Mashup Component Isolation via Server-Side Analysis and Instrumentation

K. Vikram
kvikram@cs.cornell.edu
Department of Computer Science
Cornell University, Ithaca, NY 14853

Michael Steiner
msteiner@us.ibm.com
IBM T.J. Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532

***Abstract*—Web 2.0 and mashups provide opportunities for exciting new applications. However, the security model of the underlying browser technology is quite inadequate to deal with the new trust and security issues. In particular, it provides no good and easy way to isolate mashup components from each other. While browsers might eventually fix these problem, this will take its time. One promising approach which works with current browsers is based on server-side analysis and code instrumentation.**

I. Background

Web 2.0 mashups provide exciting new ways to aggregate information services from multiple providers, and present them to users. However, given that these services stem from different and not necessarily mutually trusting providers, it is clear that such mashups should be built on a sound security foundation protecting the interests of the various involved parties such as the providers and the end-user. For example, in a mashup providing a one-stop car purchase portal combining information from different dealers and the user's bank, neither should dealers be able to modify each others car prices nor should they be able to spy on a user's bank account.

Unfortunately, mechanisms offered by current browsers[1] are rather weak and lack clean ways to isolate different client-side components as well as limit their interaction to tightly controllable channels. In particular, the same-origin policy turns out to be deficient: On the one hand, it is too restrictive as it prevents safe communication between different sites which often results in developers using dynamically inserted `<script>` tags, e.g., JSONP[2], which give the remote side arbitrary control over the page content. On the other hand, the policy is too weak as it provides no separation between components from the same site, even though such information might stem from server-side aggregation combining sources of different trustworthiness such as is seen often in Internet portals and advertisement-sponsored web-pages. Even for situation such as enterprise portals where arguably information comes from the same trust domain, the sensitivity of salary data and alike makes security-in-the-depth a necessity to protect against programming errors such as cross-site-scripting attacks.

[1]In the following, we are focusing on core DHTML technologies such as (X)HTML, CSS, DOM and JavaScript. In particular, we do not address Java Applets and Flash which are either rarely used or are proprietary. Besides, neither of their secure models provide the answer for the problems identified.

[2]http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/

While secure solutions often could be built in principle, the involved subtleties are beyond what we can reasonably expect from the usual mashup developers. What we need are new high-level and fail-safe programming abstraction and corresponding isolation mechanisms for securely separating components of a mash-up. While eventually browsers might offer such abstractions natively based on proposals such as the W3C HTML component model[3] or Doug Crockford's `<module>` tag[4], this will take a while. Therefore, it seems wise to explore alternatives which work with current browsers.

II. Server-Side Analysis and Instrumentation

One promising way to get such a foundation for component separation is based on server-side static analysis and code instrumentation. While code instrumentation has been used in much prior work, e.g., IRM [1], only recently did it get targeted to JavaScript [2]: BrowserShield [3] used JavaScript code interposition to prevent exploitation of browser vulnerabilities whereas Yu et al. [4] explored the theoretical foundations. Independent of above-mentioned work, we were exploring similar techniques focusing on JSR 168 [5] portlets. The security model we want to enforce is isolation of portlets from each other. More specifically, portlets and their associated JavaScript code should be contained to disjoint well-identified DOM subtrees.

JavaScript poses a number of new challenges due to its dynamic nature which allows to modify virtually any code and to evaluate — using a multitude of ways — arbitrary code at runtime. Furthermore, to address the browser environment one also has to incorporate the Document Object Model (DOM)[5], which in turn also adds additional ways for self-modification of code and data. This makes it hard to analyze arbitrary code and to make interposition code tamper-proof.

Our approach to tackle portlet isolation roughly comprises the following steps: (1) For each portlet fragment, we check a number of syntactic constraints[6] and mark each fragment with

[3]http://www.w3.org/TR/NOTE-HTMLComponents

[4]http://json.org/module.html

[5]http://www.w3.org/DOM/

[6]For example, two malicious portlets are not prevented by JSR 186 from wrapping a form element of a third good portlet with another form and hence hijack information submitted from the good portlet, even when JavaScript is disabled. This attack is possible with at least one commercial portal server.

its corresponding security domain by wrapping it in a special `div` element `portlet-root`; (2) After aggregation of the portlet fragments into a whole HTML page, we convert the page into an equivalent JavaScript program, i.e., one which renders the exact same content; (3) Together with an object model of the browser's DOM, also defined in JavaScript, we perform a static analysis of isolation and integrity constraints using a predecessor of IBM Research's WALA[7] libraries; (4) Finally, we rewrite certain code constructs, e.g., to separate name spaces. Converting everything into JavaScript allows for a unified analysis approach. For instance, having converted the HTML into equivalent JavaScript, the analysis engine automatically constructs an object model for the DOM tree for the page, which is used to perform precise alias analysis of DOM objects. Uniformly using JavaScript also enables easy customizations to particular browsers which are usually not 100% standards-conformant and provide various security-sensitive extensions.

