Available online at www.sciencedirect.com

SciVerse ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Noncespaces: Using randomization to defeat cross-site scripting attacks

*Matthew Van Gundy**, *Hao Chen*

Department of Computer Science, University of California, One Shields Ave., Davis, CA 95616-8562, USA

ARTICLE INFO

Article history:
Received 8 April 2011
Received in revised form
5 November 2011
Accepted 5 December 2011

Keywords:
Security
Defense
Cross-site scripting
Client-side policy enforcement
Information flow tracking
Web application
World wide web

ABSTRACT

Cross-site scripting (XSS) vulnerabilities are among the most common and serious web application vulnerabilities. It is challenging to eliminate XSS vulnerabilities because it is difficult for web applications to sanitize all user input appropriately. 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 the (X)HTML tags and attributes in each document before delivering it to the client. As long as the attacker is unable to guess the random mapping, the client can distinguish between trusted content created by the web application and untrusted content provided by an attacker. 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 design a policy language for Noncespaces, implement a training mode to assist policy development, and conduct extensive security testing of a generated policy for two large web applications to show the effectiveness of our technique.

1. Introduction

Cross-site scripting (XSS) vulnerabilities pose a serious threat to the security of modern web applications. Year after year, XSS vulnerabilities top lists of the most dangerous and the most commonly reported vulnerabilities (CWE, 2010; MITRE Corporation, 2007). They are surprisingly easy to create and difficult to mitigate completely. Any web application that fails to properly sanitize user input before displaying it to other users will be vulnerable to XSS attacks.

Web browsers protect multiple web applications running within the same browser instance by isolating them according to the Same Origin Policy. The Same Origin Policy prevents web applications from accessing the private data of other web applications. However, the Same Origin Policy presumes that all content from a single web application is equally trustworthy and can be granted access to all data associated with the web application. A cross-site scripting (XSS) vulnerability allows an attacker to inject malicious content into web pages served by a trusted web application. Because the browser receives the malicious content from a trusted server, the malicious content will run with the same privileges as trusted content allowing it to run malicious code within the browser, impersonate the user to trusted servers, steal a victim user's private data and authentication credentials, or present forged content to the victim.

Fig. 1 shows an example web page template like those used by many web applications to render dynamic web pages. A sequence of the form `{x}` will be replaced at runtime by the value of the variable `x`. For instance, if an attacker can submit `<script src='http://badguy.com/attack.js'/>` as a review, the template variable review.text will be replaced

* *Corresponding author*. Tel.: +1 805 699 6134; fax: +1 865 357 7210.
E-mail addresses: mdvangundy@ucdavis.edu (M. Van Gundy), hchen@cs.ucdavis.edu (H. Chen).
0167-4048/$ – see front matter © 2012 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2011.12.004

```
<!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" lang="en">
<head>  <title>nile.com : ++Shopping</title>  </head>
<body>  <h1 id="title">{item_name}</h1>
  <p class='review'>{review.text}
     -- <a href='{review.contact}'>{review.author}</a>  </p>
</body>
</html>
```

Fig. 1 – Vulnerable web page template. This template is used to render dynamic web pages. It is written in a Smarty-like language where the appearance of the token `{x}` instructs the template engine to replace the token by the value of the variable named "x".

with this script tag. When a victim visits the page, the victim's web browser will download and execute http://badguy.com/attack.js with the same permissions as legitimate scripts. It has long been recognized that non-script elements pose a threat as well. (CERT Coordination Center, 2000; The Open Web Application Security Project, 2010) For instance, an attacker could inject a fake login form and style it to obscure a legitimate login form. When the victim attempts to login, their credentials could be sent to a site of the attacker's choosing. Nevertheless, many existing XSS defenses focus on preventing the execution of untrusted scripts without addressing malicious non-script content.

To prevent XSS vulnerabilities, all the untrusted (user-contributed) content in a web page must be sanitized. However, proper sanitization is very challenging. The context in which untrusted data is interpreted determines the forms of sanitization that are appropriate. If sanitization is performed by the server, but the browser interprets the content in a way that the server did not intend, there are many ways for an attacker to take advantage of this discrepancy (RSnake, 2008). The Samy worm (Samy, 2006), one of the fastest spreading worms to date, used this ambiguity between server sanitization and client parsing to propagate. 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 a web page since both appear to originate from the trusted server.

We can avoid ambiguity between the client and server by requiring the server to identify untrusted content and requiring the client to ensure that it is displayed safely. However, challenges remain. 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 a Node-splitting attack (Jim et al., 2007). To defend against this attack without restricting the richness of user provided content, the server must take care to remove only those node delimiters which would introduce new trusted nodes.

We present Noncespaces, an end-to-end mechanism that allows a server to identify untrusted content, to reliably convey this 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 (Kc et al., 2003; Barrantes et al., 2003), which randomizes the processor's instruction set to identify and defeat injected malicious binary code. Analogously, Noncespaces randomizes (X)HTML tags and attributes to identify and defeat injected malicious web content. Randomization serves two purposes. First, it identifies untrusted content so that the client can use a policy to limit the capabilities of untrusted content. Second, it prevents 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 parsing errors.

We make the following contributions:

- We leverage the similarities between injected code in executable programs and injected content in web pages to apply techniques from Instruction Set Randomization to defend 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 design a flexible yet simple language for specifying common security policies. We then extend the basic language to support organizing trust classes into an arbitrary lattice for expressing a wider range of information flow policies. In contrast to most existing XSS defenses, our technique allows prevention of both script and non-script XSS attacks.
- We create a training mode to facilitate rapid policy development.
- We demonstrate the effectiveness and usability of Noncespaces by porting a 155 K SLOC blog application to work with Noncespaces, using the training mode to develop a policy for the application, and conducting an extensive security evaluation with the generated policy.

2. Threat model

In this paper, we restrict our attention to XSS attacks where the malicious content is unintentionally delivered to the victim user by a trusted server. Specifically, our solution addresses reflected (Type I) and stored (Type II) XSS attacks. In a reflected XSS attack, the victim visits a page controlled by the attacker. The attacker encodes malicious content into a link or web form that targets a trusted web site. The victim clicks the link or submits the form and the malicious content is reflected by the trusted web server back to the victim's web browser. In a stored XSS attack, the attacker causes malicious content to be stored directly on a trusted web server. Later, when the victim visits the trusted web server, the attacker's malicious content will be delivered to the victim's web browser.

We do not address DOM-based (Type III) XSS attacks, where trusted client-side JavaScript permits the injection of untrusted content in violation the web application's security policy; Universal XSS Vulnerabilities (Shezaf, 2007), where a browser extension can be tricked into violating the

browser's own security policies; or Cross-Site Request Forgery (CSRF), where a malicious web server tricks the client into sending a malicious request to a trusted server. Though we do not address DOM-based XSS and Universal XSS attacks in this work, our approach can be employed as a component in solutions to these problems.

Because the goal of Noncespaces is to defend against XSS attacks, we assume that the attacker's only means of attack is to submit malicious data to XSS-vulnerable web applications. Existing mechanisms can be used to ensure that an attacker cannot compromise the web server or browser directly via buffer overflow attacks, malware, etc.

3. Related work

Given the high impact of XSS attacks, there is a significant amount of related work. Here we attempt to compare our work with a sampling of other relevant work.

3.1. *Randomization*

Our work was inspired by Instruction Set Randomization (ISR) (Kc et al., 2003; Barrantes et al., 2003) — a technique for defending against code injection attacks in executables by randomly remapping a computer's instruction set architecture. SQLrand (Boyd and Keromytis, 2004) first employed randomization to defeat SQL Injection attacks. Noncespaces is an analogous approach that protects web applications from Cross-Site Scripting attacks. ISR and SQLrand prevent an attacker from injecting meaningful code and SQL keywords by forcing an attacker to guess a random mapping. Noncespaces also forces an attacker to guess a random mapping in order to inject trusted content. ISR and SQLrand consider static program code and SQL queries trustworthy. Similarly, our prototype considers all static (X)HTML template content to be trustworthy. Unlike ISR or SQLrand, Noncespaces must support web applications which permit rich user input. Therefore, Noncespaces expands the ISR approach by using a configurable policy to constrain the capabilities of untrusted content on the client side.

3.2. *Preserving document structure integrity*

Document Structure Integrity (DSI) (Nadji et al., 2009) was developed independently and contemporaneously with our work. Each system has advantages over the other in different areas. Like Noncespaces, DSI uses randomized delimiters to allow a web browser to distinguish between trusted and untrusted content. In DSI, the server identifies untrusted content using a prototype taint tracking implementation for PHP. The browser enforces a simple policy that limits untrusted content to terminals in (X)HTML and JavaScript and to tags and attributes whitelisted on a per-page basis. DSI also augments the browser with information flow tracking in order to defeat DOM-based (Type III) XSS attacks. Noncespaces's policy language is more expressive than the policy language provided by DSI. DSI's policy language does not capture position of whitelisted elements or provide an ability to constrain terminal values. Both of these capabilities are important for defeating injection of non-script elements. In many scenarios, non-code injection attacks can be just as dangerous as code injection attacks (Chen et al., 2005). For instance, an attacker can steal login credentials by injecting a fake login form onto a bank's website. The whitelisting approach employed by Noncespaces is superior to the blacklisting approach employed by DSI (called "minimal-serialization") insomuch as it respects the Principle of Fail-Safe Defaults. If the classification mechanism is incomplete, Noncespaces would classify trusted data as untrusted and therefore might refuse to render a legitimate page, but DSI would classify untrusted data as trusted, resulting in security vulnerabilities.

