Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks

Matthew Van Gundy
University of California, Davis
mdvangundy@ucdavis.edu

Hao Chen
University of California, Davis
hchen@cs.ucdavis.edu

Abstract

Cross-site scripting (XSS) vulnerabilities are among the most common and serious web application vulnerabilities. XSS vulnerabilities are difficult to prevent because it is difficult for web applications to anticipate client-side semantics. We present Noncespaces, a technique that enables web clients to distinguish between trusted and untrusted content to prevent exploitation of XSS vulnerabilities. Using Noncespaces, a web application randomizes the XML namespace tags in each document before delivering it to the client. As long as the attacker is unable to predict the randomized prefixes, the client can distinguish between trusted content created by the web application and untrusted content provided by the attacker. Noncespaces uses client-side policy enforcement to avoid semantic ambiguities between the client and server. To implement Noncespaces with minimal changes to web applications, we leverage a popular web application architecture to automatically apply Noncespaces to static content processed through a popular PHP template engine. We show that with simple policies Noncespaces thwarts popular XSS attack vectors. As an additional benefit, the client-side policy not only allows a web application to restrict security-relevant capabilities to untrusted content but also narrows the application's remaining attack vectors, which deserve more scrutiny by security auditors.

1 Introduction

Cross-site scripting (XSS) vulnerabilities constitute a serious threat to the security of modern web applications. In 2005 and 2006, the most commonly reported vulnerabilities were cross-site scripting vulnerabilities [13]. XSS vulnerabilities allow an attacker to inject malicious content into web pages from trusted web servers. Since the malicious content runs with the same privilege as the trusted content from the web servers, the malicious content can steal the victim users' private data or take unauthorized actions on the users' behalf. To prevent XSS vulnerabilities, all the untrusted content from users in web pages must be sanitized. However, proper sanitization is very challenging. One could let the server sanitize the untrusted content before delivering it to the browser. However, when a browser interprets certain content differently from how the server intends, attackers can take advantage of this discrepancy, exemplified in the Samy worm [4], one of the fastest spreading browser worms to date. Alternatively, one could let the client sanitize untrusted content. However, without the server's help, the client cannot distinguish between trusted and untrusted content in the web pages.

After the server identifies untrusted content, it needs to tell the client the locations of the untrusted content in the document tree. However, if the untrusted content (without executing) could distort the document tree, it could evade sanitization. To achieve this, the untrusted content could contain node delimiters that split the original node where untrusted content resides into multiple nodes. This is known as the *Node-splitting attack* [9]. To defend against this attack, the server must remove all node delimiters from untrusted content, but this would restrict the richness of user provided contents.

We present *Noncespaces*, a mechanism that allows the server to identify untrusted content and to reliably convey such information to the client, and that allows the client to enforce a security policy on the untrusted content. Noncespaces is inspired by Instruction Set Randomization [10], which randomizes the instruction set to identify and defeat injected malicious binary code. Analogously, Noncespaces randomizes XML namespace prefixes to identify and defeat injected malicious web content. These randomized prefixes serve two purposes. First, they identify untrusted content so that the client can enforce a security policy on them. Second, they prevent the untrusted content from distorting the document tree. Since the randomized tags are not guessable by the attacker, he cannot embed proper delimiters in the untrusted content to split the containing node without causing XML parsing errors.

We make the following contributions:

- We draw the analogy between injected code in executable programs and injected content in web pages to apply the idea from Instruction Set Randomization to the defense against XSS attacks.
- We observe that current web application design practices lead to simple, effective policies for defending against popular XSS attack vectors.
- We modify a popular template engine to facilitate automatic deployment of our technique.
- We define a flexible yet simple policy language for client-side policy enforcement.

2 Cross-Site Scripting Vulnerabilities

Noncespaces defends against cross-site scripting (XSS) vulnerabilities. An XSS vulnerability allows an attacker to inject malicious content into a web page returned by a legitimate web server to an unsuspecting client. Typically, when the client receives the document, it cannot tell the difference between the legitimate content provided by the web application and the malicious payload injected by the attacker. The malicious content can disclose private data or authentication credentials allowing the attacker to impersonate the client to the web application.

Figure 1 shows a web page template used by a fictitious web application to render dynamic web pages. The template is written in a language similar to Smarty where content between "{" and "}" characters denotes instructions to the template engine [6]. "{*identifier*}" instructs the template engine to replace the string by the value of the variable given by *identifier*. "{`foreach` $identifier_1$ `in` $identifier_2$} *content* {`/foreach`}" instructs the template engine to evaluate *content* repeatedly, once for each member in the array variable named by $identifier_2$, binding the variable named by $identifier_1$ to the current element of the array for each iteration.

