Appeared in 30th IEEE Symposium on Security and Privacy, Oakland, May 2009.
Acceptance : 26 out of 254 papers, 10.2 %.

BLUEPRINT: Robust Prevention of Cross-site Scripting Attacks for Existing Browsers

Mike Ter Louw
mter@cs.uic.edu

V.N. Venkatakrishnan
venkat@cs.uic.edu

Department of Computer Science
University of Illinois at Chicago

Abstract

As social networking sites proliferate across the World Wide Web, complex user-created HTML content is rapidly becoming the norm rather than the exception. User-created web content is a notorious vector for cross-site scripting (XSS) attacks that target websites and confidential user data. In this threat climate, mechanisms that render web applications immune to XSS attacks have been of recent research interest.

A challenge for these security mechanisms is enabling web applications to accept complex HTML input from users, while disallowing malicious script content. This challenge is made difficult by anomalous web browser behaviors, which are often used as vectors for successful XSS attacks.

Motivated by this problem, we present a new XSS defense strategy designed to be effective in widely deployed existing web browsers, despite anomalous browser behavior. Our approach seeks to minimize trust placed on browsers for interpreting untrusted content. We implemented this approach in a tool called BLUEPRINT *that was integrated with several popular web applications. We evaluated* BLUEPRINT *against a barrage of stress tests that demonstrate strong resistance to attacks, excellent compatibility with web browsers and reasonable performance overheads.*

1. Introduction

Cross-site scripting (XSS) attacks are the number-one security threat on the Internet today. These attacks breach confidentiality of sensitive data, undermine authorization schemes, defraud users, defame web sites, and more. The web site `www.xssed.com` documents recently successful XSS attacks on major blog and social networking sites. Notably Facebook, LiveJournal, MySpace and Orkut have all been hit by these attacks. XSS attacks can be self-propagating [1], and have the potential to rapidly victimize millions of people.

Broadly speaking, XSS is injection of unauthorized script code into a web page. As a web application processes input from untrusted users, it generates some low-integrity output web content which we term *untrusted HTML*. The goal of an XSS attack is to embed malicious script code in untrusted HTML, causing the script to be executed on a victim's web browser within the context of the conduit web application. We say the attack script is *unauthorized* because the application does not intend to allow scripts in untrusted HTML. Defenses for XSS aim to prevent unauthorized script execution by enforcing a *no-script* policy on untrusted HTML.

1.1. Defense approaches

To disallow script execution in untrusted web content, a web application might possibly take one of the following approaches.

Content Filtering. The application may attempt to detect and remove all scripts from untrusted HTML before sending it to the browser.

Browser Collaboration. The application may collaborate with the browser by indicating which scripts in the web page are authorized, leaving the browser to ensure the authorization policy is upheld.

Content filtering. Content filtering is otherwise known as sanitization. This defense technique uses filter functions to remove potentially malicious data or instructions from user input. Filter functions are applied after user input is read by a web application, but before the input is employed in a sensitive operation or output to the web browser.

Removal of scripts from untrusted content is a difficult problem for web applications that permit HTML markup in user input such as blog, wiki and social networking applications. These applications are expanding and proliferating rapidly [2], [3], thus the growing need for robust XSS defenses. The WordPress

blog platform is one popular application that empowers anonymous users to control the presentation of their blog comments. It does so by permitting input of structured HTML elements for text formatting (e.g., `<b>` for bold, `<i>` for italics). Content filtering based defenses for this type of application face a difficult challenge: allowing all benign HTML user input, while simultaneously blocking all potentially harmful scripts in the untrusted output.

Simply disallowing HTML syntax control characters is not a practical filtering solution for these applications because every control character that can be used to introduce attack code also has a legitimate use in some benign, non-script context. For example, the < character needs to be present in hyperlinks and text formatting, and the " character needs to be present in generic text content. Both are legitimate and allowed user inputs, but can be abused to mount XSS attacks.

Advanced content filters try to anticipate how untrusted content will be interpreted by the client web browser's parser, as it is the browser parser that makes crucial decisions about script execution. To be completely effective in eliminating XSS, a filter function must necessarily model the full range of parsing behaviors pertaining to script execution for several browsers.

This is a very difficult problem, as diligently documented in the *XSS Cheat Sheet* [4], which describes a wide variety of *parsing quirks* exhibited by different browsers. Quirks are essentially anomalous browser parser behavior that either contradict language standards or account for conditions not well defined by these standards (such as how to parse malformed HTML). They are sometimes intentionally introduced and retained in a browser's code base to correctly render existing web sites that depend on the quirks of older browsers. Quirks vary by browser, are complex to model, not entirely understood and not all known (especially for closed-source browsers). Therefore, from a web application perspective, the task of implementing *correct* and *complete* content filter functions is very difficult, if not impossible.

Browser collaboration. Robust prevention of XSS attacks can be achieved if web browsers are made capable of distinguishing authorized from unauthorized scripts. This vision was first espoused in BEEP [5], wherein this approach was implemented by (a) creating a server–browser collaboration protocol to communicate the set of authorized scripts, then (b) modifying the browser to understand this protocol and enforce a policy denying unauthorized script execution.

Although the defense strategy envisioned by the authors of BEEP is a compelling and effective long-term solution, their implementation approach leaves a large void in near-term protection. This is because web applications adopting this approach require their users to employ custom BEEP-enabled browsers for protection from XSS attacks. To scale this approach there must first be agreement on any proposed standards for server–browser collaboration, then these new standards must be incorporated in the normal browser implementation and deployment cycle for millions of installed browsers. This is a long, complicated process that can take several years.

This inherent practical limitation makes browser collaboration unsuitable for adoption in the near future by existing web applications. However, a robust, solution to XSS defense is desperately needed *now* to prevent the immediate, ongoing spate of XSS attacks.

1.2. Objectives and approach

The above discussion highlights the immediate need for a solution that:

- *robustly protects* against XSS attacks, even in the presence of browser parsing quirks,
- *supports* benign, structured HTML derived from untrusted user input, and is
- *compatible* with existing browsers currently in use by today's web users.

To clearly define compatibility, we seek a solution that works on existing, currently deployed browsers in their default configuration and settings, without any modifications, either directly to their code base or through plug-ins. We share this perspective with web application developers and service providers, who can not presume that their users will install a customized browser or download a plug-in for XSS protection.

Our contribution. Due to the prevalence of XSS attacks and current trends in web applications, there exists a strong need for preventing these attacks. We address this need by presenting the design and implementation of BLUEPRINT: an XSS defense that satisfies all three objectives mentioned above.

We observe that existing web browsers cannot be entrusted to make script identification decisions in untrusted HTML due to their unreliable parsing behavior. Therefore, in BLUEPRINT, *we enable a web application to effectively take control of parsing decisions*. By systematically reasoning about the flow of untrusted HTML in a browser, we develop an approach that provides facilities for a web application to automatically create a structural representation — a "blueprint" — of untrusted web content that is free of XSS attacks.

Our approach employs techniques to carefully transport and reproduce this blueprint in the browser exactly as intended by the web application, despite anomalous browser parsing behavior. Our general approach offers strong protection against script injections, and enables support for complex script-free HTML user input.

Extensive experiments with BLUEPRINT demonstrate its resilience against subtle XSS attacks, reasonable performance overheads, compatibility and effectiveness on over 96% of existing browser market share.

The remainder of this paper is organized as follows. Our proposed solution is introduced on a conceptual level in Section 2, followed by technical details given in Section 3. Integration with web applications is discussed in Section 4. A thorough evaluation of our proposed techniques is presented in Section 5. Related works are discussed in Section 6 and we conclude in Section 7.

2. Approach overview

The main obstacle a web application must overcome when implementing XSS defenses is the divide between its understanding of the web content represented by an HTML sequence and the understanding web browsers will have of the same. For trusted HTML or JavaScript content, this divide is not an insurmountable problem: in fact, web application developers routinely perform testing of trusted content on a variety of browsers to ensure each browser's understanding of the content is consistent with their understanding. For the remainder of this paper, we assume that web applications are fully capable of arriving at this common understanding of trusted content.

With respect to untrusted content, addressing this divide becomes very challenging. Most often, untrusted content such as user input is included dynamically in a web application's output, therefore the application's developer does not have the luxury of arriving at this common understanding beforehand through testing. As explained in the introduction, current browser parsers have many quirks that make the job of arriving at this common understanding difficult. The technical goal of our approach is to avoid the problem of understanding how a browser will parse arbitrary data. Instead we propose an approach that *enforces* the application's understanding of web content on the browser. To illustrate the ideas behind our approach, we now analyze the flow of untrusted data through the browser's HTML interpretation process.

