End-to-end Web Application Security

Úlfar Erlingsson Benjamin Livshits Yinglian Xie

Microsoft Research

Abstract

Web applications are important, ubiquitous distributed systems whose current security relies primarily on server-side mechanisms. This paper makes the end-to-end argument that the client and server must collaborate to achieve security goals, to eliminate common security exploits, and to secure the emerging class of rich, cross-domain Web applications referred to as Web 2.0.

In order to support end-to-end security, Web clients must be enhanced. We introduce *Mutation-Event Transforms*: an easy-to-use client-side mechanism that can enforce even fine-grained, application-specific security policies, and whose implementation requires only straightforward changes to existing Web browsers. We give numerous examples of attractive, new security policies that demonstrate the advantages of end-to-end Web application security and of our proposed mechanism.

1 Introduction

Web applications provide end users with client access to server functionality through a set of Web pages. These pages often contain script code to be executed dynamically within the client Web browser.

Most Web applications aim to enforce simple, intuitive security policies, such as, for Web-based email, disallowing any scripts in untrusted email messages. Even so, Web applications are currently subject to a plethora of successful attacks, such as cross-site scripting, cookie theft, session riding, browser hijacking, and the recent self-propagating worms in Web-based email and social networking sites [2, 17, 24]. Indeed, according to surveys, security issues in Web applications are the most commonly reported vulnerabilities on the Internet [16].

The problems of Web application security are only becoming worse with the recent trends towards richer, "Web 2.0" applications. These applications enable new avenues of attacks by making use of complex, asynchronous client-side scripts, and by combining services across Web application domains [8]. However, the shift towards Web 2.0 also presents an opportunity for enhanced security enforcement, since new mechanisms are again being added to popular Web browsers.

Therefore, we believe it is time to rethink the fundamentals of Web application security. It is our position that the client Web browsers must be given a greater role in enforcing application security policies. In this paper, we support our position with examples and a simple end-to-end argument: constraints on client behavior are enforced most reliably at the client. We also propose *Mutation-Event Transforms*: a novel, flexible mechanism for client-side enforcement.

1.1 Motivating Attacks

Of the current attacks on Web applications, those based on *script injection* are by far the most prominent. For example, script injection is used in cross-site scripting [1] and Web application worms [2, 24].

A script injection vulnerability may be present whenever a Web application includes data of uncertain origin in its Web pages; a third-party comment on a blog page is an example of such untrusted data. In a typical attack, malicious data with surreptitiously embedded scripts is included in requests to a benign Web application server; later, the server may include that data, and those scripts, in Web pages it returns to unsuspecting users. Since Web browsers execute scripts on a page with Web application authority, these returned scripts can give attackers control over the users' Web application activities [1, 22].

Script injection attacks typically affect non-malicious users and succeed without compromising Web application servers or networks. For example, in 2005, the self-propagating Samy worm on MySpace used script injection to infect over a million users [24]. As a MySpace user viewed the MySpace page of another, infected user, the worm script would execute and send a page update request to the server, causing the worm script to be included also on the viewing user's page.

In an attempt to prevent script injection, most Web application servers try to carefully filter out scripts from untrusted data. Unfortunately, such data sanitization is highly error prone (see Section 2.1). For example, the Samy worm evaded filtering, in part, by the unexpected placement of a newline character [24].

Script injection is just one means of attack: there are many ways to exploit Web applications by presenting them with attacker-chosen data. As we demonstrate in this paper, end-to-end Web application security is not only a reliable means to prevent these attacks. Our proposals for enhanced, client-side security enforcement also form a simple, flexible foundation for the general security of Web applications, including future, more complex Web 2.0 applications.

2 The Case for End-to-end Defenses

In general, it is often best to establish systems guarantees at the point where they are needed, with an end-to-end check, rather than with earlier, piecemeal checks [21].

This end-to-end argument applies directly to Web application security. Although security policies should be determined and specified at the server, enforcement of policies about Web client behavior should be guaranteed at the client. The corresponding server-side checks are difficult to perform and, in practice, incomplete in ways that enable attacks.

2.1 Server-side Defenses and their Limitations

Web applications must consider the possibility of malicious attackers that craft arbitrary messages, and counter this threat through server-side mechanisms.

