ZAC: Access Control in JavaScript

Rodolfo Toledo Éric Tanter
PLEIAD Lab
Computer Science Department (DCC)
University of Chile
Blanco Encalada 2120
Santiago, Chile
{rtoledo,etanter}@dcc.uchile.cl

Abstract

ZAC is a practical lightweight library for access control in JavaScript based on aspect orientation. The ZAC access control architecture is stack-based, very similar to the ones of Java and C#. However, ZAC integrates other interesting features for more expressive access control. First, access control policies can be enforced at the level of objects, which permits more fine-grained control over the access to resources. Second, policies in ZAC can base their decisions on the execution history of scripts, which permits to express policies that are impossible to define using other models, such as bounded-time execution.

Index Terms

D.3.3 Language Constructs and Features, D.3.2 Scripting languages, D.3.1 Semantics

I. Introduction

Third-party JavaScript code inclusion is a cornerstone of today's Web-2.0 applications. Without it, rich personalized start pages, like iGoogle and My Yahoo, would be unthinkable. Facebook would not be as rich as it is without external applications. Even targeted advertising, like Google AdSense, would be impossible. However, despite being very useful, including external code poses a threat to the hosting page: JavaScript can be used to modify both the layout and the functionality of a page to a potentially arbitrary degree.

The questions are therefore: What are the rights of external code? Is it allowed to arbitrarily modify the host page layout or functionality? How can it be restricted to only certain actions? Can we make sure external code does not degrade the interactive user experience with the page? Other languages that also support dynamic loading of code, like Java and C#, have a powerful access control architecture, allowing developers to expressively specify the rights of code. While the most recent specification of JavaScript includes some basic access control features [10], external tools must be used for more advanced control over the access to sensitive resources; these tools rely on a variety of approaches, ranging from code analyses, code transformation, and runtime libraries.

Current Approaches for Access Control in JavaScript

The only way provided by the HTML standard to control the rights of external code is to use frames: including a script inside a frame gives full permissions to the script, but only over its enclosing frame. However, in the majority of cases, this option is not expressive enough, mainly because it is too coarse-grained. For instance, it is impossible to grant certain permissions to external code, like showing alert dialogs, while forbidding others, like accessing cookies.

For this reason, several proposals have been presented for controlling the actions that scripts can perform within a web page (Figure 1). AdSafe [1] limits the JavaScript features external code can use, leaving only a "secure" subset of the language that can be statically checked to ensure it does not perform potentially dangerous actions. FBJS [6] rewrites the code to replace references to standard objects with limited, but

Name	Enforcement	Based on	Extensible/Specification	Granularity
AdSafe	static	static analysis	no/-	script
FBJS	static + dynamic	object wrappers	no/-	page
BrowserShield	dynamic	program monitoring	yes/blacklisting	page
Caja	static + dynamic	object capabilities	yes/whitelisting	script
ZAC	dynamic	dynamic AOP	yes/blacklisting	object

Fig. 1. JavaScript proposals for access control.

equivalent objects, preventing any potentially dangerous action. This is what Facebook uses for third-party applications. BrowserShield [13] is a more flexible proposal that can be used in more scenarios. BrowserShield transforms code to make it trigger notifications of its own activity during execution. An observer entity then decides whether the activity of the code should be allowed or not. New policies can be specified as functions using the JavaScript language itself. Finally, Caja [2] is a more principled approach to access control based on the object-capability model. In this model, external code can access (and therefore use) references to other objects only if the host page provides them. If no reference is provided, the external code can still compute based on its own (harmless) references. Caja is used in iGoogle and My Yahoo.

