template Engines (bracket-jinja2.yaml)*

```
---
Expression:
    open: '{{'
    close: '}}'
Statement:
    open: '{%'
    close: '%}'
Comment:
    open: '{#'
    close: '#}'
```

As mentioned above, there are Web Template Engines, where two or more types of template tags have the same bracket pair. Therefore we cannot be sure if a piece of text is parsed to i.e. a *TemplateExpression* or a *TemplateStatement*. In such a case we parse it to that type, the bracket pair appears first[13] and flag this object via an attribute as *ambiguous* and shift the final decision to the parsing of the second stage AST where we can determine the correct type by looking at the entry in the Template-object.

[11]The indentation denotes the hierarchical level in the tree.
[12]YAML is a human friendly data serialization standard. See *http://yaml.org*.
[13]The template tag types are checked in the order Expression, Statement and Comment.

The Second Stage Ast with Back-References to the First Stage Ast

For the second stage we walk the first stage AST in depth-fist-order and build the second stage AST from the entries of the *Template** object (where the *TemplateComment* objects will be ignored). The following classes forming the nodes of the second stage AST (bold and italic text denotes base classes):

- Loops are built from the following classes: ***Loop***, *For*, *Foreach* and *While*.
- For conditions we have the classes ***Condition***, *If* and *ElseIf*. Furthermore there is also the class for ***Else***.
- To set the value of a variable the ***Set*** class is used.
- To transform variables there are ***Function*** and *Filter* classes.
- *Variable* and *Literal* are Subclasses of ***Expression***.
- To mark the end of block in the template there are the following classes: ***End***, *EndFor*, *EndForeach*, *EndWhile*, *EndIf* and *EndSet*.

By parsing the first stage AST in Figure 4 we get the second stage AST:

Figure 6. *The Second Stage AST[14] of (Figure 4)*

```
1 Foreach(item=Variable(name="name", ref=None),
2         collection=Variable(name="usernames", ref=None),
3         ref=7)
4   Variable(name="name", ref=10)
5 EndForeach(ref=12)
```

As you can see in Figure 6 the outer objects have a reference to the first stage AST *Template**objects (denoted by the line number). The two *Variable* objects within the *Foreach* have no reference back to the first stage AST.

As with the first stage AST, we parametrized the parser for the second stage in the form of a YAML-file:

Figure 7. *A Part of the Parameter File for the Second Stage AST for a Jinja2-like Web Template Engines (ast-jinja2.yaml)*

```
1 ---
2 foreach:
3     active: True
4     str: 'for {item} in {collection}'
5     regex: '^\s*for\s+(?P<item>.*)\s+in\s+(?P<collection>.*)\s*$'
6     templatetype: 'TemplateStatement'
7 endforeach:
8     active: True
9     str: 'endfor'
10    regex: '^\s*endfor\s*$'
11    templatetype: 'TemplateStatement'
12 # [...]
```

[14]The indentation denotes the hierarchical level in the tree.

For each class, of the used Web Template Engine, there is an entry in the YAML-file with the following parameters:

- *active*: Indicates if this object is used in this Web Template Engine. I.e. Jinja2 has a *Foreach*-loop but no *while*-loop.
- *str*: This entry is used when we generate source code from the object in the AST.
- *regex*: The regular expression (in Python notation) to parse the entry of the corresponding *Template** node in the first stage AST.
- *templatetype*: Indicates in which *Template** class the object can be found in this Web Template Engine.[15]

Finding Vulnerable Pieces in the Template via the Internal Representation

Vulnerable places in the template are the parts where Expression template tags with variables are located (*Literal*s are of no concern because they cannot be changed at rendering time). Furthermore are these variables of interest, which are not set in the template itself. The value of this variable (henceforth referred to as free variable) is set by the backend at rendering time.

To find these two types of variable is quiet easy: While building the second stage AST you build up a data flow graph and put all variables that are in a*TemplateExpression* object in a list (the "output list").

A data flow graph can be constructed the following way: its nodes are all literal and variables. *Function*s or *Filter* that manipulates the variable are of no concern for the data flow graph because we use it only to determine which variable influence which other variable. There is an edge from node A to node B, when the value of A is directly written in the variable B[16].

