Aspectizing JavaScript Security

Florent Marchand de Kerchove, Jacques Noyé, Mario Südholt

Aspectizing JavaScript Security

Florent Marchand de Kerchove Jacques Noyé Mario Südholt

ASCOLA team (Mines Nantes, Inria, LINA)
École des Mines de Nantes, Nantes, France

Abstract. In this position paper we argue that aspects are well-suited to describe and implement a range of strategies to make secure JavaScript-based applications. To this end, we review major categories of approaches to make client-side applications secure and discuss uses of aspects that exist for some of them. We also propose aspect-based techniques for the categories that have not yet been studied. We give examples of applications where aspects are useful as a general means to flexibly express and implement security policies for JavaScript.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.4.6 [*Operating Systems*]: Security and Protection—Access controls, Information flow controls

Keywords Aspect-Oriented Programming, Web Application Security, JavaScript

1. Introduction

As the usage of JavaScript to provide a richer web experience for websites grows, so does the concern about the security guarantees these websites provide. The static web pages of the past have turned into full-fledged *web applications*: collections of HTML pages including CSS and JavaScript files to provide a user experience nearing those usually associated with so-called native software. Consequently, web applications are increasingly complex, hence prone to the design flaws and programming errors found in their native cousins. These defects can range from mere nuisances to critical security vulnerabilities, especially as web applications rely on ever-evolving standards, heterogeneous running platforms, and a dynamic scripting language with reflection capabilities but lacking proper isolation mechanisms.

Since security is a crosscutting concern of many applications, its definition and implementation is often aptly expressed using Aspect-Oriented Programming (AOP). Using AOP, security concerns can be factorized and cleanly separated from the core logic of the application; thus making secure applications easier to write, comprehend, maintain and extend. Furthermore, aspects are well-suited to the dynamic nature of JavaScript, from the analysis of running programs to the dynamic definition of security policies.

Applications of AOP to security are neither novel nor exclusive to JavaScript; several approaches have targeted Java for example (see Toledo et al. [22] and its references). However, the peculiarities of the JavaScript language and the complex ecosystem of web browsers and web standards altogether present a new, and largely unexplored, challenge to AOP.

Our primary goal is the expression of security policies for web application using AOP. Following Groef et al. [9], we consider three large categories of security properties and corresponding techniques for JavaScript-based web applications: fine-grained access control, capability-secure scripting and dynamic information flow. In the first section, we show how aspects and aspects-like techniques are already employed by access control mechanisms. These successes give credence to the relevance of AOP for JavaScript security, and for the other two categories in particular. In the following two sections, we show how they can be tackled using aspects. We then discuss the use of AOP to secure web applications from a higher viewpoint.

2. Fine-grained access control

The arguably most basic level of web application security is access control: authorizing scripts for execution.

At the coarsest level, access control prevents a whole script from executing on the basis of its origin. The same-origin policy [27] and the Content Security Policy [19] header are two examples from this category. The CSP header complements the same-origin policy with a whitelist of allowed origins for web content: images, fonts, styles, and more importantly scripts. While effective against most forms of content injection attacks such as cross-site scripting (XSS), this coarse-grained policy is also inherently heavy-handed.

Indeed, if the content origin is the only criterion for security, neutralized content could actually be harmless, while authorized content could be compromised after the definition of the content policy, and thus could contain potentially dangerous code. To prevent these scenarios from happening, more flexibility is required when including a script in a web application; the application developer could disable only a subset of the functionality provided by an included script, in order to prevent common attack vectors (e.g., by restricting uses of `eval`) or information leaks (e.g., by filtering calls to `XMLHttpRequest`). That way, even scripts from trusted origins would be prevented from using unsafe language constructs or calling privacy-sensitive functions.

Fine-grained access control mechanisms precisely aim to provide flexible security policies to web application developers. On the other hand, due to the dynamic nature of JavaScript, fine-grained access control mechanisms have to rely, partly at least, on the runtime observation and manipulation of executed scripts. This runtime monitoring of an existing script for security purposes is advantageously expressed using AOP [24, 25]. Prior work such as ConScript [15], WebJail [2], Phung et al. [17] and ZAC [20] is a testament to the relevance of AOP in the context of JavaScript access control.

```
around(document.createElement,
       function (c : K, tag : U) {
  let elt : U = uCall(document, c, tag);
  if (elt.nodeName == "IFRAME")
    throw 'err';
  else
    return elt;
});
```

Figure 1. An advice to the function `document.createElement` in ConScript that throws an exception when the resulting element is an IFRAME (taken from [15]).