If the web application does not properly sanitize user input, pages rendered from this template may be vulnerable to XSS attacks. For instance, if an attacker can submit the string "`<script src='http://badguy.com/attack.js'/>`" as a review, the template variable `review.text` will be assigned this string during one iteration of the `foreach` loop. When a client visits the page, the client's web browser will download and execute `http://badguy.com/attack.js` with the permissions of the web application.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head><title>nile.com : ++Shopping</title></head>
<body>
<h1 id="title">{item_name}</h1>
<h2>Description</h2>
<p id='description'>{item_description}</p>
<h2>Reviews</h2>
<div id='reviews'>
  {foreach review in reviews}
    <p class='review'>{review.text}
    -- <a href='{review.contact}'>{review.author}</a></p>
  {/foreach}
</div>
</body>
</html>
```

Figure 1: Vulnerable web page template used to render dynamic web pages

There are a number of ways by which an XSS vulnerability can appear in a document. We call these attack vectors. The following are common attack vectors:

- *tag body*: Malicious content embeds new tags in the body of another tag. E.g. `review.text = <script>attack()</script>` in the template in Figure 1.
- *node splitting*: Malicious content closes previously opened tags to traverse up the document tree. This attack can effectively defeat a security policy that constrains the capability of code based on its ancestry in the document. E.g. `review.text = </p></div><script>attack()</script><div><p>`
- *attribute value*: The malicious content embeds a malicious attribute value that violates security without escaping from the attribute value. E.g. `review.contact = javascript:attack()`
- *attribute splitting*: An attribute value breaks out of the intended attribute and defines a new attribute. E.g. `review.contact = ' onclick='javascript:attack()`
- *tag splitting*: An attribute value breaks out of the currently open element to define new elements. E.g. `review.contact = '><script>attack()</script>`

In addition to the numerous vectors that an XSS attack may exploit, discrepancies in parsing HTML can cause the client to interpret content in ways not anticipated by the server. The XSS Cheat Sheet [14] catalogs numerous examples of (often non-intuitive) character sequences that may lead to script execution in various clients.

In this paper, we restrict our attention to XSS attacks where the attack delivers malicious content to the victim user via a trusted server. We do not address Cross-Site Request Forgery (CSRF) attacks, where a malicious web server tricks the client into sending a malicious request to a trusted web site. We also do not address Universal Cross-Site Scripting Vulnerabilities [15] where a browser extension can be tricked into violating the browser's own security policy.

3 Noncespaces

The goal of Noncespaces is to allow the client to reliably distinguish between trusted content, which the web application generates, and untrusted content, which an untrusted user provides, on the same web page. To accomplish this goal, the web application partitions content on a web page into different trust classes. A policy specifies the browser capabilities that each trust class can exercise. This way, an attacker's malicious content can do no more harm than what the policy allows for its trust class.

Noncespaces involves both server-side and client-side components. The server annotates every element and attribute of the delivered XHTML document with its trust classification. We represent each trust class by a random XML namespace prefix. As long as the attacker cannot guess the random prefix, his malicious content cannot change its trust classification. The server also delivers a policy specifying which elements, attributes, and values each trust class permits. The user's browser then verifies that the document it parses conforms to the policy.

3.1 Document Annotation with Trust Classes

To check the conformance of a document against a policy, the client must be able to determine the trust class of every element and attribute in the document. Since the server annotates each element and attribute with its classification and does not sanitize the content, the server must ensure that malicious content cannot change its trust classification. The server could indicate the classification with attributes of elements . However, malicious content may contain elements with attributes that designate trusted content. Alternatively, the server could indicate the classification by the ancestry of a node, e.g. restricting the capabilities of all descendents of a specific document node – a sandbox node. However, malicious content may contain tags that split its original enclosing node into multiple nodes so that malicious nodes are no longer descendents of the sandbox node. This is the node-splitting attack discussed in Section 2.

To reliably annotate content with a trust classification without having to sanitize the content, we use randomized XML namespace prefixes. To illustrate this solution, we draw an analogy between buffer overflow attacks and XSS attacks. During a typical buffer overflow attack, the attacker injects malicious binary code in the overflown buffer. Similarly, during an XSS attack, the attacker injects malicious web content. Our solution is inspired by Instruction Set Randomization. Instruction Set Randomization defends against binary code injection attacks by randomly perturbing the instruction set of an application. If an attacker wishes to inject code into the application, she must correctly guess the randomization used. This is very difficult if the number of randomizations possible is sufficiently large. The attacker is effectively prevented from injecting code because she cannot name the instructions with the desired semantics with sufficient probability.

XML namespaces qualify elements and attributes [8] by associating them with namespaces identified by URL references. To denote the namespace of a tag, the user chooses a string as the prefix of the tag and associates the prefix with the namespace URI in the document. The namespace determines the semantics of a tag. For instance, both `<p:a xmlns:p='http://www.w3.org/1999/xhtml'>` and `<q:a xmlns:q='http://www.w3.org/1999/xhtml'>` specify the `<a>` tag in the XHTML namespace (`http://www.w3.org/1999/xhtml`). XML namespaces are typically used for distinguishing tags that have similar names but different semantics. We leverage namespace prefixes to annotate the trust class of each element and attribute in the document. In other words, each namespace prefix string indicates the trust class of the element or attribute.

To prevent an attacker from forging the trust class designating trusted content and to prevent untrusted content from escaping from its enclosing node (e.g., the node-splitting attack), we must prevent the attacker from guessing the appropriate namespace prefix, i.e. trust class of the trusted content. Otherwise, the attacker can embed a closing tag with the correct prefix in his malicious content to escape from the current node. To this end, we randomly choose the namespace prefixes on every document delivery – hence the term Noncespaces. For instance, if we annotate the document from Figure 1 with the randomly chosen prefix `r617` to indicate trusted code and the empty prefix to indicate untrusted code, the resulting document is shown in Figure 2.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<r617:html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" xmlns:r617="http://www.w3.org/1999/xhtml">
<r617:head><r617:title>nile.com : ++Shopping</r617:title></r617:head>
<r617:body>
<r617:h1 r617:id="title">Useless Do-dad</r617:h1>
<r617:h2>Description</r617:h2>
<r617:p r617:id='description'>Need we say more.</r617:p>
<r617:h2>Reviews</r617:h2>
<r617:div r617:id='reviews'>
    <r617:p r617:class='review'>
        </p></div><script>attack()</script><div><p>
    --
    </r617:p>
</r617:div>
</r617:body>
</r617:html>

```

Figure 2: Random prefix applied to trusted content in XHTML document

As illustrated by the embedded node-splitting attack, the attacker cannot inject malicious content and annotate it with the trusted class `r617` because he does not know the random prefix `r617`. He also cannot escape from the node, because he does not know the random prefix `r617` and therefore cannot embed a closing tag with this prefix. When a closing tag tries to close an open tag but the prefixes of the two tags mismatch, the XML parser will fail with an error.[1]

Since the server chooses new random prefixes each time it serves a document, even if the attacker knows the prefixes in one instance of the document, he cannot predict the prefixes in future instances of the document.