Before Noncespaces and DSI, work on ensuring document structure integrity typically focused on ensuring that output documents were valid (Kirkegaard and Møller, 2006) and that web designers would not inadvertently print unsanitized output to the browser (Genshi, 2008). However, neither approach is sufficient to mitigate XSS attacks.

The Noncespaces and DSI approach of defeating XSS by ensuring document structure integrity has appeared in subsequent research. Blueprint (Ter Louw and Venkatakrishnan, 2009) provides a DSI-like terminal confinement and no script policy mechanism for unmodified modern browsers. An application developer manually annotates all web application statements that output untrusted content. Untrusted content will then be transmitted to the browser where client-side JavaScript ensures that it cannot invoke the JavaScript interpreter. By contrast, Noncespaces does not require the developer to manually identify and modify untrusted output statements. Noncespaces integrates with the Smarty template engine, allowing it to intercept all untrusted template outputs automatically. Because Blueprint only encodes untrusted content, incomplete sanitization can result in successful XSS attacks. Noncespaces encodes trusted data to ensure that any failure of complete mediation will not result in a successful attack. Noncespaces also provides a significantly more flexible policy mechanism.

SWAP (Wurzinger et al., 2009) takes a complementary approach on the server-side—it attempts to whitelist all trusted scripts. SWAP identifies all static scripts and replaces them with non-executable script identifiers. Before delivering the response to the browser, SWAP invokes a server-side script detector (consisting of a browser residing on the server) to determine if any scripts have been injected. If no scripts are detected, the script identifiers in the response are replaced with the original scripts and the response is delivered to the client. Both Blueprint and SWAP incur higher server-side overheads than Noncespaces because they perform all policy enforcement on the server, preventing clients from sharing the computational burden. Noncespaces's client-side enforcement also allows for the possibility of browser or user-contributed policies. Also, like DSI, Blueprint and SWAP do not defend against non-script attacks.

Alhambra (Tang et al., 2010) is a pragmatic approach to ease deployment of a document structure integrity system. It is an entirely client-side mechanism that attempts to infer a document structure integrity policy from multiple web page visits. Client-side information flow tracking is also employed to prevent DOM-based XSS attacks and certain common script abuse patterns. Alhambra's learning capabilities can be

compared to a more advanced rendition of Noncespaces's training mode. Greater learning algorithm intelligence is necessary for Alhambra because, unlike the training environment in Noncespaces, the resulting policy is not reviewed by an expert before being employed to defend against attacks. Alhambra's client-side only approach imposes several additional challenges including the ability of a malicious attacker to mislead the policy learner and attempting to remain robust to legitimate website modifications.

Robertson and Vigna present another approach for ensuring the structural integrity of a document by defining a strongly-typed web application framework (Robertson and Vigna, 2009). Instead of generating unstructured strings, applications output an Abstract Syntax Tree. The AST is then translated by a trusted renderer which ensures that all content incorporated into document terminals is correctly escaped. This prevents attacks which rely on violating the document's structural integrity. In addition to defeating attacks that violate structural integrity, Noncespaces goes one step further by protecting against attacks which leverage unsafe attribute values and vulnerabilities caused by web applications that permit an untrusted user to create security-sensitive structural content.

3.3. *Client-side policy enforcement*

Client-side policy enforcement mechanisms enforce a security policy in the browser to avoid the semantic gap between the client and server. BEEP (Jim et al., 2007) allows a server-specified JavaScript security handler to decide whether to permit or deny the execution of each script based on a programmable policy. The BEEP authors present two example policies: an ancestry-based sandbox policy, which prohibits scripts that are descendants of a sandbox node, and a whitelist policy, which allows a script to execute only if it is known-good. Mutation Event Transforms (Erlingsson et al., 2007) extend the mechanism of BEEP to all operations that modify the DOM. Before each DOM modification, a JavaScript callback is invoked that can allow, deny, or arbitrarily change the operation performed.

Noncespaces is similar to both of these approaches in that the server delivers a policy that the client enforces. Like BEEP, our policy language is able to express both ancestry-based sandbox and whitelist policies. Additionally, like Mutation Event Transforms, our policy language is also able to express policies which constrain non-script content of a web page. This is important because, as mentioned above, malicious non-script content can successfully exploit security vulnerabilities. It would have been possible to leverage Mutation Event Transforms as our client-side policy mechanism. However, giving policies the full power of JavaScript would make it hard to reason about a policy's effects and could also provide a new vector for bugs and vulnerabilities to be introduced. This is why our client-side policy language consists of simple rules that match nodes and attributes in the DOM and declare whether they should be allowed or denied. We also note that the main contributions of our work are a mechanism for reliably communicating trust information from server to client and leveraging properties of the web application to determine trustworthiness of content automatically. Neither BEEP nor Mutation Event Transforms addresses these issues.

Noncespaces is also closely related to Content Restrictions (Markham, 2007), Mozilla's Content Security Policy (CSP) (Stamm et al., 2010), Script Keys (Markham, 2005), and Brendan Eich's proposed <jail> tag (Eich, 2007). Content Restrictions allow the server to specify certain restrictions on the content that it delivers, such as: whether scripts may appear in the document body, header, only externally, or not at all; which hosts resources may be fetched from; which hosts scripts may be fetched from; etc. Mozilla's Content Security Policy is an implementation of Content Restrictions for Firefox with some additional features. Noncespaces client-side policies are able to specify most of the same restrictions as Content Restrictions and CSP. CSP provides a few features outside the domain of Noncespaces. However, Noncespaces can prevent malicious content from attacking trusted origins or exfiltrating data to untrusted domains via forms and links, but CSP cannot. Content Restrictions and CSP also do not provide a mechanism for differentiating between server-trusted content executing a script in an approved location or injected content doing the same. Both Script Keys and Noncespaces provide a way to differentiate between the two scenarios.

Script Keys prohibits scripts from running unless they include a server-specified key in their source. The proposed <jail> tag does just the opposite: it prohibits active content in the document subtree below it and uses a nonce embedded in the opening and closing tags to prevent a node-splitting attack from closing a <jail> tag prematurely. In the limit, when the script key is changed on every page load, Script Keys behaves like Noncespaces — the attacker must guess the randomly generated key for each request to enable their script to run. To an extent, Noncespaces can be seen as the first implementation of Content Restrictions, Script Keys, and the <jail> tag. However, none of these other proposals provide a means to restrict non-script content with the same level of precision as Noncespaces. Noncespaces and CSP may be useful in conjunction with one another allowing specification of policy constraints at the most natural layer.

ConScript (Meyerovich and Livshits, 2010) enables client-side policy enforcement for JavaScript code by providing an Aspect Oriented Programming model for JavaScript. Noncespaces's client-side policy enforcement and ConScript are orthogonal. Noncespaces determines whether or not a tag or attribute should be added to the DOM. ConScript constrains the behavior of scripts after that point. ConScript also only effects the execution of scripts and thus does not help defend against non-script attacks.

3.4. *Prohibiting anti-patterns*

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 (Kirda et al., 2006). Vogt et al. track the flow of sensitive information in the browser to prevent malicious content from leaking such information (Vogt et al., 2007). 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. Internet Explorer 8's XSS filter (Ross, 2008) attempts to prevent reflected XSS attacks by disabling scripts that embed certain substrings seen in an outgoing request. This only prevents reflected XSS attacks while Noncespaces is also able to prevent stored XSS attacks. Attackers have also been able to leverage the semantic gap between the server and the client to cause IE 8's XSS filter to create new security vulnerabilities (Nava and Lindsay, 2010) illustrating the need for and end-to-end solution such as Noncespaces.

3.5. *Leveraging language techniques*

SQLCheck (Su and Wassermann, 2006) defines a sub-language of SQL that untrusted user input can safely express. In theory, the same approach can be employed to prevent XSS attacks; however, specifying an appropriate subset of (X)HTML is more difficult because web application security policies cut across multiple languages (including JavaScript, URIs, and CSS), can depend on position in the document hierarchy, and may depend on specific terminal values. Therefore, we believe that expressing a policy in terms of XPath expressions is more straightforward for web developers than modifying a unified grammar for web pages in order to restrict untrusted input to a suitably safe subset for each web application.

3.6. *Static analysis*

Numerous papers (Wassermann and Su, 2007, 2008; Xie and Aiken, 2006; Livshits and Lam, 2005) have employed static program analysis to detect XSS vulnerabilities. Static analysis approaches cannot be both sound and complete, forcing a choice between false positives or missed vulnerabilities. By favoring a dynamic analysis approach, Noncespaces avoids loss of precision due to round-trips to the browser and difficult to support PHP features. Our use of NSmarty to determine trust classifications is a conservative approximation; however, if greater precision is needed, our technique can be integrated with a full information flow tracking system.