We can divide these results depending on whether aspects are deployed at the system level, within the browser to alter JavaScript interpretation, or at the application level.

2.1 Aspects in the browser

In ConScript [15], security policies imposed upon included scripts are expressed as aspects. Each aspect can enforce very specific and dynamic security restrictions to the application. For example, in Figure 1, an aspect is defined around calls to the DOM function `document.createElement` in order to prevent the creation of IFRAME elements. Since JavaScript lacks a static type system, such restrictions have to be enforced at runtime; and these are conveniently expressed using aspects.

WebJail [2] also uses aspects for advising native functions. Here however, security policies are not directly expressed as aspects, but as JSON configuration files. WebJail then uses these JSON policies to build the corresponding advices. Here, aspects are not exposed to the developer of the web application, but they are still an important part of the security mechanism.

Both ConScript and WebJail apply aspects to the JavaScript interpreter embedded in the browser. A pointer to an advice is added to the memory representation of each JavaScript function, native or created by user scripts. Implementing aspects at this level in the browser has strong benefits. Firstly, the runtime performance penalty is low (between 1 and 7% in this case) when compared to rewriting techniques (30% and up). Secondly, securing policy advices is easier, since they can be separated from potentially malicious user scripts. Nevertheless, the main drawback of this implementation scheme is its specificity: ConScript targets only Internet Explorer 8, and using it requires a modified version of that browser. The same can be said for WebJail, which is specific to Mozilla Firefox 4.

2.2 Aspects in the application

Instead of implementing aspects at the browser level, an alternative is to provide the aspect weaver as a separate script. Lightweight self-protecting JavaScript (LSP) [17] and ZAC [20] are two such approaches: written in JavaScript and executed as any included script. Consequently, they are portable; any browser capable of executing JavaScript can use them.

Like ConScript, LSP and ZAC express security policies as aspects. LSP aims to be a simple lightweight reference monitor, whereas ZAC is built upon AspectScript [21], a library that rewrites JavaScript code to allow aspects to capture all operations (function calls, property reads and writes, assignments, etc.). Rewriting scripts incurs a strong performance penalty, because all the code is wrapped-up for monitoring. The benefit of rewriting is that AspectScript provides ZAC with a finer granularity for security policies than any of the previously presented methods.

```
ZAC.R_EVAL = {
  rule: function (event) {
    return event.isCall() && event.fun === eval;
  },
  action: function (event) {
    try {
      return JSON.parse(event.args[0]);
    }
    catch (e) {
      throw "Use eval only to parse JSON objects";
    }
}};
```

Figure 2. A security policy written as an aspect in ZAC. Calls to `eval` are allowed only when the argument is a string serializing a JSON object (taken from [20]).

3. Capabilities-based security

Capabilities provide a quite different security model than mechanisms based on access control and information flow. Capabilities are a language-level mechanism that cannot be forged and directly serves as a proxy for resource accesses. Capabilities, for example objects as capabilities in OO languages, enable certain security properties, such as isolation properties, to be implemented easily and naturally in programming language terms.

However, most existing OO languages do not provide capabilities themselves, in particular, because of the multiple means they provide for access to many resources. Restrictions on those languages are often used to define *capability-safe* subsets and thus provide a capability-based programming model that is similar to that of mainstream languages, In the following we first provide an overview of (categories of) the restrictions that are necessary to provide capability-safe subsets for JavaScript and Java, briefly introduce capability-based enforcement of security properties, and, finally, show that aspects can be employed to define capabilities and corresponding security properties.

3.1 Capability-safe JavaScript and Java

Objects in standard OO languages do not constitute capabilities per se. Based on analyses for the case of Java (the Joe-E system [14]) the following underlying categories of issues can be distinguished, see Fig. 3: forging of references has to be avoided or arbitrary objects can be turned into capabilities; capabilities must not be leaked or arbitrary users may obtain capabilities; every access has to be mediated through capabilities and capabilities must be strongly encapsulated; otherwise access may be obtained without sufficient authority.

JavaScript (as analyzed for the standard ECMAScript 3 as part of the Caja system [16]), for example, is subject to all of these issues: the global environment may be accessed in many different ways without passing through a representing object; state can only be encapsulated using a limited notion of lexical scope and implicitly mutable state can be used to break encapsulation and propagate references in unexpected ways; references may also change due to the dynamic contexts in which functions are called due to the complex semantic rules governing `this`. All of these features of JavaScript, among others, break capability safety.