However, to date, Web application development has focused only on methodologies and tools for server-side security enforcement (for instance, see [11, 13]). At most, non-malicious Web clients have been assumed to enforce a rudimentary "same origin" security policy [22]. Web clients are not even informed of simple Web application invariants, such as "no scripts in the email message portion of a page", since clients are not trusted to enforce security policies.

This focus on centralized server-side security mechanisms is shortsighted: server-side enforcement has difficulties constraining even simple client behavior. For example, to enforce "no scripts", the server must correctly model complex, dynamic client activities such as string manipulation, and take into account all possible client features and bugs. This entails server consideration of a myriad different tags, encodings, and operators for comments and quoting [20].

Server-side removal of scripts is especially difficult for Web applications that wish to allow visual formatting or other data richer than simple text. As shown below, there are many non-obvious means of causing code execution, including within formatting tags:

```
<SCRIPT/chaff>code</S\0CRIPT>
<IMG SRC=" &#14;  code">
<STYLE>li {list-style-image: url("code");}</STYLE>
<DIV STYLE="background-image:\0075\0072\006C...">
```

Furthermore, server-side enforcement is unsuitable for Web 2.0 cross-domain mashups [25], which may access third-party servers to load code and data. For instance, Web clients perform such accesses whenever a Web applications embeds the Google Search AJAX API [5].

2.2 Client-side Defenses and their Benefits

As described above, many security policies are best enforced at the client. Web clients are the final authority on client behavior—including where script code is found, what that code is, and from where the code was loaded. If informed of Web application security policies by the server, properly enhanced clients could reliably enforce those policies.

```
HTMLDocument.prototype.__defineGetter__(
    "cookie",
    function(){ return null; }
);
```

Figure 1: A programmatic security policy that will reliably disallow all script access to document cookies in many existing Web browsers, if included at the top of pages returned by a Web application server [3].

At the same time, the majority of users are not malicious, and would enable client-side enforcement to avoid exploits such as cross-site scripting and Web-based worms. Even if only benign users with enhanced clients might perform security enforcement, those users would be protected, and all users would benefit from fewer attacks on the Web application.

Unfortunately, there are many practical obstacles to the adoption of new, enhanced security mechanisms in popular Web browsers. Even when such enhancements are practical and easy to implement, they may not be deployed widely. Therefore, to increase its chance of widespread adoption, a Web client security mechanism should be practical, simple, and flexible, and be able to enforce multiple, attractive policies on client behavior.

3 New Client-side Security Mechanisms

In this paper we propose enhancing Web clients with new security mechanisms that can not only prevent existing attacks, but are able to enforce all security policies based on monitoring client behavior. In particular, our new mechanisms support policies that range from disallowing use of certain Web client features (e.g., `IFRAME`s or `OBJECT`s) to fine-grained, application-specific invariants such as taint-based policies that regulate the flow of credit-card information input by the user.

Concretely, we propose that client-side enforcement proceed through a new client mechanism: *Mutation-Event Transforms*, or METs. METs are introduce here; some details like how to prevent their subversion are in Appendix A. METs allow Web application security policies to be specified at the server in a programmatic manner, such that those specifications can be used directly for enforcement at the client. In this, METs are similar to the code in Figure 1, and recent proposals such as BEEP [9].

In short, with METs, Web application servers specify security policies as JavaScript functions included at the top of pages returned by the server, and run before any other scripts. At runtime, and during initial loading, these MET functions are invoked by the client on each Web page modification to ensure the page always conforms to the security policy. Before a mutation takes effect, METs have the ability to transform that mutation, and the code and data of the page, which gives METs great flexibility in enforcement. In particular, METs can

be used to implement inlined reference monitors and edit automata for security-relevant client events, which allows METs to be used to specify and enforce any security policy based on monitoring client behavior [4, 26].

METs are both simple and straightforward to adopt: Web clients need only implement a single new primitive for mutation-event callbacks, and expose already-present events and data structures. Because policies are programmatic, they can readily account for browser variation and properly limit client-side enforcement on legacy Web clients (indeed, JavaScript code is already commonly used for compatibility and debugging purposes). Furthermore, security policy enforcement using METs requires only reasonable assumptions about the attacker.