As depicted in Figure 1, ZAC combines very interesting features:

- **Dynamic enforcement of policies** enables the execution of programs that are safe but use unsafe constructs. For example, programs that use eval only to deserialize JSON objects from strings, and not to execute arbitrary code, are safe. However, because they use a potentially unsafe construct, such programs are rejected by systems like AdSafe.
- Being based on **dynamic aspect-oriented programming (AOP)** [5], allows ZAC policies to reason about program execution in its entirety. For instance, it is possible to define a policy that prevents a script from never ending, or taking too much time for acceptable web interaction. This kind of policy is impossible to express in the object-capability model because the property does not depend on object references, but rather on computation itself.
- **Extensible access control specifications** is crucial considering that different usage scenarios imply different requirements. This is acknowledged by other proposals like BrowserShield and Caja.
- **Object-level granularity** is a unique feature of ZAC. Allowing the coexistence of different policies for different scripts within a Web page is fundamental. Going beyond the script level down to the object level also enables a secure interaction among scripts: objects from one script can use objects from other scripts, possibly with different policies; the correct policy will be unequivocally enforced. This is not the case with capabilities, where an untrusted object that obtains a reference to a sensitive resource can use it without limitations.

The principle of ZAC is that foreign code can use every feature of JavaScript, including eval, and also can access every reference to any object in the system. However, the access control policy assigned to the foreign code when loaded forbids dangerous actions before they happen at runtime. In other words, ZAC follows a blacklisting approach, which, despite being considered less safe than a whitelisting approach, is actually used in real systems for access control. Even more, it turns out to be equivalent to the access control architectures of widely-used languages such as Java [8] and C# [9] (see Figure 2 for more details). Finally, ZAC also inherits the great expressive power of the underlying general-purpose library for aspect-oriented programming AspectScript [15], as will be shown later on in this article.

II. ZAC in Action

ZAC is based on assigning access control policies to scripts when loaded. At runtime, the policy is enforced for every action performed by these scripts.

Blacklisting v/s Whitelisting in Practice

Policies based on whitelisting specify what resources the entities in the system *can* access (*e.g.* user X can use the printer). Conversely, policies based on blacklisting specifies what resources *cannot* be accessed (*e.g.* user Y cannot modify system files). In general, whitelisting is considered a safer approach because access to resources can be granted gradually, minimizing the risk of inadvertently granting access to unneeded sensitive resources. This is not the case of blacklisting where it is possible to forget to restrict access to a resource.

However, and despite the fact that whitelisting is superior from a conceptual point of view, blacklisting-equivalent approaches are successfully use in practice. A compelling proof of this comes from the widespread use of Java and C#. While their access control architectures appear to be based on whitelisting (because one declares permissions, not restrictions), they are in practice equivalent to blacklisting approaches. This stems from the fact that permission checking in these architectures has to be *explicitly* triggered at each and every relevant place in the code (using SecurityManager.checkPermission(<Permission>) in Java and <IPermission>.Demand() in C#). This means that forgetting to add the permission check associated to a sensitive resource in Java/C# is just like forgetting to restrict the access to that resource in ZAC. The dependency on explicit checks implies that the set of permissions is known in advance, and therefore, the set of restrictions can be calculated as the complement of the permissions. This is what makes the architectures of Java and C# equivalent, in practice, to the blacklisting architecture of ZAC.

Fig. 2.

A. Loading Scripts

Enforcing access control policies with ZAC is very easy through the use of a simple API. For example, to load a third-party script it is only necessary to use the ZAC.load method:

```
ZAC.load("http://www.evilsite.net/evil.js", ZAC.newDefaultPolicy());
```

Loading the evil.js script using load, its execution is automatically subject to the restrictions in the policy specified as the second argument. Policies in ZAC are sets of restrictions. Figure 3 shows the restrictions in the policy returned by ZAC.newDefaultPolicy(), targeted to restrict the access to common sensitive resources. For instance, the ZAC.R_ALERT restriction forbids the use of alert dialogs. These dialogs are normally used to provide valuable information to the user, but they can also be used to turn a page (or even the whole browser) unusable by endlessly showing an alert dialog. Another example is ZAC.R_LOCATION, that forbids redirections of the page. This restriction prevents malicious scripts from sending the browser to potentially dangerous sites.