Figure 4 shows three concrete examples of capability violations in JavaScript. The first illustrates a facet of the semantics of `this`: in two very similar contexts, `this` is interpreted very differently. In the second line, `this` denotes a function but in the third line `this` allows direct access to the global scope of the JavaScript program. The next two examples illustrate cases from popular JavaScript-based systems, AdSafe and (now retired) Facebook JavaScript, in

Issue	JavaScript (examples)	Java (examples)
Forge references	Implicit mutable state, `this` semantics	Native methods
Prop. refs unexpectedly	Global env., unconstrained properties	Mutable exception throwables
Access res. w/o mediation	Global env., implicitly mutable state	Subclassable static fields, lib. func.
Break encapsulation	Implicit mutable state, limited lexical scope	Reflection API, catch Errors

Figure 3. Capability-breaking language issues.

```
// 1) Thorny semantics of ‘this‘
(x.m)(...)          // x.m function: this bound to x
(true && x.m)(...) // this bound to global scope

// 2) Example from the AdSafe library:
// ephemeral provides access to global object;
// enables alerts
var a = dom.tag("div").ephemeral;
var asd = a().alert("Hacked!");

// 3) example from the FBJS library
// Storing a DOM element yields access to
// foreign-owned documents
var a = dom.text(["hacked"]);
a[0].ownerDocument.location="http://attacker.com";
```

Figure 4. JavaScript capability violations (taken from [13, 16]).

which a reference to specific library objects requiring low authority allow access to completely different objects that should require high authority.

Enforcing security properties using capabilities. A capability-safe language allows several security properties to be enforced in a direct manner: isolation of resource access, for example, can be enforced by assigning disjoint sets of capabilities to different principals. Maffeis et al. [13] define the notion of authority safety as ensuring that an access to a resource is either granted to a principal initially or explicitly handed to that principal by another authorized one. More generally, capabilities support patterns for secure programming: by encapsulating and providing interfaces to sets of capabilities, access to resources can be flexibly managed.

3.2 Aspect-based security with capabilities

In the context of capability-based security, aspects can be used for two purposes:

i. to define capabilities, i.e., ensuring capability-safety in a language that is not safe by itself; and

ii. to ensure security properties on top of capabilities provided by a capability-safe language.

Defining capabilities. Aspects can be used to enforce capability-defining restrictions on the semantics of Java and JavaScript. For example, the JavaScript subset of Caja [16] is defined in terms of static transformations and runtime checks. Many of the static transformations do not correspond, however, to the static part of aspect systems, in particular, that of AspectJ. Most of the restrictions thus have to be implemented using aspect-based runtime checks. The JavaScript problems shown in Fig. 3 can be handled directly: accesses to the global environment and mutable state can be guarded using aspects; the different semantics of `this` can be identified and handled according to context; the scoping rules of JavaScript are essentially dynamic, and can also be restricted using aspects.

In order to make this discussion more concrete, reconsider the capability violations in Fig. 4. All three violations can be handled using aspects: concerning violation 1, the expression in line 3 can be blocked or the usage of `this` monitored later on in order to avoid access to global objects through `this`. In the case of the two latter examples, aspects can easily be used to suppress the leakage of high authority references through the names `dom.tag[].ephemeral` and `dom.text`.

```
x = false;
y = false;
if (document.cookie == "abc") {
  x = true;
} else {
  y = true;
}
if (!x) {
// leaked info: cookie != "abc"
}
if (!y) {
// leaked info: cookie == "abc"
}
```

Figure 5. Attack using implicit flows (taken from [23]).

Enforcing security. Aspects are also well-suited to support security properties that are defined based on capabilities. Authority safety [13] has been implemented for a subset of JavaScript using two rewrite rules, one for property accesses `e1[e2]` and one for `this` [13], and three initialization rules; all of these rules can be implemented simply using aspects. Finally, the enforcement of programming patterns is a standard application of aspects and can be applied straightforwardly to the patterns for secure programming mentioned above.

4. Information Flow

As a third approach to security, *information flow* mechanisms track the flow of information through an application and, basically, guarantee that no confidential information leaks and prevent the injection of untrusted data. Enforcing these properties with static analysis has been extensively studied [18]. Static analysis has the advantage that it makes it possible to reason about *non-interference*, a property stating that confidential data has no influence on public outputs. Dynamic analysis is weaker in that it can only reason on the current execution. Typically, it makes it easy to track *explicit* data flows, which occur via assignments. However, in that case, non-interference is not guaranteed. For instance, in Fig. 5, it is possible to track that the document cookie is confidential and therefore that, depending on the branch taken by the execution, either `x` or `y` carries confidential information. But the other variable remains public and can be used to leak information. *Implicit* data flows, due to conditionals, can still be taken into account, and non-interference guaranteed, by conservatively blocking the execution when a public variable is updated in a confidential context [3]. In our example, this policy would block the execution on updating `x` or `y` within the conditional testing the document cookie. Finally, information flow, including non-interference, can be tracked using

secure multi-execution (SME) [8]. This consists of running multiple copies of the application, one at each confidentiality level, with rules for input and output operations in order to coordinate the executions.