3.1 Assumptions about the Attacker

METs can reliably defend against powerful attackers that are able to present Web clients with arbitrary code and data. In particular, the attacker may be modeled as an arbitrary, malicious script within Web application pages that are subject to MET enforcement.

The correctness of MET enforcement is of concern only to non-malicious users; it relies on network integrity and depends on assumptions about the server and clients. We trust that the Web application server has not been compromised, and properly includes METs at the top of returned Web pages; however, we assume that server code may have bugs such that the returned pages may contain arbitrary attacker-chosen data. We trust the Web clients to execute METs with proper semantics and to correctly enforce the fundamental same-origin policy [22]. Finally, we trust the programmatic security policies and that they correctly reflect the security goals of the Web application developers.

4 Policy Specification and Enforcement

Web application developers must have freedom in choosing security policies, and how they are derived. We propose specifying security policies using programmatic MET callback functions written in JavaScript. At runtime, these MET callback functions operate on each new (or updated) Web page and ensure that it conforms to the security policy, either through validation or transformation of the code or data within the Web page.

As we demonstrate in this section, METs have the appealing property that simple policies are easy to specify and enforce (much as in Figure 1). Even so, although Web application developers may guide security enforcement with Web page annotations, code for METs is likely to come as pre-packaged libraries, or be determined automatically at the server.

In particular, METs can be used for client-side enforcement of application-specific *dynamic security policies* determined automatically at the server from the natural constraints imposed by the structured composition of client pages (e.g., using frameworks such as ASP.NET AJAX [14] or GWT [6]). METs can also enforce other rich policies, such as those that apply to Web 2.0 cross-domain mashups [25], where application pages are composed outside the scope of server enforcement.

4.1 Examples of General, Basic Security Policies

On the following page, Figure 2 describes examples of general policies that apply to client Web pages, their script code, and the nodes and attributes of document data. On the same page, Figure 3 shows how these policies can be readily instantiated using MET callback functions; this code should be read in conjunction with Appendix A. In what follows, these policies are referred to by their number, in parentheses.

Policies (1), (3), and (6) are examples that restrict potentially dangerous types of document nodes, allow scripts only in certain portions of the document, or limit scripts to a whitelist of trusted scripts (as in [9]).

Policies (2), (4), and (5) validate the structure of certain data structures and scripts in Web pages, which can prevent many attacks (e.g., attacks that use malformed SQL queries [23]). . Without such validation, malicious attacks may exploit may benign client-side code by presenting it with malformed data. For instance, without enforcement of policy (5), the Web client may at any time execute new, unexpected code where only a data return value was expected. (Client-side data validation may also reduce round trips to the server.)

As shown in policies (7) through (9), the set of policies supported by METs are not restricted to actions that change the document structure of Web pages. METs can also support constraints on network access or access to security-critical client variables, such as the Web browser history, and enforce containment scopes between client-side gadgets and modules.

Finally, as demonstrated by (10), security policies based on METs may even include the code for a security-enhanced JavaScript interpreter, and ensure that it is used to execute all script code. Such a custom interpreter can implement dynamic taint propagation or other complex security policies.

For reasons of space, our example METs use several support routines, that would naturally be defined by security policy code. For example, the `matchURLDomain` function in (8) might match string URLs in a policy-specific manner, while the `outmostAttr` function in (9) might recursively walk up the document tree in order to find the attribute definition closest to the root. Similarly, policy-specific variables may encode security-relevant state such as in (1) for allowed ActiveX GUIDs (e.g., the Flash player), and in (2) the identity of a particular node in the document structure of a Web page.

(1) Disallow certain dangerous nodes or attributes. For instance, <IFRAME> nodes might be disallowed, and <OBJECT> nodes only permitted when instantiating the Flash player with known content.

(2) Data invariants on certain document subtrees. The Web page document is subject to invariants, even when modified dynamically at the client; e.g., blog comments must be a well-formed list of <DIV> nodes.

(3) Disallow scripts in certain parts of a Web page. A special case of (2), for instance to disallow use of <SCRIPT> nodes in untrusted blog comments.

(4) Scripts match valid, server-defined templates. An application of (2) to scripts: new, client-defined scripts may be allowed, but, for example, the `onHover` script code for a dynamically-inserted list item might be required to match `highlight(identifier)`.