Figure 1 presents an abstract description of how HTML input (arriving through path $\mathcal{A}$) flows through a web browser as it is parsed and interpreted. HTML code is processed by the browser's HTML lexer and parser, which produces a parse tree. During this parsing stage, executable script elements in web content are identified and corresponding script nodes are created in the parse tree. This tree is supplied as input via path $\mathcal{B}$ to the document generation stage, and HTML parsing activity is complete once we enter this stage. The document generator phase then stores and interprets

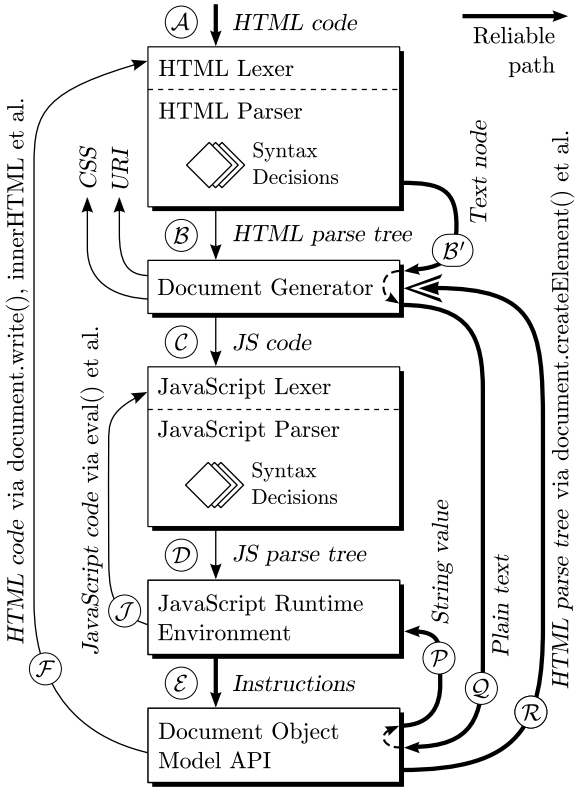


Figure 1. Generalized functional diagram of existing browsers' HTML interpretation process. Bold lines indicate the path of untrusted data in the BLUEPRINT approach, which avoids unpredictable syntax decisions.

web content described by the parse tree. For example, the document generator submits visual elements to other parts of the browser for rendering, and script elements are supplied to the JavaScript interpreter for execution via path $\mathcal{C}$.

We observe that the implicit goal of XSS prevention is to safely communicate untrusted content to the browser's document generation stage, such that the browser-generated parse tree is free of script nodes. Note that content filtering based defenses utilize the path ($\mathcal{A}$, $\mathcal{B}$) in Figure 1, and attempt to anticipate the behavior of browser parsers to ensure script nodes are not created when this path is exercised. However, after supplying untrusted HTML via path $\mathcal{A}$, *the web application has no further control* over the resulting parse tree. The web application therefore cannot ensure the resulting parse tree is free of script nodes because browser parser behavior cannot be reliably predicted due to parsing quirks.

Main idea. The crux of our approach is to eliminate any dependence on the browser's parser for building untrusted HTML parse trees. That is, we eliminate the use of path $\mathcal{B}$ and instead derive an alternative path to render untrusted content without the risk of

XSS attacks. In our approach, the following steps are performed by the web application:

1) On the application server, a parse tree is generated from untrusted HTML with precautions taken to ensure the absence of dynamic content (e.g., script) nodes in the tree.
2) On the client browser, the generated parse tree is conveyed to the browser's document generator without taking vulnerable paths such as $\mathcal{B}$ which involve unreliable browser parsing behavior.

This two-step process ensures untrusted content generated by the browser is consistent with the web application's understanding of the content. The generated document reflects the application's intention that the untrusted content does not contain scripts, therefore all unauthorized script execution is prevented.

In our approach, we create the parse tree for untrusted content programmatically using a small set of low-level *Document Object Model* (DOM) primitives that are well documented [6] and supported on all JavaScript-enabled browsers. Input is provided to these DOM APIs (via path $\mathcal{E}$) as both instructions and data: instructions define parse tree structure and node types, and data (e.g., character data in *text nodes*) annotates nodes of this tree. Once these explicit instructions are given, the browser's DOM implementation constructs the untrusted HTML parse tree, then supplies this parse tree to the document generator through the final transition $\mathcal{R}$. Our goal therefore reduces to reliably transporting both instructions and data safely to path $\mathcal{E}$ for invoking DOM APIs.

Transport of instructions. Transporting instructions is relatively easy: we simply have the web application generate *trusted* JavaScript code that flows through paths ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$). Since instructions are entirely devoid of untrusted content, we can ensure they will be correctly delivered to path $\mathcal{E}$. This claim is supported by the discussion at the start of this section about obtaining predictable behavior from trusted content through testing on various browsers.

Strictly interfacing with the DOM via trusted code is not enough to ensure the web application's intended parse tree ultimately prevails. It is still possible for trusted code to introduce script nodes into the parse tree, perhaps unintentionally if the code uses certain DOM API methods that trigger parsing activity. For instance, by invoking `document.write()` (path $\mathcal{F}$), character data may be explicitly supplied to the HTML parser, and may then violate structural integrity of the intended parse tree or even the structure of trusted HTML content. There are many similar interfaces that trigger unsafe parsing behavior, such as the `eval()` function (path $\mathcal{J}$), and the `innerHTML` property (path $\mathcal{F}$). We take care that our trusted client-side code does not use these unsafe APIs as it constructs parse trees.

Reliable transport of untrusted data. The final component of our solution is safe transport of untrusted data from path $\mathcal{A}$ to $\mathcal{E}$ for providing input to DOM APIs. If raw data is exposed to the browser's parsers, it is practically infeasible to guarantee all XSS attacks are prevented. This is because raw data might contain control character sequences uncaught by a server-side filter, and these characters can cause the formation of script nodes if the browser's parser interprets them as such. Furthermore, untrusted data must be effectively isolated from trusted code/data to preserve integrity of trusted content.

To address the threat posed by raw untrusted data, we convert this data to an encoded representation that consists only of characters from a strictly defined "safe alphabet". This alphabet (say "a–z") only contains characters that are *syntactically inert*; that is, none of the safe alphabet characters cause changes in the syntax state of the browser's HTML parser as they are processed. Therefore, for all possible sequences of safe alphabet characters the browser's parsing behavior can be reliably anticipated.

As they are safe to use for data transport without affecting document structure, we expose these encoded characters in a text node to the browser's HTML parser (shown in Figure 1 taking the alternate route $\mathcal{B}'$).

We then take steps to ensure that untrusted data completely bypasses the browser's unreliable JavaScript parser. Instructions supplied over path $\mathcal{E}$ extract the encoded text node from the DOM via paths ($\mathcal{Q}$, $\mathcal{P}$), thus avoiding the JavaScript parser altogether. We decode and recover the raw untrusted character data to the JavaScript runtime environment's memory state, then supply it to DOM APIs as input via path $\mathcal{E}$. In summary, untrusted data traverses through path ($\mathcal{A}$, $\mathcal{B}'$, $\mathcal{Q}$, $\mathcal{P}$, $\mathcal{E}$) in the figure.

We now can say the processing of raw untrusted data by the browser's DOM implementation will not result in unauthorized script execution because (as described above) we only employ DOM APIs that do not trigger parsing behavior. The parse tree ultimately generated using our approach is thus supplied via $\mathcal{R}$ to the document generator, which successfully renders the document free of any XSS attacks.

3. Implementation

We present the implementation of BLUEPRINT using a running example of a simple blog platform (modeled after WordPress) that allows untrusted, structured HTML input via article feedback *comments* from users. Figure 2(a) is an example of typical benign HTML that may occur in a comment. Figure 2(b) shows a mali-

```
1  <p>
2    Here is a page you might find
3    <b>very</b>
4    interesting:
5    <a href="http://www.cpsr.org">
6      Link</a>
7  </p><p style="text-align: right;">
8    Respectfully,
9    Alice
10 </p>
```

(a) Benign HTML blog comment

```
1  <p>
2    Here is a page you might find
3    <b """><script>doEvil(...)</script>">very</b>
4    interesting:
5    <a href=" &#14; javasc&#x0A;ript:doEvil(...);">
6      Link</a>
7  </p><p style="nop:expres/*xss*/sion(doEvil(...))">
8    Respectfully,
9    Eve
10 </p>
```

(b) Malicious HTML blog comment

Figure 2. BLUEPRINT must be permissive enough to allow (a) benign HTML content derived from untrusted user input, and still defend against (b) subtle attacks that use malformed content to trigger and exploit browser parsing quirks [4].

cious comment: one that aims to inject script content in the user's browser through several attacks. The web application source code that generates these comments is discussed in Section 4 along with steps taken to integrate our defense approach with the application.

The attack scripts in Figure 2(b) are identified in several parsing decisions made by the browser on HTML, link and CSS content (lines 3, 5 and 7, respectively). Our goals are to fully *permit* comment 2(a) to be rendered in the browser and *prevent* all attacks in comment 2(b).