Using XML namespace prefixes to represent trust classes has several benefits. First, XHTML 1.0 compliant browsers support XML Namespaces and therefore are able to render the transformed document as intended, even if they cannot check Noncespaces policies. Second, using namespaces allows us to use the standard XPath language in our policies with a minor semantic difference. In plain XPath, expressions match against the expanded name (namespace URI + tag name) of a tag. In our XPath policy expressions, the prefix used in the expression must match the prefix used in the document. Finally, using randomized namespace prefixes defeats node-splitting attacks.

[1]A subtlety occurs when two different prefixes, say `a` and `b`, are associated with the same URI. In this case, is "`<a:foo></b:foo>`" valid? Syntactically it is invalid because the prefixes in the open and close tags are different, but semantically it is valid because both the prefixes are associated with the same URI. We found that almost all the current browsers reject XML documents that are syntactically invalid; therefore, they consider the above example ill-formed. This implies that Noncespaces needs to randomize only namespace prefixes, but not the URIs that the prefixes are associated with.

3.2 Policy Specification

A Noncespaces policy specifies what browser capabilities are allowed for each trust class. We designed the policy language to be similar to a firewall configuration language. A Noncespaces policy consists of a sequence of rules. Each rule describes a set of nodes using an XPath 1.0 expression and specifies a policy decision – either allow or deny – on these nodes. The XPath expression of a node can include its trust class (i.e. its namespace prefix), name, attributes, position the document, or any other criteria expressible as an XPath expression. For instance, to allow all trusted elements, we can specify the rule "`allow //trusted:*`". To allow the `b` tag in untrusted content, we can specify the rule "`allow //untrusted:b`". Figure 3 shows a policy that allows any XHTML tag and attribute in trusted content but allows only a safe subset of the markup elements in untrusted content.

```
# Restrict untrusted content to safe subset of XHTML

# Declare namespace prefixes, which are also the names of trust classes.
namespace trusted
namespace untrusted

# Policy for trusted content
# Allow all elements
allow //trusted:*
# Allow all attributes
allow //trusted:@*

# Policy for untrusted content
# Allow safe elements
allow //untrusted:b
allow //untrusted:i
allow //untrusted:u
allow //untrusted:s
allow //untrusted:pre
allow //untrusted:q
allow //untrusted:blockquote
allow //untrusted:a
allow //untrusted:img
# Allow HTTP protocol in the href attribute
allow //untrusted:a/@untrusted:href[starts-with(normalize-space(.), "http:")]
# Allow HTTP protocol in the img attribute
allow //untrusted:img/@untrusted:src[starts-with(normalize-space(.), "http:")]

# Fail-safe defaults
# Deny all elements
deny //*
# Deny all attributes
deny //@*
```

Figure 3: Noncespaces policy restricting untrusted content to BBCode [2]

When checking a document's conformance to a policy, the client considers each rule in order and matches the XPath expression in that rule against the nodes in the document's Document Object Model. When an allow rule matches a node, the client permits the node and will not consider the node in subsequent rules. When a deny rule matches a node, the client determines that the document violates the policy and will not render the document. To have a fail-safe default, if a node does not match any rule, we consider the node in violation the policy. If one wishes to specify a blacklist policy, he can specify `allow //*|//@*`, which allows all nodes, as the last rule in the policy. Algorithm 1 in the Appendix shows the algorithm for checking policy.

We prefer this policy mechanism to more complex ones like dynamic information flow tracking or event-based policies for its simplicity and ease of implementation across browsers. It also fits naturally with a fairly common scenario in web applications where content in the application's source can be considered trustworthy while content specified by users should be allowed a minimal set of capabilities.

3.3 Server Annotation

Using Noncespaces, the server annotates nodes in an XHTML document with trust classes. The server could use a variety of techniques to determine the trust classes, ranging from whitelisting known-good code to annotating output based on program analysis or information flow tracking. Using randomized namespace prefixes as trust class annotations, the server ensures that untrusted content can never change their trust classification.

Besides annotating nodes with trust classes, the server also needs to convey a policy to the client. Noncespaces adds three HTTP protocol headers to each HTTP response: `X-Noncespaces-Version`, `X-Noncespaces-Policy`, and `X-Noncespaces-Context`. Their semantics are as follows:

- `X-Noncespaces-Version` communicates the version of the Noncespaces policy and semantics that should be used, in case future changes are required .
- `X-Noncespaces-Policy` denotes the URL of the policy for the current document. If the client does not have the policy in cache, a compliant client must first retrieve the policy before rendering the document.
- `X-Noncespaces-Context` maps the namespace prefixes in the policy to the namespace prefixes in the XHTML document contained in the response. To prevent an attacker from guessing the namespace prefixes in an XHTML document, the server must use different randomized prefixes each time it serves the document. On the other hand, it would be convenient for the server to provide the same policy file to all the requests for the XHTML document (this would also allow the client to cache the policy file). `X-Noncespaces-Context` maps the static namespace prefixes in the policy file to the randomized namespace prefixes in the XHTML document contained in the response.

We maintain backwards compatibility with XHTML 1.0 compliant browsers by using `X-` headers. If a web browser is not Noncespaces capable, it will ignore the headers and process the document as XHTML 1.0. In this case, even though the web browser will render untrusted content that the policy would deny, malicious content still cannot escape its containing node (e.g. node-splitting attacks still cannot succeed).

The server should serve Noncespaces documents with the `application/xhtml+xml` content type to activate the stricter XML parser, which rejects a document if the namespace prefixes of any pair of open and closing tags mismatch.

3.4 Client Enforcement

When receiving a response containing the Noncespaces headers from a server, the web browser must ensure that the document conforms to the policy before rendering. This requires the browser to retrieve the policy from the web server if it doesn't already have an unexpired copy in its cache. The overhead involved in policy retrieval should be minimal given that most web pages are assembled from the results of multiple requests and that we expect it to be common for a single, seldom-changing policy to be used for each web application.