(5) Cross-domain scripts return only data, properly. Instantiating (4) to prevent unexpected introduction of new code by cross-domain client-mashup applications: for instance, any script returned into a cross-domain <SCRIPT> node must have a syntax tree that matches `ajaxCallback(jsonDataValue)`.

(6) Limit scripts to a static, server-defined set. A static form of (4) that may simply match the hash of the script source text against a fixed "whitelist".

(7) Constrained access to object fields and methods. For instance, giving partial access to `document.cookie`, or limiting arguments to network-access methods.

(8) Proper network access via (cross-domain) URLs. URLs are subject to access control—both node-attribute URLs (e.g., on <IMG>) and the URLs used programmatically in scripts, (e.g., in an XML request).

(9) Containment of script activity to certain subtrees. Scripts can only modify certain document subtrees; thus, a gadget (or client-side mashup) for Web search might only be allowed to mutate a <DIV> for search results.

(10) Script execution by a secure interpreter. Scripts are not executed directly, but through a special, security-enhanced interpreter that may enforce (8), above, or even more fine-grained policies, such as variants of stack inspection or data tainting [4, 13].

Figure 2: A selection of attractive client-side security policies that can be readily enforced using programmatic MET callback functions. This list emphasizes general, widely applicable policies, while application-specific dynamic security policies are discussed further in the text (in particular in Section 4.2).

Type signature for MET callback functions

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

Example programmatic MET callback policies

(1) Limit `OBJECT` nodes (on `OBJECT` events):

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

(2) Data invariant on inserted `DIV`s (on `DIV` events):

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

(3) Limit script placement (on `SCRIPT` events):

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

(4) Restrict the code in scripts (on `SCRIPT` events):

```
var ok = ExtDOM.MatchScriptStructure(newValue,
                "highlight('identifier');"
return (ok) ? newValue : null;
```

(5) Proper data 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;
```

(6) Script whitelisting (on `SCRIPT` events):

```
var ok = whitelist[ hash(newValue.toString()) ];
return (ok) ? newValue : null;
```

(7) Disallow `history` access (on `SCRIPT` events):

```
if (! newValue instanceof ScriptBody) return null;
return ExtDOM.ReplaceScriptLiteral(newValue,
              "history", "fresh_unused_literal");
```

(8) Limit network access (on `SCRIPT` events):

```
var ok = matchURLDomain(newValue.src, "foo.com");
return (ok) ? newValue : null;
```

(9) Script containment (on any mutation event):