4.1 JavaScript Information Flow

Due to the highly dynamic nature of JavaScript, resorting only to static analysis is hardly feasible. However, practical proposals dealing with some form of non-interference, such as [23] and [7], are hybrid in order to deal with implicit flows. In Staged Information Flow [7], static analysis is performed prior to program execution based on a security policy expressed as sets of pairs $(x, \bullet)$ or $(\bullet, x)$ meaning that the value of the variable x should not flow to or from a "hole" (a dynamically loaded script). When the analysis cannot conclude, it returns a simple residual policy, expressed as a set of variables that should not be read or written by the hole, a purely *syntactic* check that is then performed at runtime. On the opposite, the analyses of [23] are performed at runtime and the system is driven by the dynamic analysis. Forgetting about non-interference, BFlow is purely dynamic [26]. It tracks explicit information flows between *protection zones* (sets of frames) with the help of a reference monitor implemented as a Firefox plugin. BFlow is however limited by its coarse-grained nature: a single untrusted browser frame cannot at the same time handle confidential and public data. All these proposals consider either a full version of JavaScript, e.g., including `eval`, or are very close to it. They are all able to capture significant instances of problematic cross-site scripting.

SME has been, from the beginning, studied in the context of JavaScript. FlowFox [10], for example, does not multi-execute the web browser as, in that case, enforcement would take place at the OS API level, which turns out to be too coarse-grained. It rather multi-executes the scripts with enforcement taking place at the browser API level. Interactions with the DOM, considered as I/O, can thus be tracked.

Finally, Austin and Flanagan have improved on their initial work on guaranteeing non-interference with dynamic analysis [4, 5]. In particular, [5] makes the link between dynamic analysis and SME through the notion of *faceted values*. Instead of equipping each value with a confidentiality label, each value has a facet value corresponding to a confidentiality level, for instance, a private value and a public value if there are only two levels. This makes it possible to simulate SME within a single execution.

4.2 Aspectizing JavaScript Information Flow

In principle, aspects are well-suited to implement dynamic analyses either by applying aspects to the program to be analyzed or by applying aspects to the corresponding language interpreter. AOP has, for instance, been applied with some success to profiling [6] and runtime verification [1]. However, different domains have different needs, in particular in terms of the join-point model and, unfortunately, tracking information flows requires to operate at a granularity level not covered by our off-the-shelf weaver for JavaScript, AspectScript. For instance, tracking explicit flows requires to capture assignments but also arithmetic and logical operations. AspectScript captures the former but not the latter. Typically, the result of an arithmetic or logical operation is confidential if one of its arguments is confidential. In this case, capturing read operations is enough but the weaver must be instructed to do so. Alternatively, each operation can be turned into a function – an overkill if this has to be done systematically. We have also seen above that reasoning on conditional statements was necessary whereas there are no join points attached to conditional statements. This situation is not essentially different from the one of AspectJ, which has led to the introduction of new join points such as loop join points [11].

Another interesting issue is the interplay between static and dynamic analysis and modifying the browser versus modifying the application. Modifying an actual browser for tracking information flows is often described as a challenging task that requires to "instrument" various parts of the browser. This suggests that it makes sense to study the design and implementation of browsers using aspects but also that wiring this facility within the browser makes it difficult to compile it away when static analysis allows it. This issue does not exist when information flow tracking is directly woven into scripts. Static analysis can then be used to control weaving or even been considered, at least conceptually, as a result of partially evaluating weaving [12].

5. Discussion

Throughout this study, we have shown useful applications of aspects for securing JavaScript-based web applications. There are some interesting uses of aspects for access control but they have not been considered yet within the two other categories. Furthermore, a comprehensive strategy for secure software systems needs to consider different security properties. Techniques to support combinations of security properties of different categories, for example for access control and information flow, are therefore of interest but have not been studied at all until now.

Aspects are well-suited to provide a general framework for the declarative expression and efficient implementation of flexible security policies for JavaScript. Specifically, we envision

i. a high-level, possibly aspect-based, domain-specific language for the declarative definition of security policies and properties;

ii. flexible implementation and tool support for dynamic analysis and rewriting that may advise, for instance, constructs at runtime in a scope of any depth;

iii. explicit support for interactions between aspects to resolve issues of precedence and specificity among security aspects.