B. Policy Enforcement

When loaded using ZAC.load, a script is unable to bypass the specified access control policy, directly or indirectly. This means that the script itself will not be able to perform any action forbidden by the policy, and also that it will not be able to lead other (possibly trusted) code to do it on its behalf. The reason is that ZAC supports *stack-based access control* [7] similar to Java [8] and C# [9]. The semantics of stack-based access control says that each time a sensitive action is about to be performed, the current stack of evaluation is inspected to determine whether all the participating entities (in the case of JavaScript, objects and functions) are allowed to perform the action. If one entity is not allowed, then the action is aborted, typically by throwing an exception. In order to diminish the performance overhead, ZAC does not inspect the stack every time a sensitive action occurs, but maintains the security state of the application at each necessary point (see Figure 4 for more details).

Figure 5 shows three different attempts to call the alert function, all ending with an exception raised by the access control policy. The first one is a direct call, the second one uses delegation: the untrusted code

Constant (ZAC.)	Description
R_ALERT	prevents alert calls.
R_LOCATION	prevents redirection of the browser.
R_C_STYLES	prevents calls to computedStyles().
R_INNER_HTML	sanitizes strings assigned to the innerHTML property.
R_COOKIES	prevents access to cookies.
R_GLOBAL	prevents access to properties of the global object.
R_EVAL	prevents arbitrary use of eval (only JSON deserialization).
R_FUN	prevents instantiations of Function objects.
R_STO_SI	prevents calls to setTimeout and setInterval with a string argument.
R_HTTP_REQ	prevents instantiations of XMLHttpRequest objects.
R_DEF_PROTOS	prevents modification of prototypes of default objects.
R_ARGS	prevents access to the arguments property of other functions.
R_WATCH	prevents calls to watch and unwatch.
R_UNENCR	prevents calls to toSource and uneval.
R_ZAC_POLICIES	prevents access to ZAC policies.

Fig. 3. Restrictions in ZAC's default access control policy. Constants are accessed as properties of the ZAC global property.

ZAC Performance overhead

The main sources of performance overhead in ZAC are twofold: the overhead of event generation, and the overhead of the stack-inspection semantics for access control. We refer the reader to [15] for an overview of the optimization techniques for event generation.

In order to diminish the performance overhead of access control, ZAC does not inspect the stack every time a sensitive action occurs. Instead, it uses a state-based approach in which the security state of the application is kept up to date. This approach is similar to the optimization for control flow pointcuts commonly used in the implementation of aspect-oriented languages [12], where determining whether an action is in the control flow of another is a matter of testing a variable, avoiding the traversal of the stack. The technique used in ZAC can also be compared to security-passing style [16], in which an extra parameter representing the current security context is passed to all functions. As future work, other implementation approaches can be integrated, such as using continuation marks [4]. This permits to get rid of the dependency on the stack to reason about access control (maintaining the same semantics), and also enables very interesting optimizations [3].

Fig. 4.

invokes the (trusted) info function, which in turn tries to call alert. All these attempts end with an exception because the stack contains an object whose policy does not permits calls to alert. The third attempt is interesting, because it uses eval. ZAC ensures that restrictions of the code that calls eval are inherited by the eval-ed code: therefore, this attempt also fails. The last attempt is more intricate: although ZAC's policies are specified at the level of scripts (loading them using load), they are enforced at the level of individual objects. The consequence is that if an object is created during the execution of a script subject to a certain policy, that object's execution will always be subject to that policy: wherever the object goes, the policy follows it. Therefore, in the example, the policy is present in the stack when the anonymous function calls info. This is the reason why the third attempt also ends with an exception. It is important to highlight that this kind of access control is impossible to achieve with any other proposal (Figure 1). For example, in capability-based access control, if the anonymous function manages to get a reference to the info function, there are no means to prevent it from calling info to display an alert dialog. Once an object obtains a reference, it can use that reference at will, no matter which was the policy originally assigned