Two examples of constraints we perform in step (3) are the restriction of DOM tree walking of a portlet to its domain and the protection of the integrity of system code.

To restrict tree-walking, we perform a pointer analysis on all operations that climb up in the tree — descending is always safe – and make sure that the points-to set does not include the `portlet-root` element. Together with the constraints guaranteed by construction in step (1), the name space separation ensured by step (4), this will guarantee the invariant that a portlet can only access its own DOM elements.

Of course, above algorithm relies also on the integrity of the system libraries which brings us to the second example of analysis. To maintain code integrity, we have to assure that no user code can redefine system code or objects. Furthermore, we have to make sure that system functions only receive objects as parameters which meet the expectation, i.e., the parameter to the method `appendChild` of `DOMNode` must be a proper `DOMNode` generated by `DOMDocument.createElement` or equivalent. This is necessary to prevent a rogue element to subvert the browser "inside-out". To achieve this, an information-flow lattice has to be enforced to prevent user information from flowing into system code. Obviously, given the multiple ways JavaScript allows to alias functions and variables, we have to be careful to do appropriate alias analysis.

III. Challenges

So far we mostly side-stepped the issue of runtime code evaluation: We restrict executed code in event handlers to calls to statically fixed functions and ban `eval` in its various incarnations. `eval` is mostly unnecessary and indicates the presences of bad coding; even for handling of JSON[8] objects, a common use-case for `eval`, a JSON parser instead of `eval` is preferable from a security-in-the-depth perspective. Such a restricted model will require changes to existing code and we might have to relax the model in the future to get acceptability.

Currently, we also do not allow for any collaboration among portlets. This is clearly not sufficient for mashups. While one can imagine extending this approach to implement fine-grained access control on individual DOM resources and stack-introspection, similar to Java, we conjecture that this is much too complex to be usable to ordinary programmers. Our view is that communication should be via declared and typed message passing interfaces and DOM resources (sub-trees) should stay completely isolated. A client-side variant similar to the event mechanisms allowing portlets to communicate on the server-side defined in the upcoming revision of JSR 168 [6] could provide an good programming model also for mashup components. Regarding access policy, RBAC or capabilities seem suitable choices.

Finally, of high importance of course is also performance. Generic code rewriting has a high overhead, e.g., BrowserShield [3] reports a slowdown of 20x-400x. While static analysis and a restricted programming model reduces the complexity of code rewriting, it can increase drastically the computation cost on the server side. It is an interesting research question whether the analysis could be done directly on the generation code, e.g., JSP or PHP. A potential hybrid approach would be for the code generation to generate additional information facilitation the analysis akin to proof carrying code [7]. Basing the original code generation on a more restricting language, e.g, providing certain type-safety as in GWT[9] could help towards that goal. This would also be helpful if the component is generated remotely, e.g., via WSRP [8].

IV. Acknowledgements

We would like to thank Julian Dolby for providing the generic JavaScript static analysis functionality and helping us getting around in the framework.

References

[1] Úlfar Erlingsson and F. B. Schneider, "IRM enforcement of Java stack inspection," in *Proceedings of the IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society, Technical Committee on Security and Privacy. Oakland, CA: IEEE Computer Society Press, May 2000, pp. 246–255.

[2] ECMA, "ECMAScript language specification," Dec. 1999, eCMA Standard 262, 3rd Edition.

[3] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir, "BrowserShield: Vulnerability-driven filtering of dynamic HTML," in *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[4] D. Yu, A. Chander, N. Islam, and I. Serikov, "JavaScript instrumentation for browser security," in *34st Symposium on Principles of Programming Languages (POPL)*. ACM Press, Jan. 2007, pp. 237–249.

[5] A. Abdelnur and S. Hepper, "Java Portlet specification: Version 1.0," Java Community Process, Java Specification Requests 168, Oct. 2003.

[6] S. H. (Editor), "Java Portlet specification: Version 2.0," Java Community Process, Java Specification Requests 286, Feb. 2007, early draft 2, rev. 12.

[7] G. C. Necula, "Proof-carrying code," in *24th Symposium on Principles of Programming Languages (POPL)*. Paris, France: ACM Press, Jan. 1997, pp. 106–119.

[8] A. Kropp, C. Leue, and R. Thompson, "Web Services for Remote Portlets Specification," OASIS, OASIS Standard, Aug. 2003, version 1.0.

[7]http://wala.sourceforge.net/
[8]http://www.json.org/

[9]http://code.google.com/webtoolkit/