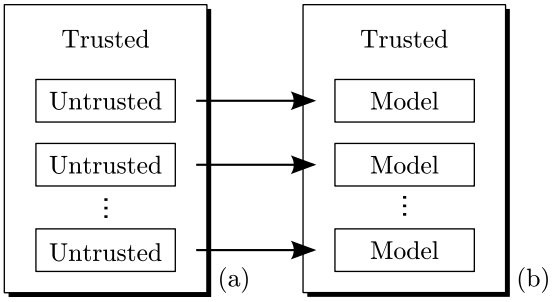
Our approach to preventing these attacks leverages on the understanding that a browser's parser can be described as the combined behavior of smaller parsers that comprise overall parser behavior. In this section, we detail our strategy to grant web applications control of generated web content by eliminating the influence of each of these smaller parsers: the HTML parser (Sec. 3.1), CSS parser (Sec. 3.2), URI parser (Sec. 3.3) and JavaScript parser (Sec. 3.4).

3.1. Reducing HTML parser influence

On the server side, we produce a parse tree from untrusted HTML using a modified version of a conservative, standards-compliant HTML parser [7]. The parse tree is transformed into a script-free approximation by pruning all nodes not allowed by a configurable whitelist [8] of known, non-executable content types. This whitelist accommodates a rich, expressive set of content consisting of HTML elements, attributes, style properties and link protocols all known to be static.

Static content types are types defined by web language standards and verified by experimental results not to invoke dynamic content such as scripts or browser plug-ins (e.g., Adobe Flash). The static content parse tree that results after pruning is marshalled into an encoded form (a *model*) suitable for embedding in the web application's output. The model encoding employs a map of content types from the whitelist to numeric values, which is decoded on the client-side using a trusted JavaScript library to reconstruct the parse tree.

The figure below gives an overview of this process. Typical web application output (a) may contain several instances of untrusted HTML. In our running example, each of these instances is a separate blog comment created by a user. Our implementation automatically generates and embeds a model for each comment. Thus the application's output is modified by replacing each instance of untrusted HTML with its corresponding model and leaving trusted content unaltered, as shown in (b).



We embed the model in web application output along with a short, trusted script that invokes the client-side JavaScript library, which in turn decodes and safely reconstructs the parse tree within the browser. Figure 3 shows an actual encoded model and accompanying script for the malicious comment in Figure 2(b). Model data is embedded in HTML by enclosing it in a `code` element. The model is never visible on the rendered web page because the `code` element's `display` property is set accordingly.

```
1 <code style="display:none;" id="__bp1">
2   =Enk/sCkhlcmUgaXMgYSBwYWdlIHlvdSBta...
3   =SkKICAgICI+dmVyeQ===C/k/QIGhlbHBmd...
4   =ECg===C/Enk/gCiAgUmVzcGVjdGZ1bGx5L...
5 </code><script id="__bp1s">
6   __bp__.cxPCData("__bp1", "__bp1s");
7 </script>
```

Figure 3. Encoded static model for blog comment (Figure 2(b)) as generated by the server-side BLUEPRINT library (Figure 4(b) lines 6–7), and trusted script to invoke the client-side BLUEPRINT model interpreter.

DOM interface	Lines in Figure 2(a)
`document.`	
`createElement(`a`);`	1, 3, 5, 7
`createTextNode(`a`);`	2, 4, 6, 8, 9
`getElementById(`a`);`	
`element``.`	
`appendChild(`a`);`	2–6, 8, 9
`insertBefore(`a, b`);`	1, 7
`parentNode;`	
`removeChild(`a`);`	
`setAttribute(`a, b`);`	5
`style[`a`] = `b`;`	7
`style.setExpression(`a, b`);`	7

Table 1. DOM APIs used for creating modeled web content such as the comment in Figure 2(a).

3.1.1. Client-side model interpreter

On the client-side, we decode models using trusted JavaScript code that we call the *model interpreter*. The model interpreter is embedded in the `head` element of the web application's output page as a link to a 15.6kB external script file. This library is cacheable by browsers to reduce load times. Decoding is performed by the model interpreter using a reverse map of numeric values to content types, and parse trees are constructed.

To programmatically reconstruct the parse tree, the model interpreter uses a small set of DOM interfaces that are present and exhibit consistent behavior in existing browsers. For instance, to create an HTML element we use `document.createElement()`, to create text content we use `document.createTextNode()` and to enforce hierarchical structure we use *`element`*`.appendChild()`.

Table 1 lists the specific DOM API calls we use to create untrusted content in the browser. As noted in Section 2, these API calls do not recursively invoke the browser's HTML or JavaScript parser and thus can be carefully used without risk of XSS attacks. Methods we use to process untrusted character data arguments are underlined in Table 1; the other calls do not process untrusted character data. For instance, `createElement()`, which is used to create an element with a defined type, takes a string of trusted characters from the whitelist (effectively an *enum* type) as its argument, and thus is free of untrusted content.

3.1.2. Model embedding strategies

The original rendering order of content on the page is preserved by our approach, as models are embedded at or near the original location of untrusted HTML, and models are interpreted synchronously as the page is rendered. When invoked, the model interpreter accepts a content model as input, constructs the parse tree, and subsequently removes the model and invocation script from the document.

Constraints on embedded model data. Since we embed models in HTML, the browser's HTML parser will unavoidably have to process these characters. If the browser could be tricked to interpret an unauthorized script within model data, XSS attacks can result. We drastically reduce this possibility and ensure reliable browser parsing behavior by imposing three general restrictions on models:

1) The model is embedded as text content in a `code` element (a narrowly defined HTML grammar context).
2) Characters used by the encoded representation are selected from a *syntactically inert* alphabet.
3) Text line lengths are conservatively restricted to a maximum length of 65 characters.

By imposing these constraints we restrict the browser's HTML parser to a simple and strictly defined role that (most importantly) does not require interpreting complex HTML syntax from untrusted data.

To restrict arbitrary untrusted text data to a sequence of syntactically-inert characters, we use the Base64 encoding scheme [9]. Base64-encoded strings can use a very limited alphabet that contains no HTML syntax control characters:

$$\{\mathrm{a}, \ldots, \mathrm{z}, \mathrm{A}, \ldots, \mathrm{Z}, 0, \ldots, 9, /, +, =\}^*$$

This encoding step enforces a whitelist of syntactically inert characters that ensure predictable transport of untrusted data through the browser's HTML parser.

If the web application serves pages using an obscure character encoding, care must be taken to ensure the employed Base64 alphabet is truly syntactically inert. For instance, the UTF-7 encoding uses the + character to alter the meaning of subsequent characters, which can be used to inject script using the standard Base64 alphabet. If stricter control of untrusted character data is required for an application, our approach can be adjusted to use an implementation-specific Base64 alphabet or more restrictive encodings such as Base32.

The additional constraint we impose is to limit the length of text lines in inserted model data. It is feasible that some browsers may be implemented without support for arbitrarily long lines of HTML, and may erroneously truncate or alter the data to suit requirements of the parser. We conservatively limit model data line lengths to 65 characters and have verified that all browsers supported by BLUEPRINT can reliably handle lines of this length.

Using techniques described in this section we ensure the web application's intended parse trees for untrusted HTML is adhered to by the browser, which addresses a significant number of XSS attack vectors. However, to fully enforce the no-script policy additional fine-grained control is needed for two types of parse tree nodes (CSS and URI), which we describe below.

3.2. Eliminating CSS parser influence

We return to the example of Figure 2. It is important for BLUEPRINT to permit use of the `style` attribute as in line 7(a), which allows the user to express *Cascading Style Sheet* (CSS) style / layout preferences for content. However, the particular syntax on line 7(b) exploits the `style` attribute and the browser's unreliable CSS language parser to inject a malicious script into the document. The available DOM APIs for creating CSS rules are not directly useful to explicitly disallow all executable CSS, so we employ additional defense techniques.

3.2.1. Disabling dynamic property values

The DOM API we use to create untrusted style properties is the *element*`.style` object. The whitelist we use contains a set of known static property names. By only using property names from the whitelist, we ensure dynamic properties that can invoke the browser's script interpreter are not created through our use of the style object. However this API is unsafe to use in Internet Explorer browsers since version 5.0 due to their support for special *dynamic property values* [10], which allow *any* property to contain executable code.

For example, if the user sets a CSS property using the `expression(...)` syntax, IE will interpret the argument in parenthesis as JavaScript code. Our initialization code detects the presence of this vector by attempting to execute a benign, trusted script. If dynamic property value support is detected, BLUEPRINT takes steps to ensure dynamic property values in untrusted content are rendered inoperable.