There a few observations on the data flow graph:

- The resulting graph is a forest[17].
- The root of each tree is either a literal or a free variable.
- All entries in the output list are leaf nodes.
- Leaf nodes that are not in the output list are conditions in the ***Condition*** objects.

The vulnerable variables are the ones in the output list that have a free variable as root in the data flow graph (the ones with literals as root are of no concern).

Now all we have to do is to extract for each vulnerable variable these parts of the second stage AST that are in the corresponding tree in the data flow graph and

[15]This is used because of the potential ambiguity of the bracket types as mentioned in the description of the first stage AST. The second stage AST parser knows so, where to find the entry for the corresponding node class and can, if necessary change the node type in the first stage AST accordingly: All second stage nodes that are not derived from ***Expression*** are in first stage nodes of type *TemlateStatement*.

[16]The destination of an edge in a data flow graph is always a variable.

[17]A graph is called a forest, when all its connected components are trees (Diestel, 2017).

extract the directly surrounding HTML tags for the output variable and generate the corresponding template code (see *str* in Figure 7). How much of the HTML code is needed will be discussed in the "Findings" section.

Construction of the Unit Test

The unit test is split in two components; a general component and a component for the used web framework (individual component).

The General Component

The general component has to do following tasks:

1. Generate a template for the individual component.
2. Send the following informations with the template to the individual component:
   - The name of the free variable.
   - The name of the template file[18].
3. Use selenium to send a payload with the XSS string to the rendered template and click on the tag to trigger the attack.
4. Check if the attack was successful.

Figure 8. *Skeleton for the Template*

```
<!DOCTYPE html>
<html>
    <head>
        <title>Unittest</title>
        <script type="text/javascript">
            function attack() { document.title="ATTACK"; };
        </script>
    <body>
        <!-- <a onclick='attack()'>Attack</a> -->
    </body>
</html>
```

To generate the template as mentioned above we split the skeleton code in Figure 8 at the comment and put the vulnerable code snippet in between.

The Individual Component

This component must be implemented for each web framework that you want to use. It has to take the template, the name of the free variable and the name of template file as input and must use these informations to start a webserver with this template. It must then tell the general template, the url to this webpage. The

[18]See the example in the "Discussion" section.

backend must be able to receive the payload from the general component and use it to fill the free variable with it during rendering.

Generating a Report of the Vulnerable Variables

Finally we create a report in which the variables (and the HTML context from the template), that are vulnerable, are listed. We also add the strings, that trigger the vulnerability to the report in order to give the developers a hint as to how the code must be protected.