4 Implementation

4.1 Server Implementation

Noncespaces requires the server to identify untrusted content in web pages. The server may choose any approach. For instance, the server may whitelist trusted content statically, or determine untrusted content dynamically by program analysis or information flow tracking. In our prototype implementation, we choose an approach that applies to a popular web application development paradigm. Current design trends for web applications advocate separating presentation from business logic. Many modern web applications employ a template system that inserts dynamic values, which business logic computes, into static templates, which decides presentation of the web page. Since web developers author templates, we may consider these templates as trusted content. By contrast, dynamic values may, and often do, come from untrusted sources, so we consider these values as untrusted content. This approach requires that scripts be placed in templates for them to be annotated as trusted content. (This requirement is reasonable because most scripts can be specified statically.)

4.1.1 NSmarty

To automatically annotate the content of web pages generated by template systems, we modified Smarty [6], a popular template engine for the PHP language. The Smarty language is a Turing-complete template language that allows dynamic inclusion of other templates. A Smarty template consists of free-form text interspersed with template tags delimited by { and }. A template tag either prints a variable or invokes a function. To use Smarty, a PHP program invokes the Smarty template engine, passes a template (or templates) to the engine, and assigns values to the template variables in the template. The template engine will then generate a document based on the template and variable substitution.

To randomize XML namespace prefixes in Smarty templates, we must be able to recognize them. Since the Smarty language allows Smarty tags to appear anywhere in a template, in element names and attribute names, we must restrict the Smarty language to be able to recognize all the XML namespace prefixes statically. Hence, we specified a subset of the Smarty language, which we call *NSmarty*. NSmarty prohibits template tags from appearing in element names or attribute names. Through these modest restrictions, we ensure that we can correctly identify all the statically specified XML tags and attributes.

The Smarty template engine operates in two phases. The first time it encounters a template, it compiles the template into PHP code and caches it. Then the PHP code runs to render the output document. On subsequent requests, the cached PHP code will rerun to render the output document, without the need to recompile the template. We provide a preprocessor to the Smarty engine, which invokes the preprocessor on the template each time before it compiles the template. Our preprocessor inserts into the template PHP code that replaces static XML namespace prefixes with random prefixes.

Ideally, we wish to map all the static prefixes that represent the same URI to the same random prefix (note that different prefixes may represent the same URI). However, since the Smarty (and also our NSmarty) language is Turing-complete, it is infeasible to determine the scope of each static prefix reliably, which implies that it is also infeasible to determine the URI that each static prefix represents. Therefore, instead we map each unique static prefix to different random prefix. This way, if the original document without prefix randomization is a well-formed XML, the new document with prefix randomization is also a well-formed XML and is semantically equivalent to the original document as long as no dynamic content (as a result of template variable substitution) contains XML tags. Figures 1 and 2 show an original XML template and the rendered document after prefix randomization respectively. Algorithm 2 in the Appendix shows the

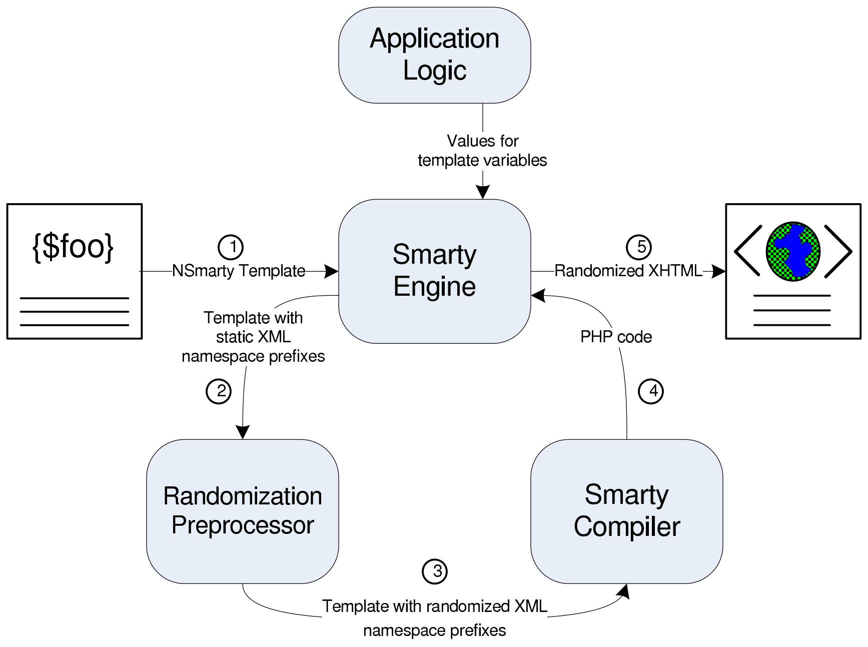
Application Logic
Values for template variables
{$foo}
1
NSmarty Template
Smarty Engine
5
Randomized XHTML
Template with static XML namespace prefixes
2
PHP code
4
Randomization Preprocessor
Smarty Compiler
3
Template with randomized XML namespace prefixes

Figure 4: Implementing Noncespaces in the Smarty template engine.

pseudocode for prefix randomization.

However, when dynamic content contains XML tags, Algorithm 2 may create ill-formed XML documents. This is because while the algorithm randomizes all the static namespace prefixes, it cannot randomize any namespace prefixes in the dynamic content. If an open tag is in the static content but its corresponding closing tag is in the dynamic content, Algorithm 2 will randomize the prefix of the open tag but not the prefix of the closing tag, resulting in an ill-formed XML document. Even though this situation is rare and is considered a bad practice, we catch this error by verifying that each document after prefix randomization is well-formed.

4.1.2 Backward Compatibility

It is easy to retrofit existing web applications with Noncespaces. Apparently, the developer needs to write a policy and, when necessary, to revise the Smarty template such that it is also a valid NSmarty template.

If the developer wishes to enforce a static-dynamic policy, where all static content in the Smarty template is trusted and all dynamic content is untrusted, he need not modify his template. Noncespaces will randomize all the static namespace prefixes. Since no namespace prefixes in the dynamic content will be randomized, they cannot gain any capabilities for trusted content as specified in the policy.

4.2 Client Implementation

The client checks the document against its policy. The check can be done either in the browser or in a proxy. Checking the policy in a proxy simplifies deployment since it requires no modification to any browser. However, the proxy and the browser may parse the same document differently in rare occasions, which may provide opportunities to attackers. Moreover, a proxy incurs runtime overhead in response time.