IE does not support a direct interface we can use for content creation that bypasses this XSS vector. However, the DOM does provide an indirect interface useful for this purpose: the `setExpression()` method. This method enables BLUEPRINT to embed a trusted script that will be executed whenever the browser requests any particular CSS property value. Through extensive experimental analysis, we have learned that the return value of this script is a *static property value* type and thus is not useful as an XSS injection vector. If a dynamic property value is returned, we have observed that the browser disregards the script and does not execute it.

Our approach for CSS property values is then a variation on the same theme we use for HTML. The goal is to transition the CSS parser to a state which does not trigger the dynamic property value parser, and thus eliminate this XSS vector. To achieve this goal, trusted scripts are added to the page via `setExpression()` for each untrusted style rule. Each trusted script added this way is executed by the browser as a function that dynamically computes a property value. As explained previously, the return type of this function is a *static* property value. Our function simply looks up the untrusted property value in an array then returns it as a static value. Thus our trusted script code explicitly creates static content and avoids APIs that can potentially create dynamic content.

3.2.2. Defending other CSS attack vectors

Some CSS properties can be used to embed script content without requiring the use of dynamic property values. For example, the `behavior` and `-moz-binding` properties allow embedding of script content. Our defense against these vectors uses a CSS whitelist consisting entirely of known static property names. Thus we use property names from the whitelist as arguments to the `style` object DOM API, and thereby ensure only static properties are ever created.

Similarly, CSS allows referencing of external style sheets using `@import` rules. If allowed, these external CSS files could embed XSS attack code so our model format and interpreter do not support `@import` rules.

Certain property values can also use the `url(...)` syntax to embed hypertext links, which are an XSS injection vector. BLUEPRINT prohibits this vector using a general defense strategy for links as described in the following section.

3.3. Response to URI parser threats

We return again to Figure 2. Line 5(a) depicts a benign use of *Uniform Resource Identifiers* (URI), or "links", which are a fundamental type of web content. URIs are most often used to indicate how the browser should retrieve a web resource (such as a web page or image file) when the resource is requested. URIs are composed of several components, the first of which is the *URI scheme* component, or "protocol".

Some URI schemes allow embedding of script code to be executed whenever the linked resource is requested, instead of directing the browser to a remote web resource. This is shown in line 5 of Figure 2(b), which uses the executable `javascript:` URI scheme. This scheme allows links of the form `<a href="javascript:...">` to execute malicious script code explicitly whenever a link is clicked, or implicitly when an image is loaded.

The `javascript:` scheme is a threat because no API exists for programming the browser to only use non-executable schemes for untrusted content. The decision over which scheme will be used is always controlled by the browser's URI parser. For example, the attack shown in Figure 2(b) is successful in IE 6.0, whose URI parser ignores the spaces and control characters to execute this URI [4]. To reduce the XSS threat posed by URIs, we propose a 3-tiered defense

consisting of: re-composing untrusted URIs, browser parse behavior sensing and impact mitigation. BLUEPRINT can use any combination of the these tiers for all URIs during model generation and interpretation.

Re-composing untrusted URIs. Our first step is to parse the untrusted URI into components. All URIs have a scheme component which indicates what other components (e.g., host, path, query) should be present based upon the scheme definition. We disallow the untrusted URI if the detected scheme component is not in a whitelist of non-executable schemes. In most benign URI, the employed scheme is one of: `ftp:`, `http:`, `https:` or `mailto:`.

Next, the URI is re-composed from its components using trusted data where possible. For example, the scheme component used in rewriting the URI comes from the whitelist, and the syntax characters that separate components (e.g., `/`, `?`, `#`) are trusted constants. Untrusted character data is percent-encoded [11] to ensure it is comprised only of syntactically inert characters. This process of reducing and encoding untrusted data ensures the URI is well formed, and declares our intent of which whitelisted scheme to use. If the browser is free of URI parsing quirks when given well-formed input, this process will successfully enforce the no-script policy for URI.

Sensing browser parse behavior. To create links we need to expose the re-composed URI to the browser's URI parser. As a safety net against parser misinterpretation, we employ a method of sensing browser URI parse behavior to potentially detect URI parser quirks. This technique attempts to observe whether the browser interprets the re-composed URI as having the correct scheme as intended by our approach.

BLUEPRINT performs this observation as each URI is processed during model interpretation. To do this carefully requires that we never expose the data used by the sensing code to an accidental URI resource request operation, such as an activation event (e.g., mouse click or loading of URI). We omit the details of the technique used here for space limitations, and refer the reader to the accompanying technical report [12] for a detailed discussion.

Mitigating XSS impact. BLUEPRINT also supports a third defense layer that forces all URI-embedded scripts not caught by the first two layers to execute in a sterile environment where no threat to sensitive data exists. Our approach rewrites all URIs as links to a redirection service hosted on a different-origin server, which the browser transparently follows to retrieve the actual untrusted URIs. The browser now treats the redirected URI request as if it came from this new domain, and URI-embedded script code (if any) executes in this domain. Due to *same-origin policy* restrictions imposed by browsers, these scripts are not allowed to access any data or state in the previous (i.e., "referring") page environment [13], [14]. For more details, we refer the reader to the technical report [12].

3.4. Reducing JavaScript parser influence

BLUEPRINT supports embedding of untrusted data in trusted scripts as string and numeric literals. This is a common use case for web applications that store user preferences in JavaScript variables for customizing runtime behavior of scripts. However, by doing so a web application may embed malicious user input which results in a successful XSS attack:

```
1 var preferredSiteTheme =
2  "';doEvil(...);</script><script>doEvil(...);";
3 var preferredFontSize =
4  12+doEvil(...);
```

Directly allowing untrusted characters in script contexts requires that they be interpreted strictly as data and never as executable code. Our approach encodes the untrusted data using syntactically inert characters and process them using trusted JavaScript code to enforce the intended data type, thereby avoiding any dependency on the browser's JavaScript parser to correctly infer the nature of the untrusted characters.

String literals. We embed untrusted string literals by replacing them with a call to the model interpreter, which returns the decoded string value to the calling context. Thus, line 2 of the above script would be replaced with the corresponding model:

```
1 var preferredSiteTheme =
2  __bp__.cxJSString("JyI7ZG9FdmlsKIUpOzwvc2N...");
```

Numeric literals. We implement numeric literals with the same technique used for string literals, but process and return a numeric value instead of a string. Thus, the model for line 4 of the above script would be:

```
3 var preferredFontSize =
4  __bp__.cxJSNumber("MTIrZG9FdmlsKIUpOw==");
```

4. Integration with web applications

The BLUEPRINT implementation consists of a server-side component and a client side script library. The server-side component is written in PHP which facilitates a natural integration with many popular PHP web applications. To support applications written in other languages, we have developed an alternative version of BLUEPRINT that runs in a separate process and communicates with the web application over a local TCP socket [12]. The client-side library is included in every web page output by the application by linking to an external JavaScript file.

To use BLUEPRINT, statements in a web application that output untrusted data need to be identified and instrumented with calls to our server-side module.

```
1  // Code for trusted blog content above^^.
2  // Code to emit untrusted comments below:
3
4  <?php foreach ($comments as $comment): ?>
5      <li>
6          <?php echo($comment); ?>
7      </li>
8  <?php endforeach; ?>
9
10 // Code for trusted footer follows...
```

```
1  // Code for trusted blog content
2  // appears untransformed above^^.
3  <?php foreach ($comments as $comment): ?>
4      <li>
5          <?php
6  $model = Blueprint::cxPCData($comment);
7  echo($model);
8          ?>
9      </li>
10 <?php endforeach; ?>
```

Figure 4. Example of (a) original PHP blog source code for emitting untrusted user comments, and (b) modified source code for automatically generating and emitting a script-free model (Figure 3) of untrusted user comments.

BLUEPRINT supports several *embedding contexts* that enforce application-intended constraints on untrusted content, such as whether the content is allowed to contain static HTML markup or must be plain text. Each instrumented call specifies the intended embedding context. We further discuss embedding contexts later in this section.

Figure 4(a) shows the PHP source for the blog application whose output was shown in Figure 2. Line 6 of this application outputs an untrusted comment. To use BLUEPRINT, this statement needs to be replaced by a call to the BLUEPRINT server-side interface as shown in Figure 4(b). Currently, our prototype relies on these changes being made by the developer of a web application. We note that these changes can be automatically made to web application source code by leveraging source-based taint tracking techniques (e.g., [15]). Once integrated into a web application as explained above, BLUEPRINT provides fully automatic protection.

4.1. Context-dependent embedding

The specific location of each embedded model in web application output depends on the application's intended use for the corresponding untrusted HTML. We define seven contexts in which BLUEPRINT supports embedding models of untrusted content. These contexts along with examples are shown in Table 2 and are explained below. We also describe the function of the client side model interpreter in each context.

Structured HTML context. HTML consisting of mixed elements and text is represented by the CXPCDATA context. In our running example, the blog comment in Figure 2(a) containing HTML appeared in this context, and the call to `Blueprint::cxPCData()` in Figure 4(b) outputs the encoded model for this context. On the client side, the BLUEPRINT model interpreter enforces a policy in this context that allows adding only child or sibling content nodes to the document tree. The content model for CXPCDATA is embedded directly in place of the original untrusted HTML.