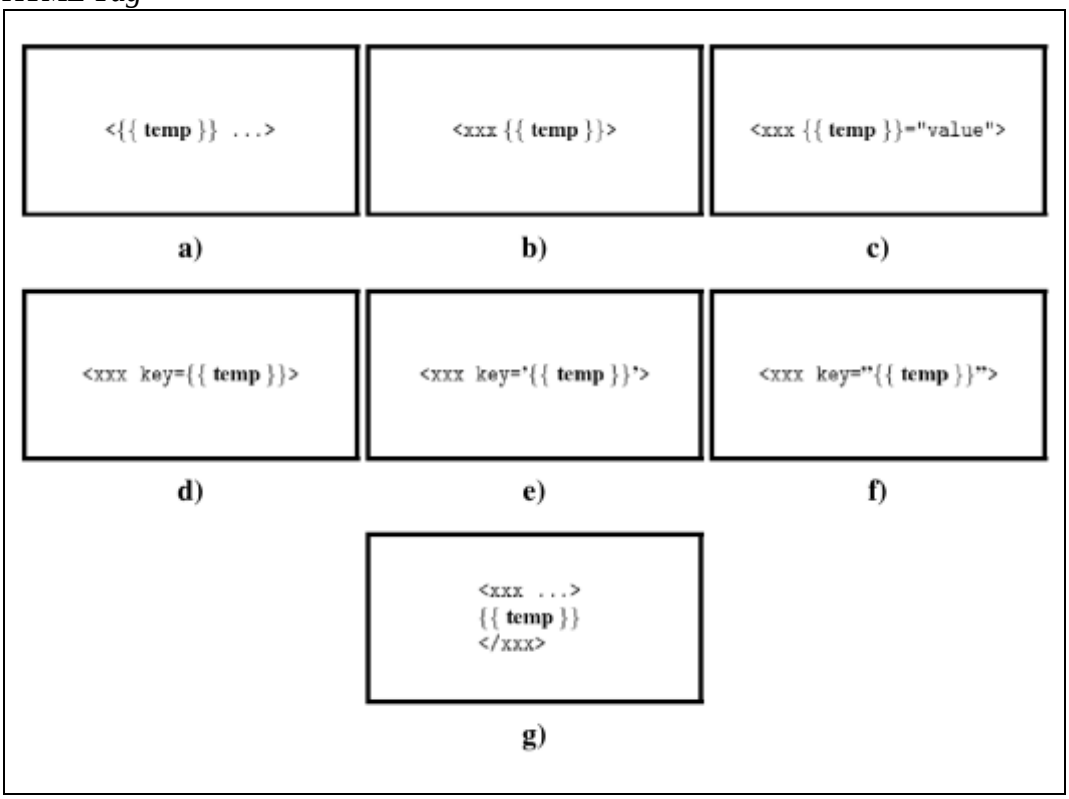
Findings

To prove that a free variable is vulnerable to an XSS attack, we need to know what type of HTML tag must be created with the corresponding output variable. The comment in Figure 8 shows such a tag.

Position of a Template Tag Relative to an Html Tag

To create this tag we need to know where an Expression template tag can be placed within a HTML tag. Figure 9 shows all possible position. In this figure the string *xxx* stands for an arbitrary HTML tag type.

Figure 9. *All Possible Places where a Template Tag can be put regarding to an HTML Tag*



Case A): The Template Tag Defines the Html Type itself

In that case we can simply set *temp* to the following value where the centered dot denotes a space character:

a · onclick = "attack();" ·

Case B)*: The Template Tag Defines the Whole Attribute of the Html Tag

Here we can set *temp* to the value:

· onclick = "attack();" ·

Case C)*: The Template Tag Defines the Key of an Attribute of the Html Tag

We can set the value of *temp* to:

· onclick = "attack();" · title

Case D)*: The Template Tag Defines the Value of an Attribute of the Html Tag (without Quotes)

In this case we have to look what the key of this attribute is. If the key is *onclick* we can set *temp* to:

"attack();" ·

Otherwise we can use the following string:

" " · onclick = "attack();" ·

Case E)*: The Template Tag Defines the Value of an Attribute of the Html Tag (with Single Quotes)

In this case we have to look what the key of this attribute is. If the key is *onclick* we can set *temp* to:

attack();

Otherwise we can use the following string:

' · onclick = 'attack();

Case F)*: The Template Tag Defines the Value of an Attribute of the Html Tag (with Double Quotes)

In this case we have to look what the key of this attribute is. If the key is *onclick* we can set *temp* to:

```
attack();
```

Otherwise we can use the following string:

```
" · onclick = "attack();
```

Case G) The Template Tag is between an Opening and Closing Html Tag

We have to distinguish two cases. The type the enclosing HTML tag does not allow an anchor tag within like the *script* tag. In this case we have to close the opening tag fist, place an anchor tag with the attack code within and then copy the opening tag like this:

</xxx><a·onclick=”attack();”>click<xxx …>

If the HTML tags allows anchor tags within, the value of *temp* can be set to:

<a·onclick=”attack();”>click

Special Cases

On the cases marked with an * we have to look at the edge case where the *onclick* attribute is not valid on the *xxx* tag[19]. We show the string for *temp* exemplary for Case B):

><a·onclick=”attack();”>click<xxx …>

Discussion

Although most modern Web Template Engines have very sophisticated mechanisms to automatically HTML-escape the value of the variables, there are cases where they not suffice. Be it the case that the frontend developer has marked a variable in the template as “save” (so that no HTML escape is necessary) without telling the backend developer or while switching to another Web

[19]For HTML4.01 some tags do not allow the *onclick* attribute. See *https://www.w3.org/TR/2018/SPSD-html401-20180327/index/attributes.html*.
HTML 5 allows the *onclick* attribute on all tags. See *https://www.w3.org/TR/2017/ REC-html52-20171214/dom.html#ref-for-dom-globaleventhandlers-onclick*.