3.7. *Information flow tracking*

A number of different information flow (or taint) tracking solutions for web applications have appeared in the literature (Nguyen-Tuong et al., 2005; Venema, 2008; Xu et al., 2006). Unfortunately, none of the solutions for PHP have seen widespread use. This prompted our development of NSmarty to simply and conservatively approximate information flow by leveraging a common web application programming paradigm. The encoding and client-side policy enforcement mechanisms that Noncespaces provides can use a mature information flow tracking system as a content classifier, when one arises.

A preliminary version of this work was presented at NDSS 2009 (Van Gundy and Chen, 2009). Since then we have extended the policy language to support hierarchical trust classes and XML namespace-specific policy rules, which are important for handling real-world web applications. During the research for (Van Gundy and Chen, 2009), we found that it could be difficult to create complete policies for large web applications. Therefore, we have implemented a training mode to facilitate policy development (Section 5.3). Finally, to demonstrate the effectiveness, usability, and backward compatibility of Noncespaces, we ported a large web application to Noncespaces and conducted a more extensive security evaluation using a policy developed with our new training mode (Section 6.1.2).

4. Noncespaces

The goal of Noncespaces is to allow the client to safely display documents that contain both trusted content generated by a web application and untrusted content provided by untrusted users. To eliminate the client-server semantic gap and to adapt to differing security needs, the browser enforces a configurable security policy. The policy specifies the browser capabilities that each type of content can exercise. In this way, malicious content injected by an attacker is restricted to the capabilities allowed to untrusted content by the policy.

If the client is to faithfully enforce a server-specified policy, the client must be able to determine the trustworthiness of all content in a document. Therefore, the server must first classify content into discrete trust classes. The server then must communicate the content, trust classification, and policy to the client. Finally, the client can enforce the policy. This process is depicted in Fig. 2.

Our architecture permits Noncespaces to defeat both reflected and stored XSS attacks. In both scenarios, untrusted user input is returned to a victim user—immediately in the case of a reflected XSS attack or at some later time in the case of a stored XSS attack. In either case, as long as the server's content classification is conservative, the server faithfully communicates its classifications to the client, and the client faithfully enforces the server-specified policy, untrusted content will be confined to the capabilities expressly permitted to it by the policy, ensuring that XSS attacks will not succeed.

We take a modular approach to classifying content. The server can use a variety of techniques to determine the trust classes of content, ranging from whitelisting known-good code to annotating output based on program analysis or information flow tracking. We present an expedient approach in Section 5. In this section, we describe our mechanisms for communicating trust information and policy enforcement.

4.1. *Communicating trust information*

There are a variety of mechanisms that a server might use to indicate content trust information to clients. The server could use a designated attribute to indicate the trust class of an element. However, malicious content may contain elements which forge the attributes that designate trusted content. Alternatively, the server could indicate the trustworthiness of content by its location in the document, e.g. restricting the capabilities of all descendants of a specific document node — a sandbox node (Eich, 2007; Markham, 2007; Jim et al., 2007). However, malicious content may contain tags that split its original enclosing node into multiple nodes so that malicious nodes are no longer descendants of the sandbox node. This is the node-splitting attack discussed in Section 1.

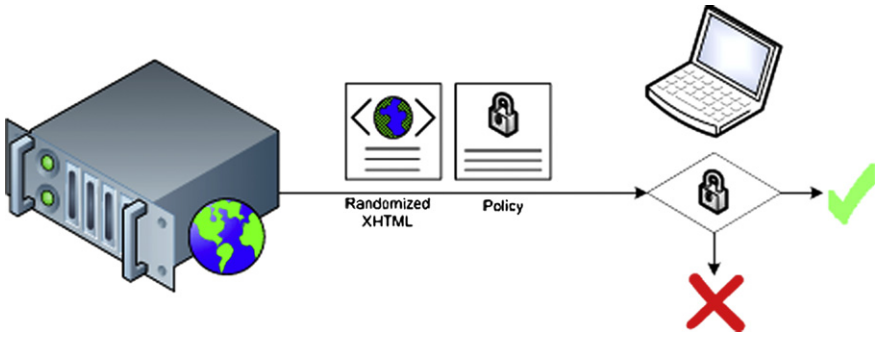

Fig. 2 – Noncespaces overview. The server delivers an (X)HTML document annotated with trust class information and a policy to the client. The client accepts the document only if it satisfies the policy.

```
<!DOCTYPE html>
<r617html r617lang="en">
<r617head>  <r617title>nile.com : ++Shopping</r617title>  </r617head>
<r617body>  <r617h1 r617id="title">Useless Do-dad</r617h1>
  <r617p r617class='review'></p><script>attack()</script><p>
    -- <r617a href=''></r617a>  </r617p>
</r617body>
</r617html>
```

Fig. 3 – HTML document randomized by Noncespaces. A random prefix has been applied to trusted HTML content in a template like that of Fig. 1. The rendered document contains a node-splitting attack injected by a malicious user.

```
<!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" r617: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:p r617:class='review'></p><script>attack()</script><p>
    -- <r617:a href=''></r617:a>  </r617:p>
</r617:body>
</r617:html>
```

Fig. 4 – XHTML document randomized by Noncespaces. A random namespace prefix has been applied to trusted XHTML content in the template of Fig. 1. The rendered document contains a node-splitting attack injected by a malicious user.

Another alternative, inspired by Instruction Set Randomization (ISR), is to greatly reduce the probability of a successful attempt to forge trust class information by preventing an attacker from being able name trusted content. ISR defends against binary code injection attacks by randomly perturbing the instruction set of an application. To inject code with predictable semantics into the application, the attacker must correctly guess the randomization used. This is very difficult if the number of possible randomizations is sufficiently large. The attacker is effectively prevented from injecting code. We propose a similar technique for (X)HTML. We associate a different randomization function with each content trust class. The names of all elements and attributes in a trust class are remapped according to the associated randomization function so that no injected content can correctly name. (X)HTML elements or attributes in other trust classes.

For example, let the randomly chosen string `r617` denote trusted content. We can defeat XSS attacks against the document from Fig. 1 by annotating it. For HTML documents, we can prefix trusted tags and attributes with our random identifier as shown in Fig. 3. For XHTML documents, we can preserve the original XML semantics of the document while annotating by using our random identifier as an XML namespace prefix[1] as shown in Fig. 4.

As illustrated by the embedded node-splitting attack, the attacker cannot inject malicious content and cause it to be interpreted as trusted because he does not know the random prefix. He also cannot escape from the enclosing paragraph element, because he does not know the random prefix and therefore cannot embed a closing tag with this prefix. (In the HTML document, the `<script>` element is the child of an `<r617p>` element, not a `<p>` element. In the XHTML document, when a closing tag tries to close an open tag but the prefixes of the two tags do not match, the XML parser will fail with an error[2].)

To prevent an attacker from guessing (namespace) prefixes, we choose the prefixes uniformly at random every time a response is rendered — hence the term Noncespaces. Given a prefix space of appropriate size, knowing the random prefixes in one instance of the document does not help an attacker predict prefixes in future instances of the document.

There is an additional complication, however. Naïvely prohibiting all untrusted content will not work because most modern web applications are designed to accept some amount of rich content from users. Though we can use

[1] To permit mixing of XML languages in a single document, XML namespaces (Bray et al., 2006a) can be used to designate elements and attributes by namespace URI and local name. The URI for the language is bound to a prefix using the `xmlns` attribute. The prefix is then used to qualify the local names of elements and attributes from that language. E.g. `<p:a xmlns:p='http://www.w3.org/1999/xhtml'>` designates an XHTML `<a>` element.

[2] 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? Even though `<a:foo>` and `<b:foo>` are semantically equivalent, XML matches opening and closing tags lexically (Bray et al., 2006b). Thus "`<a:foo></b:foo>`" is not well-formed regardless of how a and b are bound. All XHTML compliant browsers we have tested exhibited this behavior. This implies that Noncespaces needs to randomize only namespace prefixes, but not the URIs to which the prefixes are associated.

randomization to ensure integrity of trust class information, a policy that places appropriate constraints on rich content provided by untrusted users is still necessary for our solution to be useful in practice. Therefore, we provide a mechanism for the server to specify a policy for the client to enforce when rendering the document.

Noncespaces adds three HTTP protocol headers to each HTTP response:

- `X-Noncespaces-Version`: [0-9]+\.[0-9]+ communicates the version of the Noncespaces policy and semantics that should be used, in case future changes are required.
- `X-Noncespaces-Policy`: *URI* denotes the URI of the policy for the current document. If the client does not have the policy in its cache, a compliant client must first retrieve the policy and validate the document before rendering.
- `X-Noncespaces-Context`: *TrustClass* = *Rand* (*TrustClass* = *Rand*)*

  Both *TrustClass* and *Rand* reduce to *Name*. To prevent an attacker from guessing the namespace prefixes in an (X)HTML document, the server must use different randomized prefixes each time it serves the document. This header maps the random identifiers used in the (X)HTML document delivered by the response to the trust class names used in the policy. This allows clients to cache the policy because the server can provide the same policy file to all the requests for the (X)HTML document.