To overcome these problems, we can check the policy in the browser. However, this would require us to modify each browser. We decided to check policies in the proxy, since XML parsing is much stricter than HTML parsing and therefore is less susceptible to ambiguities, and we wish to deploy Noncespaces sooner to have immediate impact.

Our proxy forwards requests from a web browser to the appropriate server. When it receives a response from the server, if the response contains Noncespaces headers, the proxy attempts to check the document against the policy. If the document conforms to the policy, the proxy forwards it to the client. If the document violates the policy or fails to parse, or some other error occurs, such as the policy being malformed or inaccessible, the proxy returns an error document indicating the problem to the web browser.

5 Evaluation

To evaluate the effectiveness and overhead of Noncespaces we conducted several experiments. We evaluated the security of Noncespaces to ensure that it is able to prevent XSS attacks through various attack vectors. Our performance evaluation measures the costs of Noncespaces from both the client and server's points of view.

5.1 Security

We tested Noncespaces against six XSS exploits targeting two vulnerable applications. They were representative exploits for all the major XSS vectors discussed in Section 2. The applications used in this evaluation were a version of TikiWiki [3] with a number of XSS vulnerabilities and a custom web application that we developed to cover all the major XSS vectors.

We began by developing policies for each application. Because TikiWiki was developed before Noncespaces existed, it illustrates the applicability of Noncespaces to existing applications. We implemented a straightforward, 37-rule, static-dynamic policy that allows unconstrained static content but restricts the capabilities of dynamic content to that of bbCode (similar to Figure 3). We also had to add exceptions for trusted content that TikiWiki generates dynamically by design, such at names and values of form elements, certain JavaScript links implementing collapsible menus, and custom style sheets based on user preferences.

For our custom web application, we implemented a policy that does not take advantage of the static-dynamic model. Instead, the policy takes advantage of Noncespaces's ability to thwart node splitting attacks to implement an ancestry-based sandbox policy similar to the noexecute policy described in BEEP [9]. This policy denies script-invoking tags and attributes (e.g., `<script>` and `onclick`) that are descendants of a `<div>` tag with the `class="sandbox"` attribute. This policy consisted of 26 rules. Figure 5 shows an excerpt of the policy.

For each of the exploits we first verified that each exploit succeeded without Noncespaces randomization on the server or our client-side proxy. We then enabled Noncespaces randomization and the client-side proxy. We observed that the proxy detected all the attacks.

5.2 Performance

Our performance evaluation first seeks to measure the overhead of Noncespaces's on the server, in terms of the server's response latency, the number of requests served per second, and the time to validate that a document conforms to a policy. Our test infrastructure consisted of the TikiWiki application that we used for our security evaluation running in a VMware virtual machine with 160MB RAM running Fedora Core

```
# Deny the <script> tag
deny //*[local-name() = 'div']/@*[local-name() = 'class' and . = 'sandbox']/..//*[local-name() = 'script']
# Deny the onload attribute
deny //*[local-name() = 'div']/@*[local-name() = 'class' and . = 'sandbox']/..//@*[local-name() = 'onload']
# Deny the href attribute
deny //*[local-name() = 'div']/@*[local-name() = 'class' and . = 'sandbox']/..//@*[local-name() = 'href' \
    and starts-with(normalize-space(.), "javascript:")]
# Allow everything else
allow //*
allow //@*
```

Figure 5: Excerpt from an ancestry-based sandbox policy that denies all potential script-invoking tags and attributes that are descendants of a `<div>` node with the `class="sandbox"` attribute.

3, Apache 2.0.52, and mod_php 5.2.6. The virtual machine ran on an Intel Pentium 4 3.2GHz machine with 1GB RAM running Ubuntu 7.10. For our client machine, we used a laptop with an Intel Core 2 Duo 2.2GHz and 2GB RAM running OS X 10.4. We have spent no effort optimizing our Noncespaces prototype. In each test we used the ab (ApacheBench) [1] tool to retrieve a TikiWiki page 1000 times. We varied the number of concurrent requests between 1, 10, and 30, and the configuration of the client and server between the following:

- *No Noncespaces randomization on the server, and no proxy between the client and the server.* This configuration measures the baseline performance of the server without Noncespaces.
- *Noncespaces randomization on the server, but no proxy between the client and the server.* This configuration measures the impact of the Noncespaces randomization on server performance.
- *Noncespaces randomization on the server, and a client-side Noncespace-aware proxy between the server and the client.* This configuration measures the end-to-end performance impact of Noncespaces.

We report the median results of three trials for each test. The server and virtual machine were rebooted between tests. The target page was prefetched once before the test to warm up the systems' caches to prevent any one-time costs (such as compiling the NSmarty templates) from skewing our results.

Figure 6 shows the Cumulative Distribution Function of the time for a response to complete for our different test configurations and concurrencies. We see that for over 90% of responses, the overhead of enabling Noncespaces randomization on the server is less than 2%. Thus system administrators need not worry about significant latency due to Noncespaces randomization.

When the client is configured to check that the delivered document conforms to its policy on a proxy, the slowdown in response time is closer to 3.5x in the worst case. Even though we did not perceive any slowdown when we browsed pages on the web server interactively, we wish to determine if the slowdown was mainly caused by the policy checking code or by the architectural overhead of using a proxy. Therefore, we performed a microbenchmark. The average time to check a document retrieved in the performance tests against its policy was 1.23 seconds, which is usually much lower than the end-to-end time for fulfilling a request and is therefore likely to be tolerable for most users.