Template Engine the developer overlooked some marginal differences that where in hindsight not as marginal as thought.

An Example where a Marginal Different was not so marginal

Let's assume the following scenario. We have an experienced PHP developer that uses Twig as Web Template Engine. A typical Twig project stores its templates in the subdirectory */template* and the templates get the extension *.twig*. The PHP code is stored in */src*.

The code in Figure 10 shows a simple Twig template with the free variable *name* (that is also the output variable). Figure 11 depicts a simple PHP program which uses the template. In line 6 we simulate an XSS attack and in Figure 12 you see what will happen. Twig escapes the injected HTML code in such a way that the malicious code won't be executed. If you look at Figure 13, you will see the renderd HTML code for Figure 12.

Figure 10. *A Simple Twig Template (/template/index.twig)*

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <meta http-equiv="Content-type" content="text/html; charset=utf-8">
5     <title>Twig test</title>
6 </head>
7 <body>
8     <h1>Hello world</h1>
9     <p>And hello {{ name }}</p>
10 </body>
11 </html>
```

Figure 11. *PHP Code that Uses the Twig Template from Figure 10 with a Malicious Variable Value (index.php)*

```
1 <?php
2 require __DIR__.'/../vendor/autoload.php';
3 $loader = new Twig_Loader_Filesystem(__DIR__.'/../templates');
4 $twig = new Twig_Environment($loader, array());
5
6 $name = '<span onmouseover="alert(\'You are pwned\');">Hans</span>';
7 echo $twig->render('index.twig', array(
8     'name' => $name,
9 ));
10 ?>
```

Figure 12. *The Rendered Twig Template from Figure 10 and Figure 11*

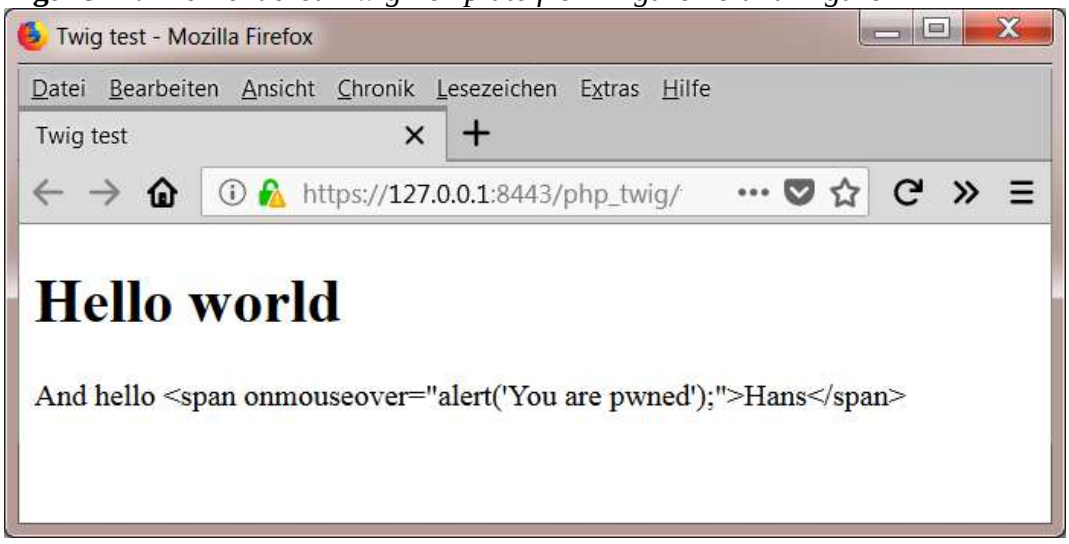


Figure 13. *The HTML Source Code from Figure 12 with HTML Escaped Template Input*

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4     <meta http-equiv="Content-type" content="text/html; charset=utf-8">
5     <title>Twig test</title>
6 </head>
7 <body>
8     <h1>Hello world</h1>
9     <p>And hello &lt;span onmouseover=&quot;alert(&#039;You are
          pwned&#039;);&quot;&gt;Hans&lt;/span&gt;</p>
10 </body>
11 </html>
```

Now the same developer has to learn a new web framework for a new project. The framework he has to learn is Flask with its Web Template Engine Jinja2. Fortunately Twig is based on Jinja2 so he can use the templates he wrote for Twig anew. The structure of a Flask project is to put the templates in the directory */template* and put the Python file in the root directory of the project.