Plain text contexts. Flat character data regions that are not allowed to contain nested elements or entity references, such as CDATA sections, are represented by the CXCDATA context. For models in this context, only a single text node will be added to the document tree. The content model for CXCDATA is embedded directly in place of the original untrusted text.

A second plain text context is the document title. Many web applications allow users to specify plain text to be contained in the `title` element of a web page. For instance, many bulletin-board systems display user-provided message subject lines in the title. To support this use case, BLUEPRINT defines the CXTITLE context to accept untrusted plain text to be inserted in the web page's `title` element.

Existing browsers require the `title` element to be a child of the document's `head` element. However, CXTITLE models are embedded as child nodes of the `body` element because it is safer to exclude untrusted data from the head, where the document character encoding is still being determined by the browser and because `code` elements cannot occur in the head.

Attribute and attribute value contexts. In these two contexts, CXATTRIB and CXATTRIBVAL, the model is embedded immediately after the open tag for elements that support nested (i.e., child) elements. For other elements, including *empty elements* which consist of a single tag (e.g., `<img />`) and no children, the model is embedded as the next *sibling* immediately after the element the attribute applies to.

JavaScript data contexts. Content models for CXJSSTRING and CXJSNUMBER are embedded in the trusted script directly where untrusted data is used.

5. Evaluation

We conducted extensive experiments to evaluate effectiveness and performance of BLUEPRINT on eight popular web browsers: Chrome 1.0; Firefox (versions 2.0 and 3.0); IE (v7.0 and 6.0); Opera 9.6; and Safari (v3.2 and 3.1). The total market share of these browsers was recently measured at over 96% [16]. Results of the

Context	Description	Example
CXATTRIB	Element attribute	`<td **align="center" nowrap**> ... </td>`
CXATTRIBVAL	Element attribute value	`<a href="**.../untrusted.html**"> ... </a>`
CXCDATA	Character data (CDATA)	`<![CDATA[ **untrusted** ]]>`
CXJSNUMBER	JavaScript numeric literal	`var x = **10**;`
CXJSSTRING	JavaScript string literal	`var x = **"untrusted"**;`
CXPCDATA	Parsed character data (PCDATA)	`<p> **untrusted <i>content</i>** </p>`
CXTITLE	Document title	`<title> Profile for user: **untrusted** </title>`

Table 2. Untrusted HTML embedding contexts supported by BLUEPRINT, as indicated by examples in **bold underline**.

evaluation fall into three categories: (1) compatibility with various types of social web applications, (2) effectiveness against attacks, and (3) several performance overheads.

5.1. Compatibility and expressiveness

For the following tests, we integrated BLUEPRINT with two popular web applications that produce HTML output based on untrusted user input: MediaWiki, the code base used for the web site Wikipedia, and WordPress, a popular blog application. Both applications directly allow HTML as input from untrusted users.

Integration with web applications. We integrated BLUEPRINT into the source code of both applications by adding calls to our server-side component to automatically generate and embed models at runtime. We modified three sensitive outputs in WordPress used for user comments to blog articles. WordPress did not exhibit any noticeable difference in functionality due to the BLUEPRINT integration.

MediaWiki was modified to protect the document title and the entire body of article (page-specific) content using BLUEPRINT, leaving the trusted border region unmodified. The document title and article content are derived from wiki markup given by the user, and thus should be defended from malicious input. We encountered a problem with MediaWiki, as it included a script (to hide and reveal the table of contents) within the untrusted article region — a dangerous practice. BLUEPRINT disallowed this script, but we easily relocated it into the trusted part of the page to restore functionality.

The effective ease with which we were able to integrate BLUEPRINT in two highly popular and expressive web application platforms demonstrates the high compatibility of our approach.

Content complexity. A primary objective of our approach is to allow untrusted content to be *expressive*, yet free from XSS attacks. For blog applications, untrusted content usually consists of user comments with simple markup and styling. A better test of the policies we employed is the wiki application, as the HTML and CSS markup used is much more complex and demanding. Elements are finely positioned, images are frequently embedded, and obscure constructs are sometimes used to fine-tune the look of a wiki page.

Our test suite included a wiki page that was *featured* by Wikipedia (available in [17]). This high standard is awarded to pages that are very well tuned, both in information accuracy and presentation. Therefore we used this page as a benchmark to demonstrate content expressiveness. In all browsers we tested, the wiki content rendered as well using our transformations as it did in original form.

We conclude that BLUEPRINT is permissive enough for many demanding web applications. The base whitelist we leveraged [8], combined with additional protections applied to potentially unsafe elements, allow a flexible yet safe palette for untrusted content.

5.2. Defense effectiveness

The primary goal of our defense is to be effective against a wide variety of XSS attacks. For our testing we chose XSS Cheat Sheet [4], which includes complex examples of XSS attack strings. Many of these examples are noteworthy for undermining sophisticated, real-world regular expression based defenses. Furthermore, the cheat sheet contains attacks that combine exploits of several browser parse quirks to achieve execution of arbitrary script commands.

We categorized all cheat sheet attacks according to vectors exploited. Also, we created an automated test platform to evaluate the threat of each attack vector on multiple browsers. To fully exercise vulnerabilities, the platform embeds attack strings in a variety of different contexts in a template web page. We then integrated BLUEPRINT into the test platform to evaluate its success rate in defending against attacks.

Results for this experiment are summarized in Table 3. At the time of testing, the growing XSS Cheat Sheet contained 113 entries. Of these, we classified 94 entries as XSS attack examples. The non-XSS attacks listed are: URI obfuscation (14x), cross-site request forgery (2x), server-side includes (1x) and PHP command injection (1x). The informational "attack" is *Character Encoding Example*. The test platform automates testing of 72 attacks in various contexts, leaving 22 for manual evaluation. For each of the eight browsers listed above, BLUEPRINT successfully

Type of attack	# of variations	# defended
Cross-site scripting	94	94
Other (non-XSS)	18	0
Informational	1	0
Total	113	94

Table 3. *XSS Cheat Sheet* defense effectiveness test results. Results are for all browsers listed in Sec. 5.

defended all 94 XSS attacks, thus protecting these browsers from their own parsing quirks.

Evaluating these 94 attack vectors on all browsers in our test set lends support for our strong XSS defense claims in Section 2. The attack strings were represented using a syntactically inert alphabet when exposed to browser parsers, and as a result there were no successful attacks due to browser parsing decisions. Because BLUEPRINT only employed reliable DOM APIs to construct content based on the attack strings, none of the untrusted input was recursively parsed resulting in an attack. Also, none of the URI attack vectors succeeded. Although our very limited expectations about browser parsing are difficult to prove (especially for closed-source browsers) they are well supported by our experimental results.

We also tested our defense against attacks on a well-known vulnerable system administration page in WordPress version 2.0.5 [18]. Our approach was successful at defending this vulnerability.

5.3. Resource utilization

We stove to stress our implementation and evaluate the performance of BLUEPRINT-enabled WordPress and MediaWiki under the worst possible conditions for varying amounts of embedded untrusted content.

To evaluate WordPress, we measured resource utilization of a real-world blog article as the number of untrusted user comments increased from 0–250, with an average of 1kB of HTML per comment (worst-case) before model generation. For MediaWiki, we measured resource utilization of the featured Wikipedia article tested above, as its untrusted content size grew in roughly 4kB increments.

Test scenario. For the WordPress tests, we considered all user-created comments as untrusted. These regions contain untrusted content in 3 contexts: the HTML comment body, plain text user name and user home page URI. In wiki pages, the document title and entire body of article content is under the control of users and can contain XSS attacks. We considered the title and all wiki article content as untrusted, allowing testing of large, complex content models.

Each tested page includes a link to our 15.6kB model interpreter, which was cached by the browsers in our

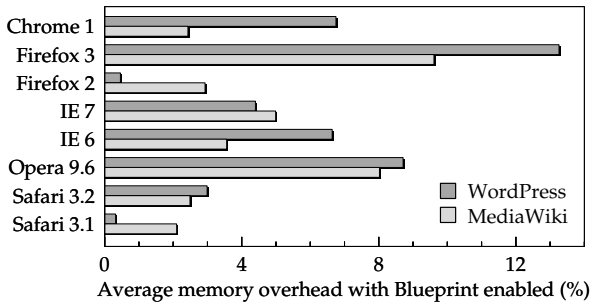


Figure 5. Average memory overhead for all tested WordPress and MediaWiki pages, by browser.

experiment. The server test platform was Apache 2.2.8 on Ubuntu 8.04 LTS Server, AMD Athlon 64 X2 Dual-core 4000+ CPU (2.1GHz), and 2GB RAM. The client test platform used Windows XP, as Windows is the only platform that natively supports all browsers in our test. The operating system was installed in a VirtualBox OSE virtual machine (allocated 1GB RAM), with host OS Ubuntu 8.10. Client hardware consisted of a ThinkPad X61s with Intel Core 2 Duo L7500 processor (1.60GHz), 4GB RAM.