4.1.1. *Compatibility with caching*

At first glance, response caching may appear to pose a problem for Noncespaces. Noncespaces requires that an attacker must not be able to predict the value of prefixes in future document instances. Local caches, caching proxies, and content delivery networks (CDNs) save a single response and may deliver it multiple times. Indeed, if it were possible to inject malicious content into a response after learning the prefix values chosen when the document was rendered, an attacker could defeat our encoding mechanism. Also, if a cache delivers a Noncespaces-encoded document without its associated headers, clients will not be able to interpret the response correctly.

To prevent an attacker from injecting content after learning which prefixes were chosen, we ensure that HTTP/1.1 compliant caches will only respond with complete Noncespaces-encoded responses. Only content injected by an attacker at the time the document was rendered, before the attacker learns the prefix values, will be included in any single response. Even if that response is subsequently rendered multiple times, it does not provide additional opportunities for an attacker to inject content.

An HTTP/1.1 compliant cache will not combine portions of multiple responses into a single response unless it can ensure that the entity's octet representation has not changed. (Fielding et al., 1999) Choosing new random prefixes every time a document is rendered ensures that the octet representation of the entity will differ, with high probability, in every response. This ensures that a cache will not provide additional opportunities to inject content by combining multiple responses.

HTTP/1.1 also requires caches to store all end-to-end headers with each cache entry and to include them in any response formed from that cache entry. This ensures that clients will receive the headers necessary to interpret each Noncespaces-encoded response whenever such a response is served by a cache.

4.2. *Policy specification*

A Noncespaces policy specifies the browser capabilities that content in a given trust class can invoke. A grammar for our policy language is given in Fig. 5. We designed the policy language to be similar to a firewall configuration language. Comments begin with an # character and extend to the end of the line. A minimal policy consists of a sequence of `allow/deny` rules. Each rule applies a policy decision—allow or deny— to a set of document nodes matched using an XPath expression. We have employed the XPath language because it was specifically designed for querying content from hierarchical documents. This allows constraints to be placed on elements, attributes, values, and position of nodes in the document hierarchy. Noncespaces provides basic functions for string normalization and additional boolean functions for matching based on trust class or whether an attribute value has changed from the default specified by the language (Fig. 6). For example, the XPath expression `//a` can be used to match all anchor elements descending from the document's root node (`//`). `//@href` will match all `href` attributes in the document. `namespace` declarations bind a namespace prefix to an XML namespace URI for use in XPath expressions.

The `trustclass` and `order` commands are used to define the hierarchy of trust classes. The optional `trustclass` command declares a trust class. A sequence of `order` commands encode the partial ordering between trust classes. This allows policy authors to specify any lattice relation over trust classes.

When checking that a document conforms to a policy, the client considers each rule in order and matches the XPath expression 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 when evaluating 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 provide a fail-safe default, if any nodes remain unmatched after evaluating all rules, we consider those nodes to be policy violations (i.e. all

Policy	::=	((*Namespace* | *TrustClass* | *Order* | *Rule* | *Comment*) \n)*
Namespace	::=	`namespace` *Name* "? *URI* "? *Comment*?
TrustClass	::=	`trustclass` *Name Comment*?
Order	::=	`order` *Relations Comment*?
Relations	::=	*Name* < *Name*
	|	*Name* < *Name* , *Relations*
Rule	::=	*Decision XPathExpression*
Decision	::=	`allow` | `deny`
Comment	::=	# .*
Name	::=	[a-zA-Z0-9_.-]+
URI	::=	As per RFC 2396

Fig. 5 – Grammar for Noncespaces Client-side policy language.

boolean ns:trust-class(node, constraint) where constraint is a string specifying a comparison operator (e.g. =, !=, >, <, >=, <=) and the name of a trust class. If the trust class of node n is tc1, ns:trust-class(n, '>=tc2') returns the value of tc1 >= tc2.

boolean ns:isspecified(attr) Returns true if attr was explicitly specified in the document (as opposed to a default attribute supplied by a parser).

string ns:tolower(string) Return argument converted to uppercase.

string ns:toupper(string) Return argument converted to lowercase.

Fig. 6 – XPath functions provided by Noncespaces.

policies end with an implicit `deny //*|//@*`). In the event that a policy author wishes to override the default behavior in order to specify a blacklist policy, he can specify `allow //*|//@*` as the last rule to allow all as of yet unmatched nodes. Fig. 7 gives the algorithm for checking a policy.

Example policies are provided in Figs. 8 and 9. The policy in Fig. 8 is a policy for XHTML documents that specifies two trust classes, trusted and untrusted. There are no restrictions on which tags and attributes can appear in trusted content. Only tags and attributes that correspond to BBCode are allowed in untrusted content: stylistic markup, links to other HTTP resources, and images. Note that lines 17–18 only permit link and image tags to specify URLs for the (non-script) HTTP protocol.

Fig. 9 demonstrates the language's flexibility for more fine-grained policies. Lines 1–2 declare that the namespace prefixes x and m used in the following patterns refer to XHTML and MathML content, respectively. Lines 4–7 declare multiple trust classes and line 8 defines the ordering between them. The policy provides four trust classes. In order of decreasing trust they are: (1) static: static web application content, (2) developer: dynamic content written by developers, (3) auth: dynamic content written by authenticated users, and (4) unauth: dynamic content written by unauthenticated users. In this web application, the capabilities of each trust class are a superset of the capabilities of all less-trusted classes. Line 11 allows static elements and attributes from any XML namespace to appear in the document.

In this policy, we make the simplifying assumption that all default attributes added by the XML parser are safe. Therefore, line 12 allows all default attributes from any XML namespace. Lines 15–17 allow dynamic XHTML `<script>` tags and href attributes to be created by developers. Lines 20–23 allow trust classes greater than or equal to `auth` to specify links and images that reference an absolute HTTP URL. Finally, lines 26–27 allow all trust classes to specify various text presentation markup and MathML content.

This policy illustrates several strengths of our policy language. By using `ns:isspecified()` on line 12 to allow all default attributes, we avoid having to explicitly allow every default attribute that the XML parser might add to each element. Each of the rules beginning from line 20 illustrates how the ability to make comparisons between trust classes saves the policy writer from having to explicitly enumerate every trust class that is permitted to specify a particular element or attribute. Line 27 allows any trust class to specify any MathML element by qualifying a wildcard with the MathML namespace prefix (m). Also, because trust classes form a lattice, there is always a unique lowest trust class. In the event that a server-side bug causes some content to escape classification, the browser can automatically place that content into the lowest trust class. (Each of these items represents an improvement over the policy language originally presented in (Van Gundy and Chen, 2009).)

Though our examples employ XML, XPath can be used with HTML documents as well. Even though HTML documents

```
Input  : A document d and a policy p.
Output: TRUE if the document d satisfies the policy p; FALSE otherwise.
1  begin
2      for Element or attribute node n ∈ d do
3          n.checked = FALSE
4      for Rule r ∈ p.rules do
5          for Node n ∈ d.matchNodes(r.XPathPattern) do
6              if n.checked == FALSE then
7                  if r.action == ALLOW then
8                      n.checked = TRUE
9                  else
10                     return FALSE
11     for Element or attribute node n ∈ d do
12         if n.checked == FALSE then
13             return FALSE;
14     return TRUE;
15 end
```

Fig. 7 – Document validation algorithm. This algorithm determines whether a document satisfies a Noncespaces policy.

```
1  # Restrict untrusted content to safe subset of XHTML
2  namespace  x   http://www.w3.org/1999/xhtml
3  # Declare trust classes
4  trustclass trusted
5  trustclass untrusted
6  order untrusted < trusted
7
8  # Policy for trusted content
9  allow //x:*[ns:trust-class(., "=trusted")]  # Allow all trusted elements
10 allow //@x:*[ns:trust-class(., "=trusted")] # Allow all trusted attributes
11
12 # Allow safe untrusted elements
13 allow //x:b  |  //x:i    |  //x:u  |  //x:s  |  //x:pre  |  //x:q
14 allow //x:a  |  //x:img  |  //x:blockquote
15
16 # Allow HTTP protocol in the <a href> and <img src> attributes
17 allow //x:a/@href[starts-with(., "http:")]
18 allow //x:img/@src[starts-with(., "http:")]
19
20 # Deny all remaining elements and attributes
21 deny //*  |  //@*
```

Fig. 8 – Example Noncespaces policy. This policy restricts untrusted content to BBCode.

```
namespace  x  http://www.w3.org/1999/xhtml
namespace  m  http://www.w3.org/1998/Math/MathML

trustclass static             # static code
trustclass developer          # dynamic code written by developers
trustclass auth               # dynamic code by authenticated users
trustclass unauth             # dynamic code by unauthenticated users
order unauth < auth, auth < developer, developer < static

# Allow all static code from any namespace and default attributes
allow //*[ns:trust-class(., '=static')] | //@*[ns:trust-class(., '=static')]
allow //@*[not(ns:isspecified(.))]

# Allow developer authored scripts and javascript: links
allow //x:script[ns:trust-class(., '>=developer')]
allow //x:script/@src[ns:trust-class(., '>=developer')]
allow //@href[ns:trust-class(., '>=developer')]

# Allow authenticated users to provide http: links and images
allow //x:a[ns:trust-class(., '>=auth')]
allow //x:a/@href[starts-with(., 'http:') and ns:trust-class(., '>=auth')]
allow //x:img[ns:trust-class(., '>=auth')]
allow //x:img/@src[starts-with(., 'http:') and ns:trust-class(., '>=auth')]

# Allow presentation and MathML markup from unauthenticated users
allow //x:b | //x:u | //x:i | //x:em | //x:strong | //x:pre
allow //m:*
```

Fig. 9 – Example multi-level Noncespaces policy. This policy illustrates multiple levels of trust, multiple XML languages, and use of custom XPath functions provided by Noncespaces.

need not be well-formed, browsers translate HTML into a Document Object Model representation that can be used to service XPath queries. We prefer this policy mechanism to more complex ones like event-based policies or dynamic information flow tracking for several reasons. Because it is not Turing complete, it is easier to reason about the effects of a policy making incorrect policy specification less likely. Enforcing policy at the syntax (Document Object Model) level also has advantages. It makes implementation less intrusive, facilitating adoption across multiple browsers and making it possible to retrofit legacy browsers without requiring internal modifications by using our proxy implementation described in Section 5.2.