Our developer uses for the following example exactly the same template as in the PHP/Twig project from above (Figure 10). In Figure 14 you can see the Python code for the Website. In line 4 we simulate a XSS attack just like we did in the PHP version of this project. As you can see in Figure 15 you can see, that there are no HTML tags visible in the browser. Figure 16 and Figure 17 show why: Flask has rendered the template (Figure 10) without escaping HTML content in *name*.

Figure 14. *Flask Code that Uses the Same Twig Template as in Figure 10 with a Malicious Variable Value (/app.py)*

```
1 from flask import Flask, render_template
2
3 app = Flask(__name__)
4 name = '<span onmouseover="alert(\'You are pwned\');">Hans</span>'
5
6
7 @app.route('/')
8 def index_twig():
9     return render_template('index.twig', name=name)
```

Figure 15. *The Rendered Twig Template from Figure 10 and Figure 14 (beware: there is no HTML code visible!)*

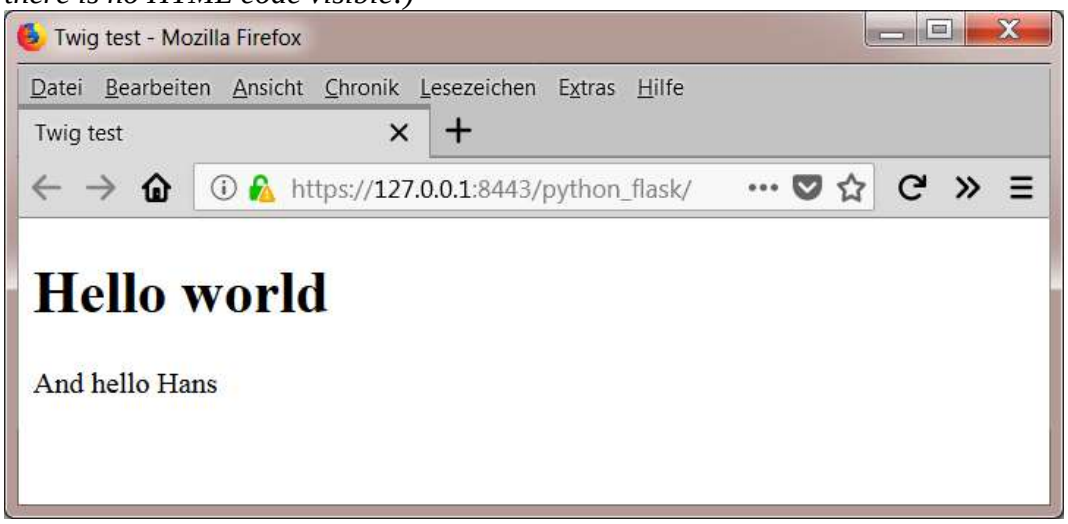


Figure 16. *When you hover with the Mouse over "Hans" in the Browser on Figure 15 Run the Malicious JavaScript Code*