Memory consumption. The total system memory consumed by each browser was measured for renderings of original, unmodified pages and pages implementing our XSS defense. Browser processes were set to initially display a blank page, and then navigated to a page in our test set for measurement. Results of this test are shown in Figure 5. We noted overheads ranging from zero to 13.6%, but averaging a modest 5%.

Page size overhead. The change in HTML size for pages with BLUEPRINT can be attributed to three factors:

1) per-model overhead due to embedding of model interpreter invocation scripts,
2) text size overhead due to Base-64 encoding, and
3) HTML markup size efficiency due to encoding of element tags, attributes and style properties.

To assess these effects, we compared the total size of untrusted content in the original HTML of each tested page to the total size of our model code in the modified page. The measured overhead for WordPress averaged 52.4%, and for MediaWiki averaged 13.9%. The difference between these can be attributed to the relatively high ratio of HTML markup to plain text content in MediaWiki, and the higher number of content models (534 models for 250 comments) in WordPress. HTML markup is very efficient in BLUEPRINT, as it takes only two bytes to encode a open / close tag or attribute / property name. However, text is less efficient as it incurs the 33% overhead due to Base-64 encoding. Also, our current implementation requires about 95

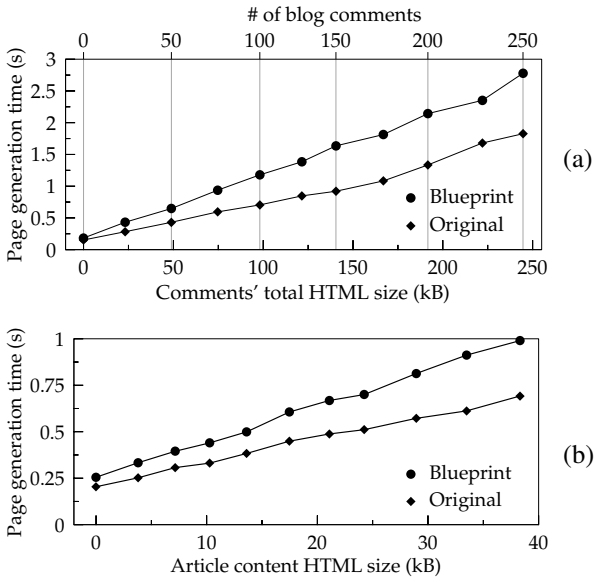


Figure 6. Time taken by server to generate (a) WordPress pages with increments of 25 user comments, and (b) MediaWiki article text as article size increases.

bytes of overhead for each embedded model, due to the addition of `<code>` tags and model interpreter invocation script.

Server page generation overhead. Time required by the server to generate web pages was measured during each of our tests and the results are given in Figure 6. We noted the average increase in processing time was 55% for WordPress and 35.6% for MediaWiki. It is important to note that each of these applications utilize built-in HTML parsing and sanitization functions that are redundant with the addition of parsing in BLUEPRINT. Eliminating this redundancy will be helpful in obtaining optimal performance from our defense.

There was some clear variation in performance between applications. The implementation of handling untrusted markup in these applications is very different: WordPress makes several (sometimes hundreds) of calls to BLUEPRINT for model generation, while MediaWiki only made 3 calls per page. To optimize page generation times on the server, the results of this test suggest using BLUEPRINT to generate fewer, larger content models can be an effective strategy. For example, instead of using separate models to convey the blog comment text, author name and URI, a single model encompassing the entire comment area could be used. This wider granularity comes at the expense of fine-grained control over specific areas of a web page that contain untrusted content.

Client page rendering latency increase. We measured the additional delay incurred while rendering BLUEPRINT-enabled pages on all 8 browsers in our test and

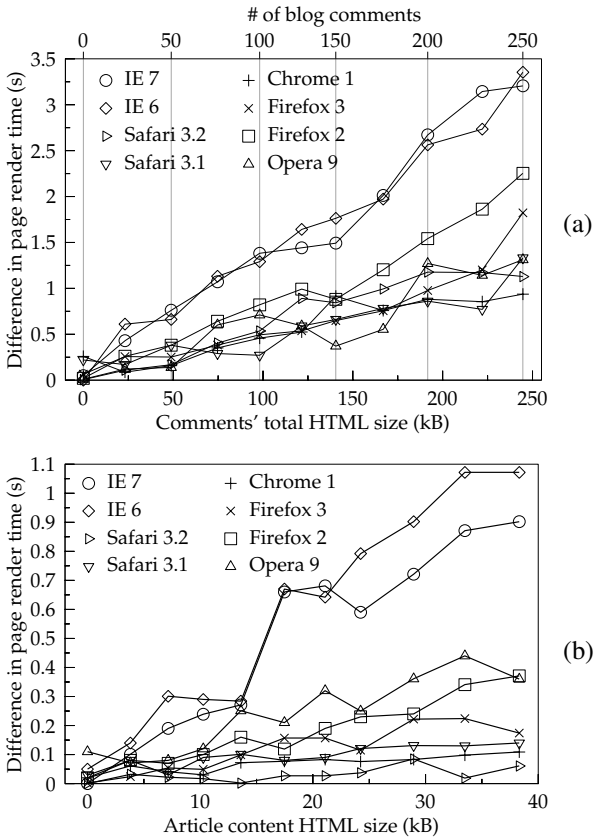


Figure 7. Additional time required to render each tested (a) WordPress blog page and (b) MediaWiki page when BLUEPRINT was enabled. Results vary by browser.

present the results in Figure 7. In these graphs, the horizontal axis represents the number of bytes the original untrusted HTML required. We feel this measurement serves to better apply our results to other applications, although the size of untrusted content models are another significant contributing factor to performance. (One can use average page size overhead results from this section to roughly convert between the two as needed.) The vertical axis shows the *extra* time required to completely render a BLUEPRINT-enabled page in the browser, when compared to the regular delay required to render each page without BLUEPRINT. To mimic worst-case browser processing overheads, our client measurements were taken by requesting pages from a server on the same local network, devoid of any effects of typical network transmission. As a result, the data in Figure 7 reflects delays on the server and delays on the browser due to changes in untrusted content, but does not reflect typical network delays.

We take a user-centric view in the discussion of client performance. For example, a user may take several tens of minutes to read a blog article including 250 comments. Using BLUEPRINT, the page is rendered with less than one second of latency in the best

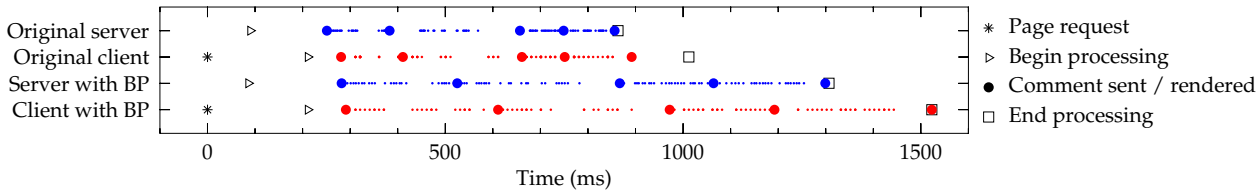


Figure 8. Comparison of client / server performance as affects end-user experience with and without BLUEPRINT. Results are from the experiment on WordPress with 100 untrusted user comments rendered by Firefox 2. Large dots depict every 25th comment sent (by the server) or rendered (by the client).

performing browser (Chrome 1) and only 3.4 seconds of latency in the worst performer (IE 6).

There is a large decrease in performance in IE browsers on the MediaWiki test at an article size of 18kB, due to an increase in markup-to-text ratio at that position (effects of which are discussed earlier) and well-known bottlenecks in the IE DOM implementation. Rendering latencies steadily increased as WordPress pages grew, while for MediaWiki tests, better performing browsers such as Safari 3.2 had relatively constant speeds. This effect can be attributed to the much more efficient DOM implementations in these fast browsers that rivals the speed of their own HTML interpreter, and the smaller file size of the wiki article (thus less network and server overheads).

CSS performance on IE. Continuing with the user-centric discussion, we look at how our IE-specific CSS transformation affects browser performance. Recall from Section 3.2.1, to fully protect against CSS-based XSS attacks, normally-static CSS property values are transformed into trusted dynamic content. The extra load of executing a short script every time the browser requests a style property value could negatively affect performance.

Of the applications we tested, only MediaWiki made use of this transformation. A total of 206 calls to `setExpression()` were made by our model interpreter while creating the full wiki article. Client rendering speed was not affected, as both tested IE versions had similar performance trends on both applications. Subjectively, we experimented with adjusting browser window geometry, which generated many thousands of calls requesting dynamic CSS property values, and did not perceive negative performance.

