

# Mutation-Event Transforms: A Flexible Client-side Foundation for End-to-end Web 2.0 Security

Úlfar Erlingsson

Benjamin Livshits

Yinglian Xie

*Microsoft Research*

## Abstract

It is our position that to reliably achieve Web application security goals, the server and the client must collaborate to enforce security policies. Such end-to-end security enforcement is particularly important in order to fully realize the promise of rich, cross-domain Web 2.0 applications. However, this improved security requires that Web clients be enhanced, and those enhancements must both be straightforward to implement and offer multiple, attractive benefits if they are to be adopted by popular Web browsers. We propose *Mutation-Event Transforms*, or METs, as a simple, flexible client-side mechanism for security policy enforcement. METs have the appealing property that simple policies are easy to specify and enforce; yet, METs are flexible enough to enforce even fine-grained, application-specific security policies.

## 1 End-to-end Security Enforcement

Web application security is based on the assumption that Web clients may be malicious and may craft arbitrary, malformed requests at any time. At most, non-malicious clients are assumed to enforce a rudimentary “same domain” security policy [8]. Since Web clients are not trusted to perform security enforcement, they are not even informed of simple Web application invariants, such as “no scripts in the email message portion of a page”.

At the same time, the majority of users are not malicious, and, if servers were to inform clients of Web application security policies, then those policies could be reliably enforced within users’ Web browsers. Even if only benign users with new, enhanced versions of Web browsers might perform security enforcement, those users would be protected, e.g., from exploits such as the Samy worm [9]. All users would benefit from fewer attacks on the servers, and, eventually, most users might be running browsers with enhanced client-side security.

Many security policies are best enforced at the client, such as policies that limit which servers are accessed, or policies that constrain where script code is found and from where it is loaded. Server-side enforcement has difficulties constraining even simple client behavior. For example, to enforce “no scripts”, server-side code must parse and sanitize all untrusted data and correctly model complex browser features and bugs to prevent the many convoluted means of embedding code for different browsers [7]. This type of data sanitation is difficult to

get right; bugs in such code are one of the most commonly reported software security vulnerabilities [4].

Furthermore, server-side enforcement is not sufficient for Web 2.0 applications, which often perform complex client processing involving code and data of disparate origin [11]. As one example, if a Web application server returns pages that use the Google Search AJAX API [2], that server will not know about the code and data loaded at the client as a result of Web searches.

Recently, a number of mechanisms have been proposed for enhanced Web client security enforcement. Some proposals, like BEEP [3], aim to prevent script injection and modify the Web browser to enforce a simple set of policies. Other proposals, like BrowserShield [6] and CoreScript [12], are more general but still have limitations; e.g., their proxy-based implementation raises the same challenges as server-side data sanitation.

Ideally, mechanisms for enhanced client-side security should share the Web browser’s view of code, data, and events, in order to ensure reliable enforcement. Also, to facilitate their adoption, such mechanisms should support multiple attractive policies. In particular, they should precisely enforce policies on both code and data (e.g., such as those in [10]), and be able to enforce even fine-grained, application-specific invariants, such as information-flow policies on user credit card input.

## 2 Mutation-Event Transforms

*Mutation-Event Transforms*, or METs, are a mechanism for client-side security enforcement that meets the above standard. METs are flexible enough to enforce all security policies that refine the “same domain” policy and are based on monitoring Web client behavior. This flexibility results from METs being inlined reference monitors for security-relevant client-side events [1].

With METs, the Web server specifies programmatic security policies by including code for JavaScript callback functions in the first script executed in returned Web pages. This code may stem from security libraries, be written by hand, or be derived through automatic analysis. The Web client enforces these policies by invoking the callbacks on each page modification, or mutation event, including initial page loading. The callbacks ensure that the new (or updated) page conforms to the security policy, either through validation or transformation of the code or data. (Because policies are programmatic, they can readily account for browser variation.)

Implementing support for METs is straightforward: Web clients need only add a single new primitive for mutation-event callbacks and expose already-present events and data structures. These changes are similar to, and simpler than, those for DOM2 mutation events [5], but require a more expressive, extended DOM that includes the abstract syntax trees of executable scripts. In fact, in the latest Opera browsers, the mechanisms for DOM2 events and “User JavaScript” could implement support for METs, without much modification.