4.3. *Client enforcement*

When receiving a Noncespaces-encoded response from a server, the web browser must ensure that the document is well-formed and conforms to the policy before rendering it. 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 multiple requests. We also expect it to be common for a single, seldom-changing policy to be used for each web application.

Client-side enforcement of the policy is necessary because it avoids possible semantic differences between the policy checker and the browser, which might lead the browser to interpret a document in a way that violates the policy even though the policy checker has verified the document.

5. Implementation

5.1. *Server implementation*

Noncespaces requires the server to identify untrusted content in web pages. The server may choose any approach from whitelisting trusted content statically to determining untrusted content dynamically by program analysis or information flow tracking. In our prototype implementation, we choose an approach that leverages a popular web application development paradigm to conservatively classify content with low overhead. The Model-View-Controller (Burbeck, 1992) design pattern advocates separating presentation from business logic. Many modern web applications employ a template system that inserts the dynamic values which business logic computes into static templates that decide the presentation of the web page. Since web developers author templates, content in templates can be trusted. By contrast, dynamic values may, and often do, come from untrusted sources. We consider dynamic values to be untrusted. This approach requires that JavaScript be placed in templates to be recognized as trusted content. This requirement is reasonable because most scripts can be specified statically. Scripts may then use DOM interaction to query for dynamic inputs.

Treating all dynamic values as untrusted is safe, but it might be too conservative in some situations. Consider the following template used to toggle the visibility of a dynamic menu: `<a onclick='toggle("{id}")'>`. The `toggle(id)` function accepts a string parameter indicating the HTML id of the element to operate on. Because the value of the `onclick`

attribute contains a template variable (`id`), it is treated as a dynamic (untrusted) value. If the policy denies all untrusted onclick attributes, the client will reject this document, even when the generated JavaScript code conforms to the developer's intentions. There are several straightforward solutions to this problem. Our use of a configurable policy allows a policy writer to explicitly whitelist safe, untrusted content through constraints on attribute values. For instance, the policy could allow untrusted `onclick` attributes which conform to the intended format: `toggle("[A-Za-z0-9]+")`. Another alternative is for certain untrusted content to be whitelisted within the web application after ensuring either proper sanitization or ensuring that it contains no malicious input by program analysis or information flow tracking. We would then consider an XHTML construct to be trusted if it is either static or on the whitelist.

5.1.1. *NSmarty*

To automatically annotate the content of web pages generated from templates, we modified the Smarty Template Engine, 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 variables referenced in the template. The template engine will then generate a document based on the template and the dynamic values provided.

For our prototype implementation, we apply randomization to XHTML documents. To randomize XML namespace prefixes in Smarty templates, we must be able to recognize static XHTML constructs in the template. Since the Smarty language allows Smarty tags to appear anywhere in a template, e.g. in element and attribute names, we must restrict the Smarty language to be able to determine all static XHTML elements and attributes. 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 XHTML 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. On each request, the cached PHP code will run to render the output document. We provide a preprocessor that is invoked by Smarty before it compiles each template. Our preprocessor inserts PHP code that replaces static XML namespace prefixes with randomly generated prefixes each time the document is rendered. The process is depicted in Fig. 10.

To preserve the semantics of the generated document, we map each static prefix to a random prefix bound to the same namespace URI as the static prefix (note that different prefixes may be bound to the same URI or the same prefix may be bound to different URIs at different points in the document). However, since the Smarty (and also our NSmarty) language is Turing-complete, it is infeasible to determine the scope of each static prefix at compile time. This implies that it is also infeasible to determine the URI that each static prefix represents. Therefore, we map each unique static prefix to a unique random prefix. If the original document without prefix randomization is well-formed and all XHTML appearing in dynamic content is well-formed, the new document with prefix randomization will also be well-formed and will be semantically equivalent to the original document. If dynamic content contains non-well-formed XHTML, our prefix

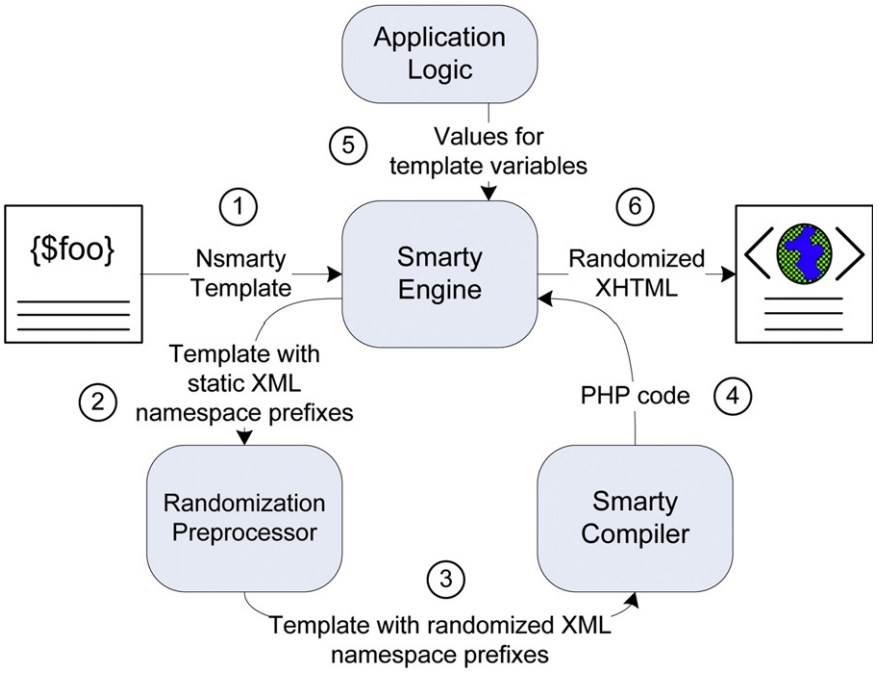

Fig. 10 – Implementing Noncespaces within Smarty. This figure illustrates the operation of our modified smarty template engine.

randomization algorithm may create non-well-formed documents. We prevent this from occurring by verifying that each document remains well-formed after prefix randomization.

Figs. 1 and 4 show an original XHTML template and the rendered document after prefix randomization, respectively. Fig. 11 shows the pseudocode for prefix randomization.

Because NSmarty takes advantages of XML namespaces, the server should serve Noncespaces documents with the `application/xhtml+xml` content type. Serving the document as XML provides other benefits, discussed below.

5.2. Client implementation

The client validates each document against its policy before rendering to ensure safety. We implemented our policy validator as a client-side proxy that mediates communication between the browser and the server. Our proxy forwards requests from the web browser to the appropriate server. When it receives a Noncespaces-encoded response from the server, the proxy attempts to validate the document against the specified policy. If the document conforms to the policy, the proxy forwards it to the client. If the document violates the policy, 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.

We chose a proxy implementation to provide a rapid proof of concept and incremental deployability for any browser that supports the use of an HTTP proxy. Also, until adoption of Noncespaces is sufficiently widespread, a Noncespaces enabled server can protect incompatible clients by employing our proxy in a reverse proxy configuration. Using Noncespaces in this configuration is reminiscent of SWAP (Wurzinger et al., 2009) and is subject to shortcomings we outline in Section 3.

Performing policy validation in a proxy, instead of within the browser, has several disadvantages. Using a proxy increases the response latency experienced at the browser. Also, because the policy validator does not have access to the browser's DOM, parsing differences between the validator and the browser may provide opportunities for attack. Our NSmarty implementation targets XHTML served as `application/xhtml+xml` to help mitigate this problem. The stricter parsing requirements of XML means that the proxy is less susceptible to parsing ambiguities than would be the case with HTML. We do not view requiring XHTML 1.0 compliance as a shortcoming of our prototype. Most modern browsers (with the notable exception of Microsoft Internet Explorer) are XHTML compliant. The restrictions imposed by XHTML are not onerous; they merely require documents to follow a simple, well-defined format. However, the prefix randomization technique that we have presented has one subtle incompatibility with XHTML that is easy to work around. While some browsers (such as Opera (Opera Browser) understand XHTML attributes that have been qualified with a prefix bound to the XHTML namespace, XHTML Modularization 1.1 (Austin et al., 2008) specifies that most XHTML attributes should not be qualified. For browsers that do not support qualified attributes, we can use a client-side JavaScript stub to unqualify attributes randomized by Noncespaces after the document has been validated.