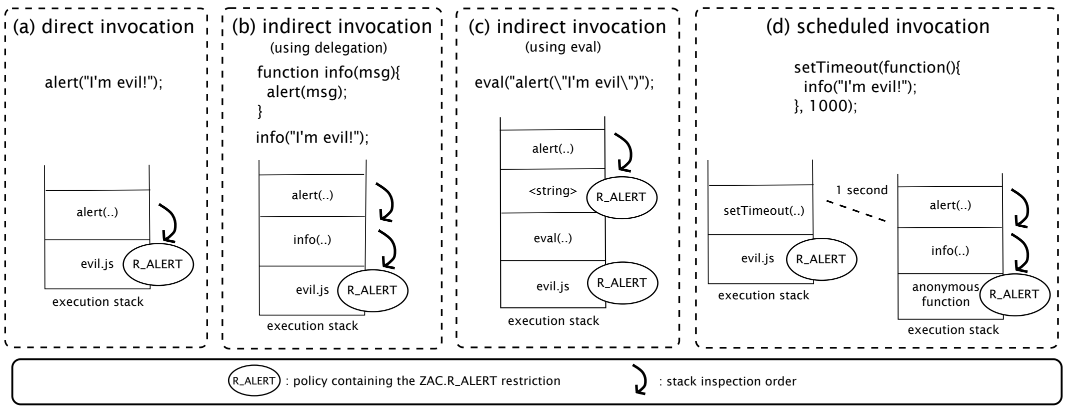

Fig. 5. Policy enforcement in ZAC. All these attempts will fail with an exception raised by the access control policy.

to it.

C. Defining Custom Policies

As typical sets, policies in ZAC can be modified by adding or removing restrictions (using the add and remove methods respectively). There are several scenarios where this can be useful. For instance, it seems natural to specify different policies for different foreign scripts, depending on the level of confidence the host page has in each of them. Another scenario is when a policy must be constructed programmatically (*e.g.* according to the preferences of the user). The API of ZAC permit to modify policies even *after* using them to load a script. These modifications will affect all the scripts already loaded using the policy.

In addition to default policies, empty policies can be created using ZAC.newPolicy(). In the following piece of code, two policies are configured to enforce different restrictions in two different scripts:

```
var softPolicy = ZAC.newPolicy(); //fresh policy (no restrictions)
//restrict only page redirections and evaluation of code in notSoEvil.js
softPolicy.add(ZAC.R_LOCATION, ZAC.R_EVAL);
ZAC.load("http://evilsite.net/notSoEvil.js", softPolicy);
//get a default policy and remove only the R_ALERT restriction for evil.js
var hardPolicy = ZAC.newDefaultPolicy();
hardPolicy.remove(ZAC.R_ALERT);
ZAC.load("http://evilsite.net/evil.js", hardPolicy);
```

III. Extending ZAC

Each restriction in a policy is targeted to restrict the access to common sensitive resources. ZAC's policies can be extended by adding new restrictions targeted to protect other resources.

A. Defining Restrictions

A restriction is a JavaScript object with two properties, both of which are functions. A rule property is in charge of identifying the access to the resource, and an action property is in charge of specifying the action to take when the resource access occurs. For example, the ZAC.R_ALERT restriction is implemented as follows:

```
ZAC.R_ALERT = {
 rule:   function(event){ return event.isCall() && event.fun === alert; },
 action: function(event){ throw "Cannot call alert"; }
};
```

Event Name	Properties	Common properties	Common methods
New	fun, args	parent	proceed(args), clone(), is<eventName>()
Init	target, fun, args		
Call	target, fun, args, context, reflective		
Exec	target, fun, args		
PropRead	target, name		
PropWrite	target, name, value		

Fig. 6. Properties and methods of events supported by ZAC. `fun`: the function being used as a constructor (`New` and `Init`), or being called/executed (`Call` and `Exec`). `args`: the arguments of the event. `target`: the target of the event. `context`: the object performing the call. `reflective`: whether the call was performed using `call` or `apply`. `parent`: the parent event (like a stack of execution). `proceed`: executes the event. `clone`: clones the event (useful to store a reference). `is<eventName>()`: boolean-returning utility methods to identify the kind of event.

The function bound to the rule property identifies calls to the alert function by returning true when such calls occur. It uses the event parameter, which is a representation of the event occurring in the script. Figure 6 shows the complete list of event types supported by ZAC, and the corresponding properties (fields and methods) in each case. The function bound to the action property simply throws an exception because invoking alert is completely forbidden.