```
var src = outmostAttr("containment", script);
var dst = outmostAttr("containment", target);
return (src == dst) ? newValue : null;
```

(10) 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 3: The type signature of MET callback functions and several possible implementations for policies like those in Figure 3. The details in Appendix A are relevant to this code. Due to space constraints, only terse, uncommented code is shown and the functionality of policy-provided variables and methods is indicated by name.

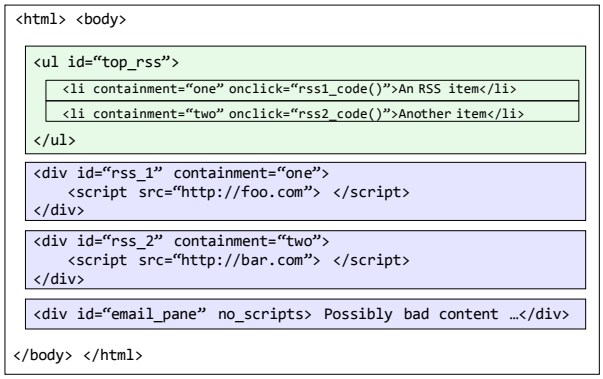

Figure 4: An outline of the document tree for an aggregation Web page that contains both RSS news items and email messages.

4.2 Application-Specific, Dynamic Security Policies

Policies can also be highly application-specific. Such policies can be either hand-written by the application developer or generated through static analysis of the Web application. This is illustrated by the examples below.

Example 1. Access control within a page. Figure 4 shows an example of a DOM tree containing data from two RSS sources: `rss_1` and `rss_2`. We would like to make sure that `rss2_code` does not modify the first element so that it is impossible to have a rogue RSS feed that changes the contents of another one. Using policy (9), we can restrict code to modify only DOM elements declared within the same scope. This policy allows isolation of code and data on a single page, and refines the "same-origin" policy of existing Web clients.

Figure 4 also shows how security policy can be directed by inline attributes on document nodes. In this case, a `no_scripts` attribute is used to direct MET enforcement of a policy such as (3) in Section 4.1.

Example 2. Google Web Toolkit (GWT). In GWT, the developer writes his or her application in Java [6]; the application is subsequently compiled by the GWT into two parts: a Java part that resides on the server and a JavaScript part that resides on the client. Unfortunately, given a client-side attack, the assumptions of the original Java application may not hold for the scripts at the client,

To prevent this, the server may generate policies that enforce consistency properties of the client code. For example, the server may wish to ensure that access control properties such as "a private method may only be invoked by methods in the same Java class" present in the original Java source code are preserved in the JavaScript code on the client side. Instantiation of such a policy would be application-dependent and could be obtained through static analysis of the original Java code.

Example 3. Server-generated content templates. Dynamic policy generation is also relevant to ASP.NET or JSP pages. Both of these technologies allow servers to mix static HTML and dynamic content. Using static analysis (e.g., that in [15]), the computed parts of Web pages can be approximated and, thereby, the structure and contents of generated pages. For example, the analysis may be directed to assume no permitted scripts in application inputs. Such page "templates" are highly suitable for client-side enforcement.

5 Discussion

End-to-end Web application security entails preventing client behavior and server interaction that should be impossible, by construction, or has otherwise been determined to be illegal. Whether policies are driven by automatic analysis, or by manual setting of policy, there is much to gain from this form of security. In particular, it is a necessary foundation for securing Web 2.0 applications like cross-domain mashups, which are often outside the scope of existing mechanisms.

Mutation-event transforms, or METs, are an attractive option for client-side security. METs are flexible enough to enforce any security policy based on execution monitoring [4, 26]. In particular, METs readily allow precise enforcement of policies on both code and data (e.g., such as those in [23]). At the same time, METs, and their supporting code, should be straightforward to implement, since they rely only on existing browser events and data structures.

In comparison, the servers can leverage the "same origin" security policy [22] to enforce some client-side policies, as done in SessionSafe [10]. Such schemes require multiple, elaborate server domains that may be cumbersome to manage. Even so, they can provide only limited, coarse protection such as disallowing access to Web application cookies—as in policy (7) in Section 4.1.

Some previous proposals enforce client-side security policies by making use of separate proxies to rewrite server requests from the Web client. Noxes [12] places simple restrictions on the URLs of requests. BrowserShield [19] and CoreScript [26] use elaborate script rewriting techniques to enforce policies such as disallowing cookie access and dangerous tags—as in policies (1) and (7) in Section 4.1. Although they are useful (e.g., for legacy support), such proxy-based mechanisms must correctly parse data and code in requests, which can be a near-intractable problem, even using structured, formal methods (see Section 2.1 and [26, Section 6]).

Indeed, like METs, reliable mechanisms for client-side security policies must necessarily build on the final parsing of code and data performed at the Web client. This approach has been taken in previous mechanisms, most notably in BEEP [9] but also in [7]. However, these proposed mechanisms have provided little flexibility in security policy specification and enforcement, only supporting policies like (3), (6), and (7) in Figure 2.