The impact of Noncespaces on server throughput can be seen in Figure 7. The leftmost bar in each group shows the baseline performance of the server without Noncespaces randomization or the client side proxy. The center bar in each group shows the performance with Noncespaces randomization enabled but no client

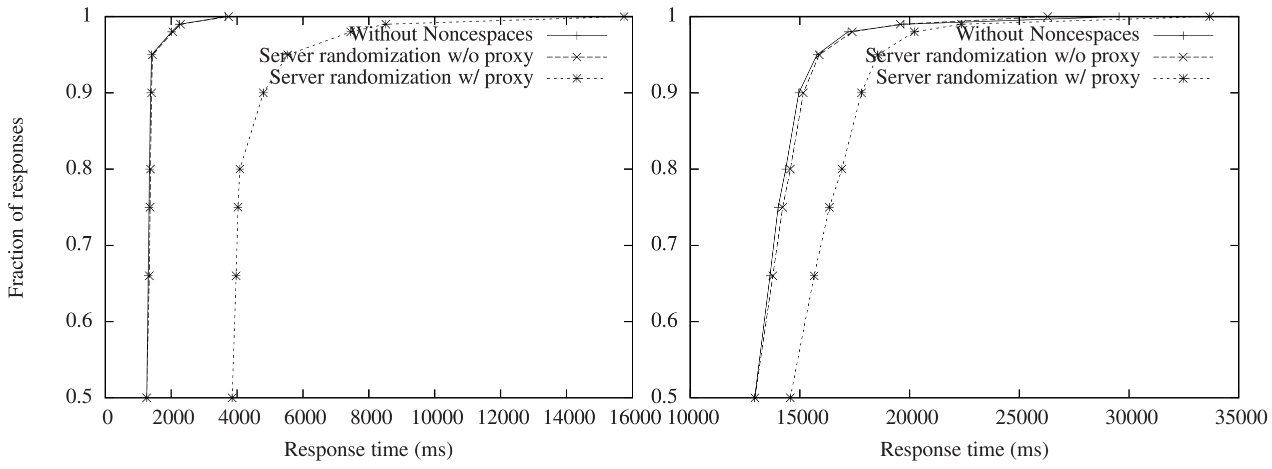

Figure 6: Cumulative Distribution Function of response times by test configuration

side proxy. And, the rightmost bar, the performance with the Noncespaces randomization enabled and client-side proxy checking. In each case, the penalty for enabling Noncespaces randomization on the server is small, 1.3% for serialized requests, no difference for 10 concurrent requests, and a 10.3% difference with 30 concurrent requests. As seen in these response times, when the client is limited to issuing requests serially, the overhead of validating proxy dominates. However, because documents can be checked independently, the reduction in throughput for concurrent requests is much less. The numbers for 30 concurrent requests should be taken with a grain of salt, as the virtual machine was swapping heavily while serving so many concurrent requests. We conjecture that swapping dominated the CPU usage in this case and caused the spurious performance differences between the three configurations.

As these tests show, the impact of Noncespaces on server performance is negligible. The client-side performance impact is more pronounced, though acceptable for interactive use.

5.3 Compatibility

Noncespaces requires that the browsers are XHTML 1.0 compliant. These browsers will reject ill-formed XML documents, and therefore can defeat all node-splitting attacks, even if the browser is not Noncespaces-aware (i.e., neither the browser or a client proxy checks Noncespaces's policy). Most modern browsers are XHTML 1.0 compliant.

To check a document against a Noncespaces policy, the browser or the proxy must be XHTML modularization 1.1 conformant [7], which requires it to process the namespace prefixes of both tags and attributes correctly. [2]

[2]XHTML Modularization 1.1 Conformance definition 3.1(5) states: The schema that defines the document type may define additional elements and attributes. However, these MUST be in their own XML namespace. If additional elements are defined by a module, the attributes defined in included XHTML modules are available for use on those elements, but SHOULD be referenced using their namespace-qualified identifier (e.g., xhtml:class). The semantics of the attributes remain the same as when used on an XHTML-namespace element.

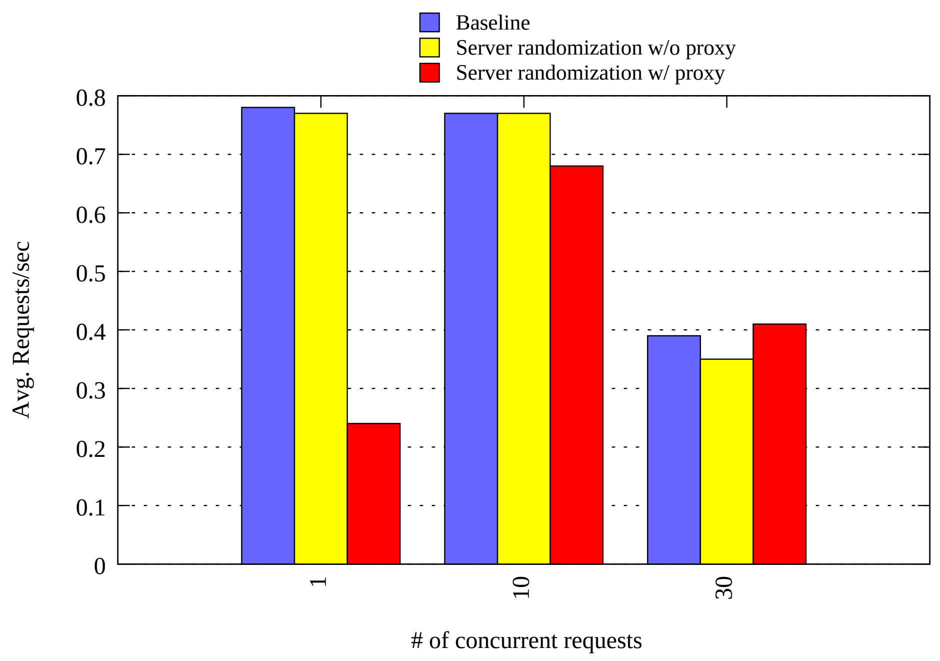

Figure 7: Average requests served per second in each configuration vs. concurrency

6 Security Analysis

6.1 Threat Model

The goal of Noncespaces is to defend against XSS attacks. We assume that the attacker can only submit malicious data to XSS-vulnerable web applications. We assume that the attacker cannot otherwise compromise the web server or client via buffer overflow attacks, malware, etc.