Effects on user experience due to delays. As a final assessment on the impact of BLUEPRINT on user browsing experience, we compare various stages in the page request / response cycle with and without our defense approach. The results of this test are shown in Figure 8. We observed that server speeds were very similar for both page requests until untrusted data regions are encountered. This can be seen in the very close times in which the first comment becomes visible to the user on both pages. Although there is an overall latency, Figure 8 indicates much of this delay is imperceptible to a user who reads the web page in a continuous manner from top to bottom. The results also suggest that overall delays can be mitigated by serving fewer comments per page as many advanced online discussion systems such as Slashdot do today. These systems select a few comments for initial display then retrieve more comments upon request from the user via Ajax remote procedure calls. BLUEPRINT fully supports this optimization, as the server can be programmed to send models instead of HTML comments in response to Ajax requests.

6. Related work

Vulnerability Analysis. There are several approaches [19], [20], [21] that rely on static source code analysis techniques to detect XSS injection vulnerabilities in web applications. These techniques identify sources (i.e., points of input) and sinks (e.g., query issuing locations), then check whether every source-to-sink data flow passes through an input validation function. Wassermann [22] and Balzarotti [23] also proposed solutions to the important problem of verifying the correctness of filter functions. These works address the important issue of analyzing and exposing a web application's vulnerabilities. However, as discussed in [24] and in Section 1, ensuring soundness of any filter function for script-free HTML is inherently a difficult problem due to unreliable browser behavior.

Client-side impact mitigation. Noxes [25] and NoMoXSS [26] are client-side XSS defenses that focus on ensuring confidentiality of sensitive data (e.g., cookies) by analyzing the flow of data through the browser, rather than preventing unauthorized script execution. These approaches empower users to protect themselves from XSS attacks without relying on web applications to implement an effective defense. However, attacks that do not violate same-origin policies are left unprotected by these schemes. Also, novice users cannot be expected to deploy a client-side XSS defense, and therefore web applications must provide additional security to protect all users.

Defense against reflected XSS attacks. Recent works have shown *reflected* XSS attacks can be detected by

comparing HTTP request parameters with response data output by the web application, and mitigated by filtering the response. This approach has been implemented on the client side [27], [28] and server side [29], [30], leveraging the unique perspective and enforcement capabilities of either side. These solutions are scalable and effective at preventing reflected XSS attacks without requiring support from the other end. Since they do not require web application changes, they can be rapidly deployed in any organization. However, these defenses can be manipulated to deny the execution of legitimate scripts via spoofing attacks. XSS-DS [29] uses a learning based approach to reduce these false alarms.

Defenses requiring changes to web standards or browser modifications. Enabling web applications to communicate web content policies to browsers for subsequent enforcement is an active research area. Works in this area [5], [31], [32], [33] leverage on the web application's ability to discern between trusted and untrusted web content, and the browser's ability to enforce web content policies. Results show this general approach is very good at preventing XSS attacks, and so these works are targeted at web standards writers and browser developers to facilitate adoption of the technique. Once standards evolve and browsers that support these techniques obtain significant deployment numbers, we expect web applications to start adopting this approach to reduce their vulnerability to XSS.

DSI [32], Noncespaces [33] and BLUEPRINT are all related in the common goal of preserving the integrity of document structure on the browser. Within this problem context, DSI explores in detail the issue of providing dynamic integrity protection, while Noncespaces explores the issue of supporting fine-grained policies such as allowed content types in untrusted content. The implementation approaches in both DSI and Noncespaces use randomized node delimiters for isolation and browser collaboration for policy enforcement. Both Noncespaces and DSI are designed to enable untrusted content to be displayed on existing browsers, but without any assurance about protection from XSS attacks on these browsers.

BLUEPRINT explores some of the open problems suggested in our previous works [34], [35] on XSS defense. In BLUEPRINT we strive to minimize trust placed on an existing browser to safely render untrusted content in a document. For dynamic protection, BLUEPRINT relies on careful implementation and testing of trusted scripts to safely handle untrusted data, by implementing safeguards such as disallowing direct writes or dynamically collaborating with the server using Ajax. Our implementation support for policies on untrusted HTML content is comparable to Noncespaces. In addition, we provide support for policy enforcement over untrusted data in JavaScript contexts.

Compared to all these approaches that leverage native browser support, BLUEPRINT requires more processing overhead on the web server and browser, and increased page sizes. However, BLUEPRINT has the important practical benefit of being an effective defense mechanism on currently deployed browsers in their default configurations.

Server-deployed XSS defenses. Several approaches in the literature [36], [37], [38], [15], [35] help prevent XSS attacks by utilizing dynamic tracking techniques to control the propagation and use of unsafe data in the web application. Although these defenses endeavor to sanitize untrusted output, they still leverage browser parsers to interpret untrusted HTML and therefore are susceptible to attacks that exploit browser parse quirks.

Many software tools [7], [39], [40], [41] use parsing as an advanced filtering method and normalize HTML output to minimize the risk of XSS attacks. Given the status of HTML parsing in various browsers, such advanced content mitigation schemes are a necessary first level of defense in preventing sophisticated XSS attacks. BLUEPRINT seeks to extend the capabilities of these approaches by minimizing the exposure of untrusted data to the browser's parser.

Use of JavaScript to mitigate XSS. In the past year or two, the application of client-side code to prevent or mitigate XSS attacks has been pursued by [42], [43], [44], [45]. BLUEPRINT systematically explores the application of this approach to minimize trust on a browser's parser, offers insights into the strengths of this idea, develops techniques for defending numerous XSS vectors, provides a novel implementation and comprehensive evaluation.

Di Paola [42] sketched the use of client-side JavaScript code to support a *data binding* scheme whereby a web application splits its output web page into two partitions: one consisting of entirely trusted HTML with destination markers for untrusted content, and a second partition containing the untrusted HTML. However, this partitioning has the unfortunate side effect of delaying the rendering of untrusted content until after all trusted content has rendered. BLUEPRINT carefully reasons through the paths traversed by untrusted HTML to minimize risks while embedding the untrusted content inline with trusted HTML. We therefore preserve the original in-order rendering sequence of the web page, and further demonstrate that the effect of our approach on the web experience of the end user is minimal.

NeatHtml [43] and Caja [44] take an alternate approach by performing parsing of untrusted HTML using a client-side trusted JavaScript library. After pars-

ing, untrusted content is filtered in a series of steps to ensure that it is free of script content. They both embed untrusted HTML in the web page using trusted client-side JavaScript code via the `innerHTML` DOM property (for Caja this is a deliberate optimization choice), which protects against node-splitting attacks. However, as discussed in Section 2, bypassing the parser and inclusion of untrusted content through `innerHTML` in a safe manner requires a very careful and complex escaping and filtering strategy, as illustrated by the nuanced implementations in these works. In contrast, BLUEPRINT does not rely on client-side filtering, and uses a much less complex client-side library that merely interprets declarative statements to programmatically create DOM nodes as intended by the server.

Confining the behavior of untrusted scripts. A number of recent efforts are actively focused on enabling safe execution of untrusted scripts by transforming JavaScript code [46], [47], [48], [49], [44], [45]. These efforts intervene on script operations to enforce high-level policy constraints. This is a finer-grained control than BLUEPRINT offers, as our approach conservatively disallows execution of all untrusted scripts even if they are benign (a sufficient requirement for many web applications such as blogs and wikis).

By restricting untrusted input to FBML [49] (an HTML-like markup language), Facebook allows embedding of FBJS [50] (a JavaScript-like script language) only in strictly defined contexts. The web application converts FBML to HTML and transforms FBJS to monitored JavaScript code. Felt [51] demonstrated this approach of restricting user input to a language that carefully limits the use of scripts is not sufficient for preventing unauthorized scripts in final output.

Caja [44] and Web Sandbox [45] are designed to safely implement dynamic gadgets for web mash-ups, and thus must allow the execution of untrusted scripts. They focus primarily on the difficult challenge of safely executing untrusted script content and do not exhaustively explore the problem of enabling a web application to exert control over how untrusted content is interpreted by the browser. In addressing this problem, BLUEPRINT takes extreme steps to prevent the influence of browser parsers over the way web content is generated from untrusted HTML. BLUEPRINT ensures untrusted data is only visible to the HTML, JavaScript and URI parsers as a sequence of syntactically-inert characters. Furthermore, our approach to setting CSS property values deliberately avoids the parsing state in IE's CSS implementation responsible for identifying script content. These measures lead to a strong assurance that BLUEPRINT can reproduce approved content models in unmodified browsers accurately, which is an important and fundamental goal for all of these works.