5.3. Policy training mode

In order to protect any web application effectively, Noncespaces requires a security policy. To create a whitelist policy manually, a policy developer must enumerate all outputs that should be permitted by the policy. This can be difficult for web applications with a significant amount of dynamic content. The developer can either attempt to statically infer all possible outputs from the application source or she can run the application in order to observe possible outputs. Achieving completeness is a challenge with either approach. The Turing-completeness of common template languages can make it impossible to statically determine all possible outputs. Likewise, output observed from running an application will not reflect any application features not exercised by the developer.

Even given a complete view of application output, the developer must write rules which accurately capture permitted outputs and then test the web application with the resulting policy to ensure that the policy is general enough to allow full use of the web application. Whenever a policy violation is encountered the developer must either modify the policy to allow the offending document structure or modify the web application to conform to the existing policy. Performing these actions by hand can be very tedious and time consuming.

To facilitate rapid policy development, we have implemented a training mode for our client-side proxy that helps the developer create a whitelist policy for their application by automating many of these steps. The developer provides a seed policy, an incomplete policy that they would like to serve as a basis for the policy generated by our training system. The developer then exercises web application functionality in a trusted environment to provide a reasonably complete view of the web application output. Whenever the proxy encounters a document node that is not permitted under the current policy, it generates a rule to allow that node and adds it to the current policy. In this way, when the developer finishes exercising the full-range of application functionality, the proxy can return a policy allowing all of the content encountered. This combines, into a single activity, the separate steps of determining possible application outputs, writing rules to permit them, testing the policy, writing rules

```
Input  : An XML document d
Output: The document d after prefix randomization
1  begin
2      for Tag t ∈ d do
3          for Attribute a ∈ t do
4              if a is a namespace declaration then
5                  if map[a.prefix] is not defined then
6                      map[a.prefix] = random()
7                  a.prefix = map[a.prefix]
8              else if a.value is static (i.e. containing no template tag) then
9                  a.prefix = map[a.prefix]
10
11         t.prefix = map[t.prefix]
12 end
```

Fig. 11 – XML Namespace prefix randomization algorithm. This algorithm prepends randomly chosen prefixes to all static XML elements and attributes within a document.

to remedy incompleteness, and subsequent re-testing of the policy.

Beginning training from a seed policy allows a policy to be incrementally updated after changes are made to the application by supplying the current policy as the seed policy and re-running the application test suite. An empty seed policy corresponds to a deny by default policy and rules will be generated to allow every document node encountered.

Exercising the application can be automated by running existing functionality and quality assurance tests in an integration testing environment. If automated tests aren't available, the developer can manually interact with the web application. Common test automation tools (Sahi; Selenium IDE) can be used to create an automated test suite from a one-time manual interaction.

When the proxy encounters content not permitted by the policy, it must derive policy rules to allow the content. This is safe because training occurs in a trusted environment. Therefore, the output will not contain any malicious content. When creating a rule to remedy a policy violation, our system must make a tradeoff between being too restrictive and too permissive. If our system attempted to find a maximally-restrictive set of rules allowing only observed behavior, the learned policy would prohibit legitimate outputs not seen during training. Instead our training mode generates simple rules allowing the node in question to appear anywhere in the document. The generated rules must be reviewed by the developer to ensure that they conform to the intended security policy. We believe that this is an acceptable tradeoff—guidance from the developer is already necessary because our system cannot know the developer's intended security policy. In practice, we find that the training mode allows us to quickly derive a policy that permits large classes of innocuous XHTML content while highlighting security-sensitive content that was not present in the stub policy.

5.4. *Deployment*

It is easy to retrofit existing web applications with Noncespaces. The developer writes an appropriate policy and, when necessary, revises the Smarty templates such that they are also valid NSmarty templates.

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, no further modification is necessary. Noncespaces will randomize all the static namespace prefixes. Because no namespace prefixes in the dynamic content will be randomized, they cannot invoke any capabilities reserved for trusted content.

6. 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 a wide variety of XSS attacks. Our performance evaluation measures the costs of Noncespaces from both the client's and server's points of view.

6.1. *Security*

6.1.1. *TikiWiki case study*

We tested Noncespaces against six XSS exploits targeting two vulnerable applications. The exploits were crafted to exhibit the various forms that an XSS attack may take (Van Gundy and Chen, 2009). The applications used in this evaluation were a version of TikiWiki (TikiWiki CMS/Groupware) with a number of XSS vulnerabilities and Trustify, 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 Fig. 8). We also had to add exceptions for trusted content that TikiWiki generates dynamically by design, such as names and values of form elements, certain JavaScript links implementing collapsible menus, and custom style sheets based on user preferences.

For Trustify, 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 (Jim et al., 2007). This policy denies common script-invoking tags and attributes from any namespace (e.g., `<script>` and `onclick`) that are descendants of a `<div>` tag with the `class = "sandbox"` attribute. (Note: the policy does not attempt to be exhaustive. It does not enumerate non-standard browser-specific tags and attributes.) To allow the rules to apply to elements and attributes in any namespace we use the common XPath idiom of matching by each node's `local-name()`. The 22 line policy is given in Fig. 12.

For each of the exploits we first verified that each exploit succeeded without Noncespaces enabled. We then enabled Noncespaces and verified that all exploits were blocked as policy violations.

6.1.2. *LifeType case study*

To gain more insight into the work involved in porting existing applications to Noncespaces, we ported LifeType, a popular blog application, to work with Noncespaces. LifeType is a mature, full-featured blog application consisting of 155 K lines of PHP and XHTML code. Enabling Noncespaces required changes to only 180 lines of code. The majority of code changes occurred in LifeType's HTTP header handling. These changes were necessary because Noncespaces needs to include its own headers before any content is sent to the client.

We developed a static-dynamic policy for LifeType that attempts to restrict untrusted content to a minimal set of capabilities. Using our proxy's training mode, it took approximately 4 hrs to exercise a significant portion of LifeType's functionality and to manually refine generated rules that were overly general. We then went through our functionality exercise again to ensure that we did not prohibit any legitimate behavior.

```
trustclass unclassified

# Blacklist (possibly incomplete)
deny //*[local-name() = 'div' and @*[local-name() = 'class' and . = 'sandbox']]\
     //*[local-name() = 'script']
deny //*[local-name() = 'div' and @*[local-name() = 'class' and . = 'sandbox']]\
     //@*[   local-name() = 'onload'      or local-name() = 'onunload' \
          or local-name() = 'onclick'     or local-name() = 'ondblclick' \
          or local-name() = 'onmousedown' or local-name() = 'onmouseup' \
          or local-name() = 'onmouseover' or local-name() = 'onmousemove' \
          or local-name() = 'onmouseout'  or local-name() = 'onfocus' \
          or local-name() = 'onblur'      or local-name() = 'onkeypress' \
          or local-name() = 'onkeydown'   or local-name() = 'onkeyup' \
          or local-name() = 'onsubmit'    or local-name() = 'onreset' \
          or local-name() = 'onselect'    or local-name() = 'onchange' \
          or (local-name() = 'href' \
              and starts-with(ns:tolower(normalize-space(.)), "javascript:")) \
          or (local-name() = 'src' \
              and starts-with(ns:tolower(normalize-space(.)), "javascript:"))]

# Allow everything else
allow //*
allow //@*
allow //namespace::*
```

Fig. 12 – Example sandbox policy. This ancestry-based sandbox policy prohibits potential script-invoking tags and attributes that are descendants of a `<div>` node with the `class = "sandbox"` attribute.

To test the effectiveness of our LifeType policy, we introduced XSS vulnerabilities into the application. We used the XSS Cheat Sheet (RSnake, 2008) to craft 100 XSS exploits. We then tested each exploit in Opera 9.27[3] Before applying Nonce-spaces, 50 of the exploits were successful. The remaining 50 exploits were unsuccessful against Opera because they exploit functionality unique to some other browser (such as executing JavaScript by invoking the `mocha:` protocol scheme present in older Netscape versions). After we applied Nonce-spaces, Noncespaces blocked 98 of the 100 exploits as either policy violations or XHTML parsing errors. These results give us confidence in our policy's ability to recognize exploits while allowing intended behavior and in Noncespaces's ability to block exploits that target multiple browsers. Since Nonce-spaces processes exploitable web pages before the browser renders them, many exploits that would have been incompatible with the browser were blocked by Noncespaces before they reached the browser. Neither of the two exploits that were not blocked resulted in a successful XSS attack: one was rendered as text, the other as a comment. That neither exploit caused a policy violation does not indicate a limitation of our approach. Our browser-agnostic prototype proxy implementation targets XHTML compliant browsers, as discussed previously. Neither exploit was valid XHTML.

The latter exploit is an Internet Explorer conditional comment (Microsoft Developer Network (MSDN), 2007). XHTML compliant browsers will render the comment as a comment and ignore its contents. However, Internet Explorer interprets the comment as HTML code if the specified conditions are met. This exploit illustrates how non-standards-compliant behavior can lead to security vulnerabilities and confirms our preference for eventual in-browser implementation. Only a Noncespaces-aware browser can ensure complete mediation—that all content interpreted as HTML code is checked for conformance to the policy. Table 1 summarizes the results.