6.2 Identifying Untrusted Content

The core idea of Noncespaces is to use randomized namespace prefixes to annotate trusted data and to prevent malicious data from escaping its containing node. As long as the attacker cannot guess the randomized prefixes for trusted content, the attacker cannot change the classification of his untrusted content. Since the server randomizes the prefixes differently each time it serves a page, the attacker would not gain an advantage by viewing previous renderings of the page that he wishes to attack.

In our prototype, we use an approach that identifies trusted content in template systems. Since our language, NSmarty, requires constant strings for tag and attribute names, we can identify all the trusted elements and attributes reliably.

Our prototype conservatively classifies all the content that might have user-contributed data as untrusted. This is safe, but it might restrict rich content in documents. For example, consider the following content in a template: `<a onclick='toggle("{id}")'>foo</a>`. Since the value of the attribute `onclick` consists of both static JavaScript code and a template variable `id`, Algorithm 2 conservatively, and often rightly, considers this attribute untrusted. If the policy denies `onclick` in untrusted content, the client will reject this document, even when this JavaScript code is harmless. We propose two solutions. First, the client could ignore the content that the policy denies but render the rest of the document, rather than rejecting the entire document. This solution may be acceptable in many situations. The advantage of this solution is that it requires no change to how we identify untrusted content. Second, the web application could whitelist certain untrusted content, after either proper sanitization or ensuring that it contains no malicious input by program analysis or information flow tracking. This solutions requires slight modification to Algorithm 2: when Algorithm 2 determines if the value of an attribute is static (Line 7), it should also consult the whitelist.

6.3 Enforcing Security Policy

The client enforces the security policy on the documents. Its security depends on the correctness of the policy and the correctness of enforcement. Noncespaces does not dictate any specific security policy. Either the server or the client may design proper policies that sufficiently restrict the capabilities of untrusted content.

A Noncespaces-aware client may reject an XML document for either of two reasons: (1) the document is not well-formed; or (2) the document violates the policy. Both of these cases may indicate an attack. In the first case, the attacker may have tried to inject a close tag to escape from its enclosing node. However, since he cannot guess the random prefix of the tag of the node, his injected close tag causes an XML parsing error. In the second case, the attacker may have injected content that requires higher capabilities than what the policy allows. Interestingly, even if a client is not Noncespaces-aware, it can still reject a malicious document in the first case above, as long as the client is XML 1.0 compatible. The first case is also known as a “node-splitting attack”. Therefore, a Noncespace-aware server can prevent node-splitting attacks even if the client is not Noncespace-aware.

The client must parse XML properly. Since HTML parsers are lenient, attackers have exploited the discrepancies between different parsers. By contrast, XML is much stricter, which results in significantly fewer, if any, discrepancies between different parsers.

7 Related Work

Our work was inspired by Instruction Set Randomization (ISR) [10] – a technique for defending against code injection attacks in executable. ISR randomly modifies the instruction set architecture of a system for each running process. As long as an attacker cannot guess the randomization employed, the attacker will not be able to inject code with meaningful semantics. Noncespaces is an analogous approach for web applications. After the server randomizes the namespace prefixes in each document, it will be simple for the client to differentiate injected content from trusted content. Noncespaces further expands the ISR idea by using a policy to constrain the capabilities of untrusted content while allowing rich trusted content. The Noncespaces policy language allows the application developer to decide what types of untrusted content to permit in each application setting.

Client-side policy enforcement mechanisms enforce a security policy in to avoid the semantic gap between the way a web application intends content to be interpreted and how the client actually interprets it. For example, BEEP [9] allows a server-specified JavaScript security handler to decide whether to permit or deny the execution of each subsequent script based on a policy. The BEEP authors present two example policies: an ancestry-based sandbox policy, which prohibits scripts that are descendants of a sandbox node from running, and a whitelist policy, which allows a script to execute only if it is known-good. Similar to BEEP, in Noncespaces the server delivers the policy and the client enforces it. Like BEEP, our policy language is able to express both ancestry-based sandbox and whitelist policies. Our policy language is also able to express policies which constrain non-script content of a web page. This is important because malicious non-script content may cause security vulnerabilities. For instance, an attacker could steal login credentials by injecting a fake login form onto a bank's website even if the attacker cannot inject scripts. More importantly, since Noncespaces annotates the trust classification of web content, it allows easy expression and enforcement of trust class based policies, which are a natural fit for defeating XSS attacks.

Advanced template systems such as Genshi [5] and static analysis techniques such as that used in [12] have considered the problem of ensuring that output documents are well-formed and valid. Genshi attempts to ensure all output documents are well-formed by requiring all templates to be valid XML document fragments. Genshi employs context-sensitive output sanitization to ensure that web developers do not accidentally include unsanitized output into their output documents. However, Genshi is unable to prevent incomplete sanitization by the web application, especially when there is discrepancy between how the server and client interpret data. When improperly sanitized content arrives at the client, the client cannot distinguish untrusted content from trusted content. In Noncespaces, we chose not to require the templates to be XML document fragments to support a large number of existing applications whose templates do not meet this requirement.

Static analysis techniques like those presented by Kirkegaard and Møller [12] could be employed to defend against XSS attacks. When faced with the undecidability of the general problem of determining if a program's output will be valid with respect to a language model, rather than pursuing statically-provable guarantees, we have focused on light-weight techniques with flexible policies. In addition, in our desire to produce a practical tool, we chose to avoid expensive, overly-conservative analyses. With Noncespaces's client-side policy enforcement, even if the web application can produce output that is not well-formed or valid with respect to a safe subset of the language, a properly written policy will be able to reject such documents.

Two main goals of XSS attacks are stealing the victim user's confidential information and invoking malicious operations on the user's behalf. Noxes provides a client-side web proxy to block URL requests by malicious content using manual and automatic rules [11]. Vogt et al. track the flow of sensitive information in the browser to prevent malicious content from leaking such information [16]. Both of these projects defeat only the first goal of XSS attacks. By contrast, Noncespaces can defeat both goals of XSS attacks because it prevents malicious content from being rendered.