7. Conclusion

In this paper, we presented the design and implementation of BLUEPRINT, a robust prevention approach to cross-site scripting attacks that was demonstrably effective on existing web browsers comprising over 96% market share. By reducing the web application's dependency on unreliable browser parsers, we provide strong assurance that browsers will not execute unauthorized scripts in low-integrity application output.

XSS attacks can be avoided with less overheads by adding native browser support for enforcing high-level policies dictated by the web application. However, it takes a long time for standards to be agreed upon and browser vendors to implement these changes. BLUEPRINT is a promising and effective intermediate solution that can safeguard end-users from XSS attacks until standards and browsers evolve to provide universal, built-in support for disallowing unauthorized script execution.

A prototype implementation of BLUEPRINT is available for experimentation at the project website:

`http://sisl.rites.uic.edu/blueprint`

Acknowledgements

We thank Prithvi Bisht and Rohini Krishnamurthi for valuable insights and support with the server-side implementation. We also thank our shepherd Giovanni Vigna, R. Sekar and the anonymous reviewers for the helpful and thorough feedback received. This work was partially supported by National Science Foundation grants CNS-0716584 and CNS-0551660.

References

[1] S. Kamkar, "I'm popular," 2005, description and technical explanation of the JS.Spacehero (a.k.a. "Samy") MySpace worm. [Online]. Available: http://namb.la/popular

[2] OECD Directorate for Science, Technology and Industry, *Participative Web and User-Created Content: Web 2.0, Wikis and Social Networking*. OECD Publishing, Oct. 2007, ch. 2, pp. 19–25.

[3] B. Newton, "The hyper-growth of web 2.0 applications," Mar. 2008, seminar. [Online]. Available: http://www.innominds.com/webinar.html

[4] R. Hansen, "XSS (cross site scripting) cheat sheet esp: for filter evasion," 2008. [Online]. Available: http://ha.ckers.org/xss.html

[5] T. Jim, N. Swamy, and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies," in *16th International World Wide Web Conference*, Banff, AB, Canada, May 2007.

[6] World Wide Web Consortium, "Document object model (DOM) level 2 core specification," Nov. 2000. [Online]. Available: http://www.w3.org/TR/DOM-Level-2-Core/

[7] E. Z. Yang, "HTML Purifier." [Online]. Available: http://htmlpurifier.org

[8] ——, "HTML Purifier: Default whitelist." [Online]. Available: http://htmlpurifier.org/live/smoketests/printDefinition.php

[9] S. Josefsson, "The Base16, Base32, and Base64 data encodings," Jul. 2003, RFC 3548. [Online]. Available: http://tools.ietf.org/html/rfc3548

[10] M. Wallent, "About dynamic properties," 1998. [Online]. Available: http://msdn.microsoft.com/en-us/library/ms537634.aspx

[11] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform resource identifier (URI): Generic syntax," Jan. 2005, RFC 3986. [Online]. Available: http://tools.ietf.org/html/rfc3986

[12] M. Ter Louw and V. N. Venkatakrishnan, "Blueprint: Robust prevention of cross-site scripting attacks for existing browsers," University of Illinois at Chicago, Tech. Rep., May 2009.

[13] Wikipedia contributors, "Same origin policy," Feb. 2008. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Same_origin_policy&oldid=190222964

[14] World Wide Web Consortium, "HTML 4.01 specification," Dec. 1999. [Online]. Available: http://www.w3.org/TR/html4/

[15] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks," in *15th USENIX Security Symposium*, Vancouver, BC, Canada, Aug. 2006.

[16] Net Applications, "Browser version market share," statistics for Q4 2008. [Online]. Available: http://marketshare.hitslink.com/browser-market-share.aspx?qprid=2&qptimeframe=Q&qpsp=39

[17] Wikipedia Contributors, "2005 Azores subtropical storm," Nov. 2008. [Online]. Available: http://en.wikipedia.org/w/index.php?title=2005_Azores_subtropical_storm&oldid=243545716

[18] D. Kierznowski, "WordPress persistent XSS," Dec. 2006. [Online]. Available: http://michaeldaw.org/md-hacks/wordpress-persistent-xss/

[19] V. B. Livshits and M. S. Lam, "Finding security errors in Java programs with static analysis," in *14th Usenix Security Symposium*, Baltimore, MD, USA, Jul. 2005, pp. 271–286.

[20] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *15th USENIX Security Symposium*, Vancouver, BC, Canada, Aug. 2006.

[21] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2006.

[22] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *30th International Conference on Software Engineering*, Leipzig, Germany, May 2008.

[23] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2008.

[24] D. Wagner, "Answers to homework #1," 2008. [Online]. Available: http://www.cs.berkeley.edu/~daw/teaching/cs261-f08/hws/hw1sol.html

[25] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic, "Noxes: A client-side solution for mitigating cross-site scripting attacks," in *21st Annual ACM Symposium on Applied Computing*, Dijon, France, Apr. 2006.

[26] 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 *14th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, Feb. 2007.

[27] D. Ross, "IE 8 XSS filter architecture / implementation," Aug. 2008. [Online]. Available: http://blogs.technet.com/swi/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx

[28] G. Maone, "NoScript features: Anti-XSS protection." [Online]. Available: http://noscript.net/features#xss

[29] M. Johns, B. Engelmann, and J. Posegga, "XSSDS: Server-side detection of cross-site scripting attacks," in *24th Annual Computer Security Applications Conference*, Anaheim, CA, USA, Dec. 2008.

[30] R. Sekar, "An efficient black-box technique for defeating web application attacks," in *16th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, Feb. 2009.

[31] A. Felt, P. Hooimeijer, D. Evans, and W. Weimer, "Talking to strangers without taking their candy: Isolating proxied content," in *1st International Workshop on Social Network Systems*, Glasgow, Scotland, Apr. 2008.

[32] P. Saxena, D. Song, and Y. Nadji, "Document structure integrity: A robust basis for cross-site scripting defense," in *16th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, Feb. 2009.

[33] M. Van Gundy and H. Chen, "Noncespaces: Using randomization to enforce information flow tracking and thwart cross-site scripting attacks," in *16th Annual Network & Distributed System Security Symposium*, San Diego, CA, USA, Feb. 2009.

[34] M. Ter Louw, P. Bisht, and V. N. Venkatakrishnan, "Analysis of hypertext isolation techniques for cross-site scripting prevention," in *2nd Workshop in Web 2.0 Security and Privacy*, Oakland, CA, USA, May 2008.

[35] P. Bisht and V. N. Venkatakrishnan, "XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks," in *5th Conference on Detection of Intrusions & Malware, and Vulnerability Assessment*, Paris, France, Jul. 2008.

[36] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *22nd IFIP TC 7 Conference on System Modeling and Optimization*, Turin, Italy, Jul. 2005.

[37] T. Pietraszek and C. Vanden Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *8th International Symposium on Recent Advances in Intrusion Detection*, Seattle, WA, USA, Sep. 2005.

[38] Z. Su and G. Wassermann, "The essence of command injection attacks in web applications," in *33rd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, Charleston, SC, USA, Jan. 2006.

[39] "PHP input filter," 2008. [Online]. Available: http://www.phpclasses.org/browse/package/2189.html

[40] "The KSES project," 2008. [Online]. Available: http://sourceforge.net/projects/kses

[41] "The htmLawed project," 2008. [Online]. Available: http://www.bioinformatics.org/phplabware/internal_utilities/htmLawed/index.php

[42] S. Di Paola, "Preventing XSS with data binding." [Online]. Available: http://www.wisec.it/sectou.php?id=46c5843ea4900

[43] D. Brettle, "NeatHtml: Displaying untrusted content securely, efficiently, and accessibly," Jun. 2008, white paper. [Online]. Available: http://www.brettle.com/NeatHtml/docs/Fighting_XSS_with_JavaScript_Judo.html

[44] Google Caja, "A source-to-source translator for securing JavaScript-based web content." [Online]. Available: http://code.google.com/p/google-caja/

[45] Microsoft Live Labs, "Web Sandbox." [Online]. Available: http://websandbox.livelabs.com

[46] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "BrowserShield: Vulnerability-driven filtering of dynamic HTML," in *7th Symposium on Operating Systems Design and Implementation*, Seattle, WA, USA, Nov. 2006.

[47] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript instrumentation for browser security," in *34th Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, Nice, France, Jan. 2007.

[48] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov, "JavaScript instrumentation in practice," in *6th Asian Symposium on Programming Languages and Systems*, Bangalore, India, Dec. 2008.

[49] Facebook Developers, "Facebook markup language." [Online]. Available: http://wiki.developers.facebook.com/index.php/FBML

[50] ——, "Facebook JavaScript." [Online]. Available: http://wiki.developers.facebook.com/index.php/FBJS

[51] A. Felt, "Defacing Facebook: A security case study," Jul. 2007, white paper. [Online]. Available: http://www.cs.virginia.edu/felt/fbook/facebook-xss.pdf