An alternative to simply throwing an exception is to provide, possibly under certain circumstances, an alternate "safe" behavior. For example, the eval function of JavaScript is widely considered dangerous because it permits to execute arbitrary, potentially malicious, code. However, eval has a very useful application: the deserialization of a JSON [11] object from a string. Because JSON object's serialization format do not permit functions, no arbitrary code can be executed when evaluating a serialized object. In other words, using eval for deserializing JSON objects is safe. The code below shows the ZAC.R_EVAL restriction that only forbids the evaluation of code that is not in JSON format:

```
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 "Eval can only be used to deserialize JSON objects.";
   }
 } };
```

Just like in the ZAC.R_ALERT restriction, the rule property identifies calls to a certain function, in this case, eval. The action property, instead of immediately throwing an exception, first tries to evaluate the first argument of eval (event.args[0]) as a JSON string. If it effectively is a JSON string, the resulting object is returned. Otherwise, an exception is thrown by JSON.parse; the restriction action then throws an exception itself.

B. Adding New Restrictions

ZAC comes with a set of predefined restrictions (recall Figure 3), which corresponds to common cases. It is possible to define whole new kinds of restrictions as well. Let us define a new restriction that limits the number of windows a script can open:

```
function nWindowsRestriction(n){
  return {
   nWindows: 0,
    rule:    function(event){ return event.isCall() && event.fun === document.open; },
    action: function(event){
      if(++this.nWindows > n){
        throw "Cannot open more than " + n + " windows.";
      }
      return event.proceed();
    }
  }; }
//add the restriction to a fresh policy
var policy = ZAC.newPolicy().add(nWindowsRestriction(3));
```

The code above shows the use of three interesting elements: a stateful restriction, a restriction factory, and the use of event.proceed().

Stateful restrictions. A restriction can have any number of additional properties apart from rule and action. In the example, the nWindows property is used to keep track of the number of windows opened by the script. Therefore the restriction returned by nWindowsRestriction is called a *stateful* restriction. The only consequence associated to a stateful restriction is that when it is added to more than one policy, its state is shared among these policies. For this reason, a restriction like the one returned by nWindowsRestriction will allow three windows in total, summing up all the windows opened by all scripts the restriction applies to.

Restriction factory. To define a restriction, creating an object with the appropriate properties suffices. However, using a restriction factory like the nWindowsRestriction function has two advantages. First, it permits to easily parameterize the restriction (the n argument in the example). And second, it permits to obtain a different instance each time the generator is invoked. This can be used to avoid the sharing issue associated to stateful restrictions.

The proceed method. The proceed method can be used to execute the original behavior the event parameter represents in the script. In the example, the *call* to the open function. The proceed method accepts the same parameters the original event does. If specified, these parameters replace the original parameters passed to the event. If omitted, the event is executed with the original parameters.

IV. Advanced Features

A. Privileged Execution

In some cases, a piece of code needs to perform a sensitive action on behalf of another piece of code. This could be the case of the info function presented before: it may allow any entity to display an alert dialog, regardless of its access control policy. However, the stack-based access control mechanism presented before does not permit that. For this reason, languages like Java [8] provide a way to relax this constraint by executing a *privileged action*: during the execution of a privileged action, the restrictions of the objects in the control flow *before* the privileged action are omitted. Only the ones pertaining to objects up to the one initiating the privileged action are considered. In terms of the stack inspection mechanism, privileged execution is analogous to stop looking in the stack of execution when the frame that initiated the privileged action is reached. It is important to notice that the privileged execution mechanism does not circumvent access control: any object can start a privileged execution, but doing so, it cannot get rid of its own restrictions, because they are maintained. In other words, a very restricted object can start a privileged execution, however, during the privileged action, the restrictions of the object are present.