The enforcement of end-to-end security policies offers benefits to all Web application users, but requires changes to existing Web browsers. The inclusion of our proposed METs mechanisms in Web clients can reliably prevent existing attacks and provide a flexible, fine-grained foundation for the enforcement of future application-specific security policies

References

[1] CGI Security. The cross-site scripting FAQ. http://www.cgisecurity.net/articles/xss-faq.shtml.

[2] E. Chien. Malicious Yahooligans. http://www.symantec.com/avcenter/reference/malicious.yahooligans.pdf, 2006.

[3] S. Di Paola. Wisec security. http://www.wisec.it/sectou.php?id=44c7949f6de03, 2006.

[4] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proc. IEEE Security and Privacy*, 2000.

[5] Google AJAX search API. http://code.google.com/apis/ajaxsearch.

[6] Google Web toolkit. http://code.google.com/webtoolkit.

[7] O. Hallaraker and G. Vigna. Detecting malicious JavaScript code in Mozilla. In *Proc. IEEE Conf. on Engineering of Complex Computer Systems*, 2005.

[8] B. Hoffman. Ajax security. http://www.spidynamics.com/assets/documents/AJAXdangers.pdf, 2006.

[9] T. Jim, N. Swamy, and M. Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW*, 2007.

[10] M. Johns. SessionSafe: Implementing XSS immune session handling. In *Proc. ESORICS*, 2006.

[11] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities. In *Proc. IEEE Symp. on Security and Privacy*, 2006.

[12] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross-site scripting attacks. In *ACM Symp. on Applied Computing*, 2006.

[13] B. Livshits and M. S. Lam. Finding security errors in Java programs with static analysis. In *Proc. Usenix Security Symp.*, 2005.

[14] Microsoft ASP.NET AJAX. http://ajax.asp.net.

[15] Y. Minamide. Static approximation of dynamically generated Web pages. In *Proc. WWW*, 2005.

[16] MITRE. Common vulnerabilities and exposures. http://cve.mitre.org/cve/, 2007.

[17] Open Web Application Security Project. The ten most critical Web application security vulnerabilities. http://umn.dl.sourceforge.net/sourceforge/owasp/OWASPTopTen2004.pdf, 2004.

[18] T. Pixley. DOM level 2 events specification. http://www.w3.org/TR/DOM-Level-2-Events, 2000.

[19] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proc. OSDI*, 2006.

[20] RSnake. XSS (Cross Site Scripting) cheat sheet. http://ha.ckers.org/xss.html, 2006.

[21] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[22] Same origin policy. http://en.wikipedia.org/wiki/Same_origin_policy, 2007.

[23] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proc. POPL*, 2006.

[24] The Samy worm. http://namb.la/popular.

[25] Web Mashup. http://www.webmashup.com.

[26] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *POPL*, 2007.

A Mutation-Event Transforms

Here, we presents some details on METs, our proposed new mechanism for flexible client-side enforcement.

Mutation events are defined in the proposed Document Object Model (or *DOM*), level-2, as events caused by any action that modifies the document structure [18]. METs are similar to, but simpler than, these standards proposals. METs are also more expressive since they operate on extended data that include both the standard DOM tree model [18] and the abstract syntax trees (ASTs) of executable scripts. Both mutation events and the ASTs of scripts are abstractions already implemented in Web clients; thus, support for the METs primitive should not require substantial client additions, or changes.

Importantly, METs provide two mutation events for <SCRIPT> nodes: first, an event when the script node is inserted in the DOM, and another event when that inserted node is populated with the AST for its script code. This separation allows METs to enforce security policies that limit network access caused by SRC host URL attributes in script nodes. Other nodes, such as <STYLE>, are also handled in this manner.

The type signature of MET callback functions is given at the start of Figure 3. Both the `script` and `target` are regular DOM nodes in the document tree. The `script` refers to the node containing the script that is attempting the document mutation (e.g., by writing to an `innerHTML` field). The `target` refers to the parent node where `newValue` is about to be inserted to replace `oldValue`. Both `oldValue` and the `newValue` are well-formed, properly nested subtrees of our extended DOM that includes ASTs; either may be `null` to denote empty subtrees. The callback functions return an extended DOM subtree to be used (instead of `newValue`).

MET callback functions may be registered for DOM elements of particular types, e.g., as follows :

```
add_MET_callback(nodeType, policy)
```

In the above, `policy` would be invoked whenever a DOM element of type `nodeType` is affected (i.e. inserted, replaced, or deleted). This would happen at runtime, right before the mutation, but after the Web client has parsed the new, proposed extended DOM values.

All programmatic security policy variables and functions, and MET callback registration, may naturally occur in the first <SCRIPT> tag of Web pages. To prevent subversion of security enforcement, the script language scoping rules (or other means) should prevent access to security policy code and further MET callback registration might be disabled, e.g., by simply setting `document.prototype.add_MET_callback` to `null` in the code. (More flexibly, Web clients might allow other scripts to register MET callback functions, if they carefully ensure security policy always takes precedence.)