Our study suggests that security is an interesting testbed for (and probably strongly benefits from) advanced mechanisms in AOP such as history-based pointcuts, execution levels, and scoping strategies.

Performance is also an important issue when considering practical applications. To mitigate the costs of rewriting, we can sacrifice portability and alter the JavaScript interpreter instead. Another gain could be achieved by increasing the portion of static computation, at the cost of runtime flexibility; hot-swapping of security policies, for example, can be harnessed.

Eventually, there is the looming matter of validating aspect-based security policies: ensuring they are effective and without unintended side-effects. Existing formal frameworks for JavaScript security should thus be integrated with aspect formalisms.

6. Conclusion

To summarize, we reviewed major categories of approaches to secure JavaScript-based client-side applications. We first discussed uses of aspects (and their limitations) that exist for some of them. Then, we outlined the use of aspects for categories where aspects have not yet been studied. We provided concrete examples of the usefulness of aspects as a general means to flexibly express and implement security policies for JavaScript.

Acknowledgments

This work has been partially funded by the SecCloud project of the French "Laboratoire d'Excellence" CominLabs. The authors would also like to thank the reviewers for their helpful comments.

References

[1] M. Achenbach and K. Ostermann. A meta-aspect protocol for developing dynamic analyses. In *Proceedings of the First International Conference on Runtime Verification (RV)*, number 6418 in Lecture Notes in Computer Science, pages 153–167, Nov. 2010.

[2] S. V. Acker, P. D. Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: least-privilege integration of third-party components in web mashups. In *27th Annual Computer Security Applications Conference (ACSAC)*, 2011.

[3] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the 4th Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, 2009.

[4] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, 2010.

[5] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL)*. ACM, 2012.

[6] W. Binder, D. Ansaloni, A. Villazón, and P. Moret. Flexible and efficient profiling with aspect-oriented programming. *Concurrency and Computation: Practice and Experience*, 23:1749–1773, 2011.

[7] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proceedings of the 2009 Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[8] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *31st IEEE Symposium on Security and Privacy (S&P)*, 2010.

[9] W. D. Groef, D. Devriese, and F. Piessens. Better security and privacy for web browsers: A survey of techniques, and a new implementation. In *Formal Aspects of Security and Trust - 8th International Workshop (FAST)*, 2011.

[10] W. D. Groef, D. Devriese, N. Nikiforakis, and F. Piessens. FlowFox: a web browser with flexible and precise information flow control. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[11] B. Harbulot and J. R. Gurd. A join point for loops in AspectJ. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 63–74. ACM, Mar. 2006.

[12] K. Klose, K. Ostermann, and M. Leuschel. Partial evaluation of pointcuts. In *Proceedings of the 9th international conference on Practical Aspects of Declarative Languages (PADL)*, pages 320–334. Springer-Verlag, 2007.

[13] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *31st IEEE Symposium on Security and Privacy (S&P)*, 2010.

[14] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[15] L. A. Meyerovich and V. B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *31st IEEE Symposium on Security and Privacy (S&P)*, 2010.

[16] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. *Caja: Safe active content in sanitized JavaScript*. Google, June 2008.

[17] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting JavaScript. In *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.

[18] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[19] B. Sterne and A. Barth. *Content Security Policy 1.0*. W3C, Nov. 2012. `http://www.w3.org/TR/CSP/`.

[20] R. Toledo and É. Tanter. Access control in JavaScript. *IEEE Software*, 28(5):76–84, 2011.

[21] R. Toledo, P. Leger, and É. Tanter. AspectScript: expressive aspects for the web. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD)*. ACM, 2010.

[22] R. Toledo, A. Núñez, É. Tanter, and J. Noyé. Aspectizing Java access control. *IEEE Transactions on Software Engineering*, 38(1):101–117, 2012.

[23] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2007.

[24] B. D. Win, F. Piessens, and W. Joosen. How secure is AOP and what can we do about it? In *Proceedings of the 2006 international workshop on Software engineering for secure systems (SESS)*. ACM.

[25] B. D. Win, B. Vanhaute, and B. D. Decker. How aspect-oriented programming can help to build secure software. *Informatica*, 26(2), 2002.

[26] A. Yip, N. Narula, M. N. Krohn, and R. Morris. Privacy-preserving browser-side scripting with BFlow. In *Proceedings of the 2009 EuroSys Conference (EuroSys)*. ACM.

[27] M. Zalewski. *Browser Security Handbook, part 2*. Google, 2008–2011. `https://code.google.com/p/browsersec/wiki/Part2`.