METs have the appealing property that simple policies are easy to specify and enforce. Figure 1 shows how some example client-side policies can be readily instantiated using MET callback functions, given the type signature for such functions shown below. These policies vary in complexity, but demonstrate how METs can usefully constrain both code and data, to the point of a custom interpreter that could implement dynamic taint propagation or other complex security policies.

```
ExtendedNode
MET_callback(in Node script, // source of mutation
             in Node target, // target parent in DOM
             in ExtendedNode oldValue,
             in ExtendedNode newValue);
```

MET callbacks may be registered for particular types of DOM elements; they are invoked right before element mutation, but after the Web client has parsed the new extended DOM value proposed for the element. Both the `script` and `target` are DOM nodes, with `script` being the node containing a script that is attempting a page mutation. The `target` is where `newValue` is about to be inserted to replace `oldValue`; both values are well-formed, properly-nested subtrees of our extended DOM, or null to denote empty subtrees. The return value is eventually what is inserted.

Security enforcement using METs requires the resolution of many details that space prohibits describing here fully. Crucially, as with all inlined reference monitors, the security policy code must take steps to ensure that METs cannot be subverted or circumvented, and that they completely mediate security-relevant events [1]. This may involve significant machinery; for instance, function closures to protect the integrity of METs internal state, and restrictions on operations with dangerous side effects (such as modifying JavaScript prototypes) within all Web page scripts. Also, enforcement requires properly handling calls to `eval` (e.g. creating an extended DOM node with the caller as the parent node) and `<SCRIPT>` nodes with `SRC` attributes (e.g., generating MET callbacks both on creation and when scripts arrive from a `SRC` URL). Finally, METs might be extended to allow their composition—as long as a root, trusted policy applies to the final value of each mutation—and with extra functionality, e.g., to allow a server to prevent all network requests originating from pages in other domains.

---

#### Limit OBJECT nodes (on OBJECT events):

```
var ok = (newValue.classid == theAllowedGUID);
return (ok) ? newValue : null;
```

#### Data invariant on blog comments (on DIV events):

```
if (target.id != blogCommentsID) return null;
var ok = ExtDOM.MatchStructure(newValue,
    "<div><ul><li></li></ul></div>");
return (ok) ? newValue : null;
```

#### Most scripts cannot modify the DOM (on all events):

```
var ok = isItTheSame(script, specialGoodScript);
return (ok) ? newValue : oldValue;
```

#### Limit script placement (on SCRIPT events):

```
var ok = ! findParentAttr("no_scripts", target);
return (ok) ? newValue : null;
```

#### Proper JSON in AJAX replies (on SCRIPT events):

```
if (! newValue instanceof ScriptBody) return null;
var ok = ExtDOM.MatchScriptStructure(newValue,
    "callback({count: 1; sum: 5;});");
return (ok) ? newValue : null;
```

#### Secure interpreter (on SCRIPT events):

```
if (! newValue instanceof ScriptBody) return null;
var arg = ExtDOM.CreateScriptNode("Arguments",
    newValue.ToJSON());
var func = "special_js_interpreter";
return ExtDOM.CreateScriptNode("Call", func, arg);
```

**Figure 1:** Examples of the programmatic specification of security policies using MET callback functions.

End-to-end Web application security policies prevent client behavior and server interaction that should be impossible, by construction, or has been determined to be illegal. Whether policies are driven by automatic analysis, or by manual access control, METs are a flexible, reliable foundation for their client-side enforcement.

## References

- [1] Ú. Erlingsson and F. Schneider. IRM enforcement of Java stack inspection. In *Proc. IEEE Symp. on Security and Privacy*, 2000.
- [2] Google AJAX search API. <http://code.google.com/apis/ajaxsearch>.
- [3] T. Jim, N. Swamy, and M. Hicks. BEEP: Browser-enforced embedded policies. In *Proc. WWW*, 2007.
- [4] MITRE. Common vulnerabilities and exposures. <http://cve.mitre.org/cve/>, 2007.
- [5] T. Pixley. DOM level 2 events specification. <http://www.w3.org/TR/DOM-Level-2-Events>, 2000.
- [6] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proc. OSDI*, 2006.
- [7] RSnake. XSS (Cross Site Scripting) cheat sheet. <http://hackers.org/xss.html>, 2006.
- [8] Same origin policy. [http://en.wikipedia.org/wiki/Same\\_origin\\_policy](http://en.wikipedia.org/wiki/Same_origin_policy), 2007.
- [9] The Samy worm. <http://namb.la/popular/>, Oct. 2005.
- [10] Z. Su and G. Wassermann. The essence of command injection attacks in Web applications. In *Proc. POPL*, 2006.
- [11] Web Mashup. <http://www.webmashup.com>.
- [12] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *Proc. POPL*, 2007.