8 Conclusion

We have presented Noncespaces, a technique for preventing XSS attacks. The core insight of Noncespaces is that if the server can reliably identify and annotate untrusted content, the client can enforce flexible policies that prevent XSS attacks while allowing rich safe content. The core technique of Noncespaces uses randomized XML namespace prefixes to identify and annotate untrusted content, similar to the use of Instruction Set Randomization to defeat injected binary code attack. Noncespaces is simple. The server need not sanitize any untrusted content, which avoids all the difficulties and problems with sanitization. Once the server annotates a node as untrusted, no malicious content in the node may escape the node or raise its trust classification. A Noncespaces-aware client can reliably prevent all the attacks that the policy denies. Even if a client is not Noncespaces-aware, it can still prevent the node-splitting attack, a form of XSS that is otherwise difficult to defeat. We implemented a prototype of Noncespaces on a template system on a web server and on a proxy at the client side. Experiments show that the overhead of Noncespaces is moderate.

References

[1] ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html.

[2] BBCode. http://www.phpbb.com/community/faq.php?mode=bbcode.

[3] TikiWiki CMS/Groupware. http://info.tikiwiki.org/tiki-index.php.

[4] Technical explanation of the MySpace worm, February 2006. http://web.archive.org/web/20060208182348/namb.la/popular/tech.html.

[5] Genshi: Python toolkit for generation of output for the web, 2008. http://genshi.edgewall.org/.

[6] Smarty Template Engine, June 2008. http://www.smarty.net/.

[7] XHTML Modularization 1.1, June 2008. .

[8] Tim Bray, Dave Hollander, Andrew Layman, and Richard Tobin. Namespaces in XML 1.0 (Second Edition). Technical report, W3C, August 2006. http://www.w3.org/TR/REC-xml-names/.

[9] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[10] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, New York, NY, USA, 2003. ACM.

[11] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, Dijon, France, April 2006.

[12] Christian Kirkegaard and Anders Møller. Static analysis for Java Servlets and JSP. In *Proc. 13th International Static Analysis Symposium, SAS '06*, volume 4134 of *LNCS*. Springer-Verlag, August 2006. Full version available as BRICS RS-06-10.

[13] MITRE Corporation. Vulnerability Type Distributions in CVE, May 2007. http://cwe.mitre.org/documents/vuln-trends/index.html.

[14] RSnake. XSS (Cross Site Scripting) Cheat Sheet, June 2008. http://ha.ckers.org/xss.html.

[15] Ofer Shezaf. The Universal XSS PDF Vulnerability, January 2007.

[16] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2007.

Appendix

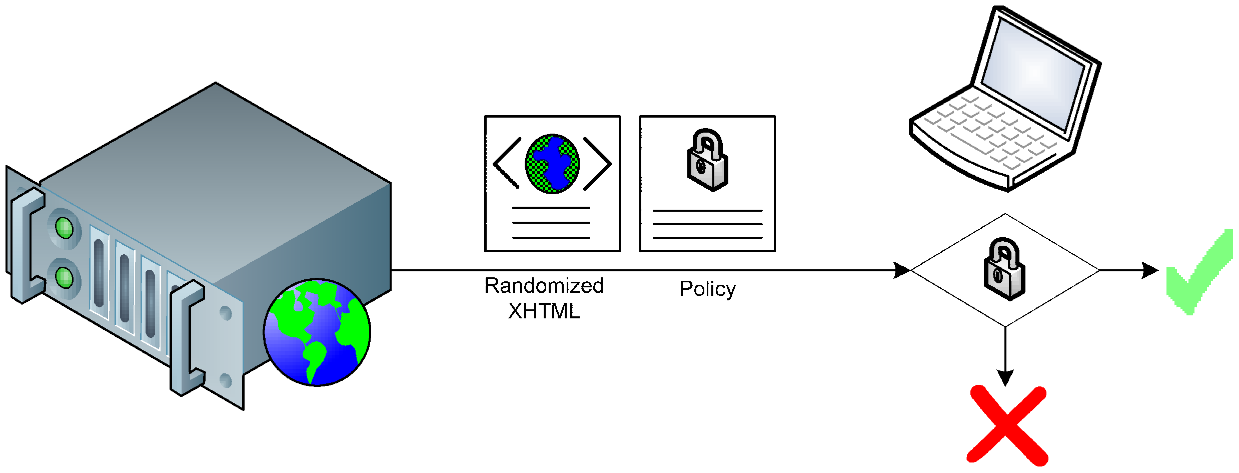

Figure 8: Noncespaces Overview. The server delivers a XHTML document with randomized namespace prefixes and a policy to the client. The client accepts the document only if it is a well-formed XML document and satisfies the policy.

```
Input  : A document $d$ and a policy $p$.
Output: TRUE if the document $d$ satisfies the policy $p$; FALSE otherwise.
begin
    for Element or attribute node $n \in d$ do
        $n.checked = FALSE$
    end
    for Rule $r \in p.rules$ do
        for Node $n \in d.matchNodes(r.XPathPattern)$ do
            if $n.checked == FALSE$ then
                if $r.action == ALLOW$ then
                    n.checked = TRUE
                else
                    return FALSE
                end
            end
        end
    end
    for Element or attribute node $n \in d$ do
        if $n.checked == FALSE$ then
            return FALSE;
        end
    end
    return TRUE;
end
```

Algorithm 1: An algorithm for checking whether a document satisfies a policy

```
Input  : An XML document $d$
Output: The document $d$ after prefix randomization
begin
    for Tag $t \in d$ do
        for Attribute $a \in t$ do
            if $a$ is a namespace declaration then
                map[a.prefix] = random()
                a.prefix = map[a.prefix]
            else if $a.value$ is static (i.e. containing no template tag) then
                a.prefix = map[a.prefix]
        end
        t.prefix = map[t.prefix]
    end
end
```

Algorithm 2: An algorithm for randomizing XML namespace prefixes