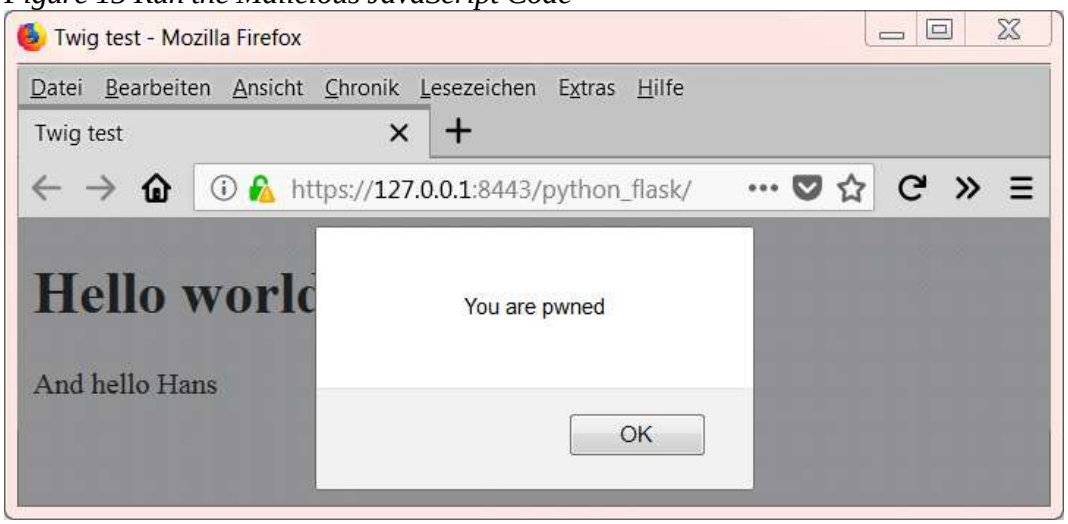


Figure 17. *The HTML Source Code from Figure 15 with Unescaped Template Input*

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Twig test</title>
</head>
<body>
    <h1>Hello world</h1>
    <p>And hello <span onmouseover="alert('You are pwned');">Hans</span></p>
</body>
</html>
```

The question is: Why does Twig with PHP escape the HTML input, while Flask with Jinja2 does not escape it? The answer lies in the fine print of the documentation of Flask and Twig. While for Twig it is totally irrelevant what extension a template has, Flask activates its autoescape functionality only when the template has the extension *.htm*, *.html*, *.xhtml* or *.xml*. So if you rename the template */template/index.twig* to */template/index.html* and change the name of the template file in */app.py* accordingly, then you will get the following rendered HTML code from Flask (Figure 18).

Figure 18. *The HTML Source Code from Flask App with the Correctly Named Template with HTML Escaped Template Input*