[3] We used Opera for our evaluation due to its native support for namespace qualified attributes.

6.2. *Performance*

Our performance evaluation seeks to measure the overhead of Noncespaces in terms of response latency and server throughput. Our test infrastructure consisted of the applications that we used for our security evaluation running in a VMware virtual machine with 512 MB RAM running Fedora Core 3, Apache 2.0.52, and mod_php 5.2.6. The virtual machine ran on an Intel Pentium 4 3.2 GHz machine with 1 GB RAM running Ubuntu 7.10. Our client machine was an Intel Pentium 4 2 GHz machine with 256 MB RAM running Ubuntu 8.10 Server. These results represent an upper bound on performance penalty as we have spent no effort optimizing our Noncespaces prototype. In each test we used ab (ab - Apache HTTP server benchmarking tool) to retrieve an application page 1000 times. We varied the number of concurrent requests between 1, 5, 10, and 15, and the configuration of the client and server between the following:

- Baseline: measures original web application performance before applying Noncespaces.
- Randomization Only: measures impact of Noncespaces randomization on server without policy validation on client-side.
- Full Enforcement: measures the end-to-end impact of Noncespaces.

Table 1 – Security analysis results. This table summarizes the results of our security analysis of Noncespaces-enabled LifeType using a policy developed with our new training mode.

	Total exploits	Failed exploits		Successful exploits
		Blocked by Noncespaces	Incompatible with Browser	
Without Noncespaces	100	–	50	50
With Noncespaces	100	98	2	0

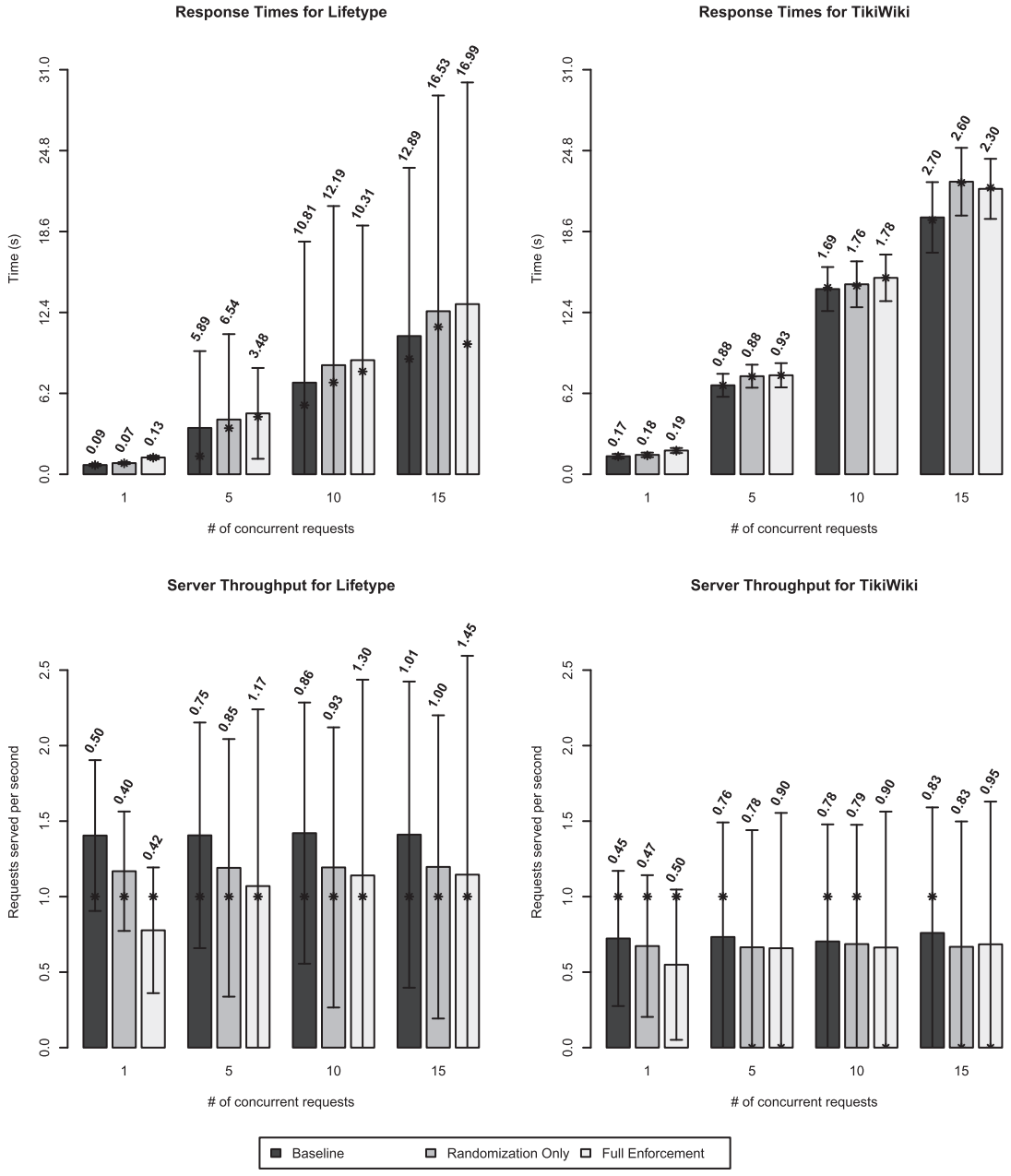

Fig. 13 – Noncespaces performance results. Performance results for Noncespaces-enabled TikiWiki and LifeType applications. Each graph groups response (or throughput) times by the number of concurrent requests. Within each group, the bars correspond to the following configurations from left to right: Baseline (Noncespaces disabled), randomization only (no policy validation), and full enforcement (randomization and policy checking). The bars depict the mean value, asterisks the median, and the vertical line segments show the mean plus or minus the standard deviation. The value of the standard deviation is shown by label above each line segment.

We ran three trials with each test configuration against both the TikiWiki and Lifetype applications.[4] We report the mean, median, and standard deviation of results over all trials. The server virtual machine was 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. Our results are shown in Fig. 13.

The graphs of response latency show that enabling Noncespaces randomization on the server increased response time by (at most) 14% for TikiWiki and 20% for LifeType. Enabling the policy checking proxy resulted in response times that were (at most) 32% higher than the baseline response time for TikiWiki and 80% higher for LifeType. Though the overhead may appear significant at first glance, during interactive use latency typically increased by no more than 0.6 s.

We also examine the effect of Noncespaces on server throughput. With randomization enabled throughput is reduced by about 10% for TikiWiki and 20% for LifeType. After enabling policy checking, the throughput of both TikiWiki and LifeType decreases by an additional 3% for higher numbers of concurrent requests. Because policy checking is performed on the client side the effect of policy checking on server throughput is minimized when multiple clients make requests simultaneously.

7. 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 safely allowing rich user-contributed content. The core technique of Noncespaces uses randomized (X)HTML tags to identify and annotate untrusted content, similar to the use of Instruction Set Randomization to defeat injected binary code attacks. Noncespaces is simple. The server need not sanitize any untrusted content. This 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. Noncespaces-aware clients can reliably prevent all the attacks that the policy prohibits, and even Noncespaces-unaware clients can prevent node-splitting attacks. We implemented a prototype of Noncespaces for a web application template system and on a proxy at the client side. Experiments show that the overhead of our prototype Noncespaces implementation is moderate.

Funding

This research is partially supported by the National Science Foundation through grants CNS 0644450 and 1018964, and by an AFOSR MURI award. Neither source played a direct role in the design or implementation of our system, authoring this paper, nor the decision to submit it for publication.

[4] We do not report performance results for Trustify. We developed Trustify for our security evaluation to exhibit all forms of XSS vulnerability vectors. It is not representative of realistic web application workloads.

Conflict of interest

Both authors maintain that, to the best of their knowledge, they are free of potential conflicts of interest due to employment, funding, financial interests, or other factors.

Acknowledgments

We would like to thank: Francis Hsu for his assistance with the figures in this paper and valuable help proofreading, Zhendong Su and his research group for critical input during the early stages of this work, and the anonymous reviewers for their helpful comments.

REFERENCES

ab - Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html; 2010. [accessed 20.03.11].

Austin D, Peruvemba S, McCarron S, Ishikawa M, Birbeck M. XHTML modularization 1.1 [accessed 20.03.11]. W3C; 2008. Technical Report.

Barrantes EG, Ackley DH, Forrest S, Palmer TS, Stefanović D, Zovi DD. Randomized instruction set emulation to disrupt binary code injection attacks. In: Proceedings of the 10th ACM conference on computer and communications security (CCS 2003). Washington D.C., USA: ACM; 2003. p. 281–9.

Boyd SW, Keromytis AD. SQLrand: preventing SQL injection attacks. In: Proceedings of the 2nd applied cryptography and network security (ACNS) conference; 2004. p. 292–302.

Bray T, Hollander D, Layman A, Tobin R. Namespaces in XML 1.0 [accessed 20.03.11]. 2nd ed. W3C; 2006a. Technical Report.