```
<!DOCTYPE html>
<html>
<head>
    <meta http-equiv="Content-type" content="text/html; charset=utf-8">
    <title>Twig test</title>
</head>
<body>
    <h1>Hello world</h1>
    <p>And hello &lt;span onmouseover=&#34;alert(&#39;You are
        pwned&#39;);&#34;&gt;Hans&lt;/span&gt;</p>
</body>
</html>
```

Conclusions

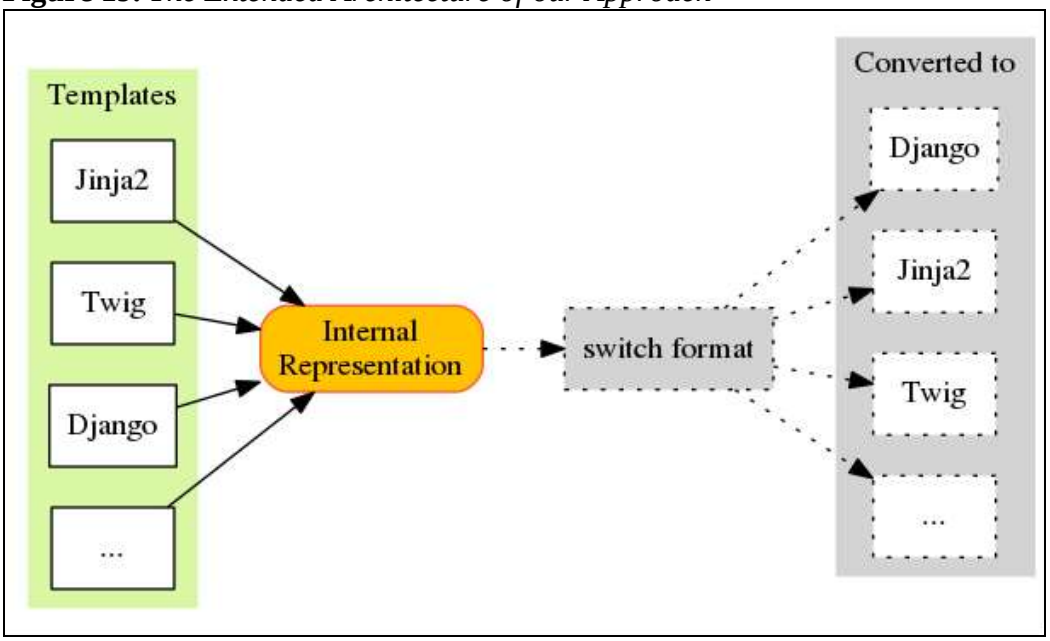
In this article we have shown you our new approach in finding security flaws in Web projects that are located in the templates for the Web Template Engines. What is new to this approach is that we do these tests without the provided backend and are able to test the same template against multiple different Web Template Engines and web frameworks.

To be able to do so, we use a common internal representation (IR) for the templates and parse each template to this IR first. Then we do the vulnerability check on this IR and use parts of this IR to generate a unit test form the template to test it against a web framework with Web Template Engine.

As we designed our tool to be modular and very easily extendable, it is not difficult to teach our tool a new Web Template Engine or a new web framework.

A side effect of our architecture is, that we can, with nearly no effort, use our tool to convert templates in the syntax for other Web Template Engines (see (Figure 19). All we have to do is introduce a translation function that converts unknown structures from the source template to the target template. For example the source template could contain a *while*loop, while the target Web Template Engine only supports *for* loops. Due to the restrictions we made on the Web Template Engine in the "Methodology" section there are no complicated code transformations necessary.

Figure 19. *The Extended Architecture of our Approach*



References

Bozic, J., Garn, B., Kapsalis, I., Simos, D. E., Winkler, S., & Wotawa, F. (2015). Attack pattern-based combinatorial testing with constraints for web security testing. *2015 IEEE International Conference on Software Quality, Reliability and Security*, *6*(4), 207-212. https://doi.org/10.1109/QRS.2015.38.

Conrad, E., Misenar, S., & Feldman, J. (2016). Chapter 7 - Domain 6: Security Assessment and Testing (Designing, Performing, and Analyzing Security Testing). In E. Conrad, S. Misenar, & J. Feldman (Eds.), *{CISSP} study guide (third edition)* (Third Edition, pp. 329-345). Boston: Syngress. https://doi.org/B978-0-12-802437-9.00007-2.

Diaz, G., & Bermejo, J. R. (2013). Static analysis of source code security: Assessment of tools against samate tests. *Information and Software Technology 55 (2013) 1462-1476*, *55*(2), 1462-1476. https://doi.org/10.1016/j.infsof.2013.02.005.

Diestel, R. (2017). *Graph theory*. Springer Berlin Heidelberg. Retrieved from https://books.google.de/books?id=FY5BvgAACAAJ.

Goseva-Popstojanova, K., & Perhinschi, A. (2015). On the capability of static code analysis to detect vulnerabites. *Information and Software Technology 68 (2015) 18-33*, *68*(1), 18-33. https://doi.org/10.1016/j.infsof.2015.08.002.

Hermerschmidt, L., Kugelmann, S., & Rumpe, B. (2015). Towards more security in data exchange: Defining unparsers with context-sensitive encoders for context-free grammars. *2015 IEEE Security and Privacy Workshops*, *8*(6), 134-141. https://doi.org/10.1109/SPW.2015.29.

Lattner, C. (2012). Chapter 11 - LLVM. In A. Brown & G. Wilson (Eds.), *The architecture of open source applications* (pp. 151–166). lulu.com. Retrieved from http://aosabook.org/en/llvm.html.

Mohammadi, M., Chu, B., Chu, H. R. L. B., &Lipford, H. R. (2017). Detecting cross-site scripting vulnerabilities through automated unit testing. *UNC Charlotte, Charlotte, NC, USA. 2017. IEEE.*, *10*(5), 364-372. https://doi.org/10.1109/QRS.2017.46.

Parvez, M., Zavarsky, P., & Khoury, N. (2015). Analysis of effectiveness of black-box web application scanners in detection of stored sql injection and stored xss vulnerabilities. *The 10-Th International Conference for Internet Technology and Secured Transactions (ICITST-2015).*, *6*(3), 186-191. https://bit.ly/2okdF7A.