Bray T, Paoli J, Sperberg-McQueen CM, Maler E, Yergeau F. Extensible markup language (XML) 1.0 [accessed 20.03.11]. 4th ed. W3C; 2006b. Technical Report.

Burbeck S. How to use model-view-controller (MVC) [accessed 20.03.11], http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html; 1992.

CERT Coordination Center. CERT advisory CA-2000-02 malicious HTML tags embedded in client web requests [accessed 20.03.11], http://www.cert.org/advisories/CA-2000-02.html; 2000.

Chen S, Xu J, Sezer EC, Gauriar P, Iyer RK. Non-control-data attacks are realistic Threats. In: USENIX security symposium. USENIX the advanced computing systems association. Baltimore, MD, USA: USENIX Association; 2005. p. 177–92.

CWE/SANS. Top 25 most dangerous software errors [accessed 20.03.11], http://cwe.mitre.org/top25/; 2010.

Eich B. JavaScript: mobility & ubiquity. In: Barthe G, Mantel Y, Müller P, Myers AC, Sabelfeld A, editors. Mobility, ubiquity and security. Dagstuhl, Germany: Internationales Begegnungs-und Forschungszentrum für Informatik (IBFI); number 07091 in Dagstuhl seminar proceedings; 2007.

Erlingsson Ú, Livshits B, Xie Y. End-to-end web application security. In: Proceedings of the 11th USENIX workshop on hot topics in operating systems. USENIX the advanced computing

systems association. San Diego, CA: USENIX Association; 2007. p. 1–6.
Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P, et al. Hypertext Transfer Protocol – HTTP/1.1; 1999.
Genshi. Python toolkit for generation of output for the web [accessed 20.03.11], http://genshi.edgewall.org/; 2008.
Jim T, Swamy N, Hicks M. Defeating scripting attacks with browser-enforced embedded policies. In: Proceedings of the international World Wide Web conference (WWW). Banff, Alberta, Canada: ACM; 2007. p. 601–10.
Kc GS, Keromytis AD, Prevelakis V. Countering code-injection attacks with instruction-set randomization. In: CCS '03: Proceedings of the 10th ACM conference on computer and communications security. Washington D.C., USA: ACM; 2003. p. 272–80.
Kirda E, Kruegel C, Vigna G, Jovanovic N. Noxes: a client-side solution for mitigating cross site scripting attacks. In: Proceedings of the ACM symposium on applied computing (SAC); 2006. p. 330–7. Dijon, France.
Kirkegaard C, Mřller A. Static analysis for Java Servlets and JSP. In: Proceedings of the 13th international static analysis Symposium, SAS '06. LNCS, vol. 4134. Springer-Verlag; 2006. p. 336–52. Full version available as BRICS RS-06-10.
Livshits VB, Lam MS. Finding security vulnerabilities in Java applications with static analysis. In: USENIX security symposium. USENIX the advanced computing systems association. Baltimore, MD, USA: USENIX Association; 2005. p. 271–86.
Markham G. Script Keys. Last accessed: Mar 20, 2011, http://www.gerv.net/security/script-keys/; 2005.
Markham G. Content restrictions. Last accessed: Mar 20, 2011, http://www.gerv.net/security/content-restrictions/; 2007.
Meyerovich LA, Livshits B. ConScript: specifying and enforcing fine-grained security policies for JavaScript in the browser. In: IEEE Symposium on security and privacy. Berkeley, CA, USA: IEEE Computer Society; 2010. p. 481–96.
Microsoft Developer Network (MSDN). About conditional comments. Last accessed: Mar 20, 2011, http://msdn.microsoft.com/en-us/library/ms537512.aspx; 2007.
MITRE Corporation. Vulnerability type distributions in CVE [accessed 20.03.11], http://cwe.mitre.org/documents/vuln-trends/index.html; 2007.
San Diego, CA Nadji Y, Saxena P, Song D. Document structure integrity: a robust basis for cross-site scripting Defense. In: Proceedings of the 16th annual network and distributed system security symposium (NDSS); 2009. p. 17–36.
Nava EV, Lindsay D. Abusing Internet Explorer 8's XSS filters [accessed 20.03.11], http://p42.us/ie8xss/Abusing_IE8s_XSS_Filters.pdf; 2010.
Nguyen-Tuong A, Guarnieri S, Greene D, Shirley J, Evans D. Automatically hardening web applications using precise tainting. In: Proceedings of the 20th IFIP international information security conference (SEC 2005); 2005. p. 372–82. Chiba, Japan.
Opera Browser. http://www.opera.com/browser/; 2008. [accessed 20.03.11].
Robertson W, Vigna G. Static enforcement of web application integrity through strong typing. In: USENIX security Symposium. USENIX the advanced computing systems Association. Montreal, Canada: USENIX Association; 2009. p. 283–98.
Ross D. IE8 security part IV: the XSS filter [accessed 20.03.11], http://blogs.msdn.com/b/ie/archive/2008/07/02/ie8-security-part-iv-the-xss-filter.aspx; 2008.
RSnake. XSS (cross site scripting) cheat sheet [accessed 20.03.11], http://ha.ckers.org/xss.html; 2008.
Sahi http://sahi.co.in/; 2011. [accessed 20.03.11].
Samy. Technical explanation of the MySpace worm [accessed 20.03.11], http://web.archive.org/web/20060208182348/namb.la/popular/tech.html; 2006.
Selenium IDE. http://seleniumhq.org/projects/ide/; 2011. [accessed 20.03.11].
Shezaf O. The universal XSS PDF vulnerability [accessed 20.03.11], http://www.owasp.org/images/4/4b/OWASP_IL_The_Universal_XSS_PDF_Vulnerability; 2007.
Smarty Template Engine. http://www.smarty.net/; 2008. [accessed 20.03.11].
Stamm S, Sterne B, Markham G. Reining in the web with content security policy. In: Proceedings of the 19th international World Wide Web conference (WWW). Raleigh, North Carolina, USA: ACM; 2010. p. 921–30.
Su Z, Wassermann G. The Essence of command injection attacks in web applications. In: Proceedings of the 33rd ACM SIGPLAN-SIGACT symposium on principles of programming languages. Charleston, South Carolina, USA: ACM; 2006. p. 372–82.
Tang S, Grier C, Aciicmez O, King ST. Alhambra: a system for creating, enforcing, and testing browser security policies. In: Proceedings of the 19th International World Wide Web conference (WWW). Raleigh, North Carolina, USA: ACM; 2010. p. 941–50.
Ter Louw M, Venkatakrishnan VN. Blueprint: robust prevention of cross-site scripting attacks for existing browsers. In: IEEE symposium on security and privacy. Berkeley, CA, USA: IEEE Computer Society; 2009. p. 331–46.
The Open Web Application Security Project. Cross-site scripting (XSS) [accessed 20.03.11], http://www.owasp.org/index.php/Cross-site_Scripting_%2528xSS%2529; 2010.
TikiWiki CMS/groupware. http://info.tikiwiki.org/tiki-index.php; 2010. [accessed 20.03.11].
San Diego, CA Van Gundy M, Chen H. Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks. In: Proceedings of the 16th Annual network and distributed system security symposium (NDSS); 2009. p. 55–67.
Venema W. Taint support for PHP [accessed 20.03.11], http://wiki.php.net/rfc/taint; 2008.
Vogt P, Nentwich F, Jovanovic N, Kirda E, Kruegel C. Vigna G cross-site scripting prevention with dynamic data tainting and static analysis. In: Proceedings of the network and distributed system security symposium (NDSS); 2007. San Diego, CA.
Wassermann G, Su Z. Sound and precise analysis of web applications for injection vulnerabilities. In: Proceedings of the ACM SIGPLAN 2007 conference on programming language design and implementation. San Diego, CA: ACM Press; 2007. p. 32–41. New York, NY, USA.
Wassermann G, Su Z. Static detection of cross-site scripting vulnerabilities. In: Proceedings of the 30th international conference on software engineering. Leipzig, Germany: ACM; 2008. p. 171–80.
Wurzinger P, Platzer C, Ludl C, Kirda E, Kruegel C. SWAP: mitigating XSS attacks using a reverse proxy. In: Proceedings of the 2009 ICSE workshop on software engineering for secure systems. Washington, DC, USA: IEEE Computer Society; 2009. p. 33–9.
Xie Y, Aiken A. Static Detection of security vulnerabilities in scripting languages. In: USENIX security symposium. USENIX the advanced computing systems association. Vancouver, B.C., Canada: USENIX Association; 2006. p. 179–92.
Xu W, Bhatkar S, Sekar R. Taint-Enhanced policy enforcement: a Practical approach to defeat a wide range of attacks. In: USENIX security symposium. USENIX the advanced computing systems Association. Vancouver, B.C., Canada: USENIX Association; 2006. p. 121–36.

Matthew Van Gundy is a Ph.D. candidate in Computer Science at the University of California, Davis. He received his B.S. in Computer Engineering from the University of California, Santa Barbara in 2005. His research focus is computer security including privacy preserving technologies, censorship resistance, information flow security, and web security.

Hao Chen is an associate professor of Computer Science at the University of California, Davis. He received his Ph.D. in Computer Science from the University of California, Berkeley in 2004. His primary research interest is computer security, particularly web security, wireless security, and privacy. He won the NSF CAREER award in 2007.