In ZAC, a privileged action is started by a self call to doPrivileged, and the code to execute in this privileged context is specified as the body of the function passed as parameter. Therefore, the info function can be implemented this way:

```
function info(msg){
  this.doPrivileged(function(){
    alert(msg);
  }); }
```

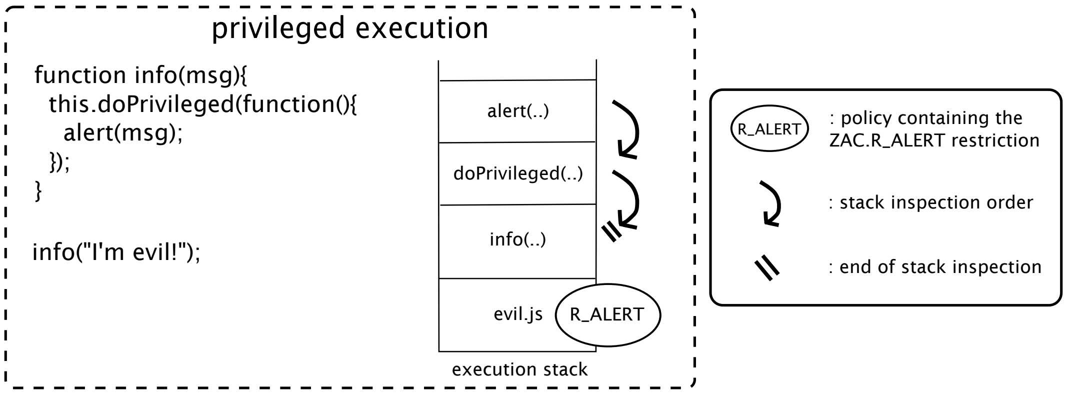

Fig. 7. Stack inspection in presence of privileged execution.

Figure 7 depicts the stack-based access control process in presence of privileged execution: when the alert function is about to be called, the stack is inspected up to the frame that initiated the privileged action, thus the frame corresponding to the object that has the ZAC.R_ALERT restriction is not reached.

doPrivileged acts just as a marker method signaling a privileged execution. Objects can use their own implementations of doPrivileged, the only requirement is to maintain the name. A default implementation of this method is to simply invoke the function argument:

```
obj.doPrivileged = function(action){
  action();
};
```

The semantics of ZAC for privileged execution defines that any self call to doPrivileged starts a privileged action. Non self calls to doPrivileged are not considered privileged executions. In consequence, an untrusted object cannot call doPrivileged on a trusted object to bypass access control.

B. Taking Advantage of AspectScript

As mentioned before, ZAC is implemented on top of AspectScript. AspectScript is an extension to JavaScript adding support for aspect-oriented programming [5]. In aspect-oriented programming, execution of a program is represented as a series of join points (a function call, a property access, an object creation, etc.—see Figure 6). A pointcut identifies a set of join points, and a piece of advice is the action to be taken at a join point matched by a pointcut. In AspectScript, an aspect is a pointcut-advice pair, where both pointcuts and advices are plain JavaScript functions that receive a join point as parameter.

AspectScript's aspects corresponds exactly to ZAC's restrictions, where the rule property is the pointcut, and the action property is the piece of advice. The policy enforcement (stack-based access control and privileged actions) is implemented based on dynamic deployment of aspects and expressive scoping [14]. ZAC depends only on AspectScript to be secure: if AspectScript generates all the events associated to the execution of all untrusted objects, these objects will not be able to do anything without ZAC being aware of it. This is the reason why ZAC policies cannot be circumvented.

The fact that ZAC is implemented on top of AspectScript brings many benefits, of which we highlight two: the ample variety pointcuts included in AspectScript, and the ability to reason about the program execution.

AspectScript includes several predefined pointcuts that can be composed to identify more intricate actions. For example, the following pointcut identifies the calls to alert that occur inside the body of any method of obj:

```
var PCs = AspectScript.Pointcuts;
var pc = PCs.call(alert).and(PCs.within(obj));
```

Pointcuts in AspectScript can also match sequences of events, optionally restricted by temporal conditions. This can be useful in access control to, for instance, forbid the invocations to alert that occur too frequently, say more than one per second.

The ability of ZAC to reason about the execution of a program derives from the fact that AspectScript generates join points for every action in the scripts, evaluating pointcuts for each one of them. In a capability-based model, for instance, reasoning about the history of program execution is impossible. This is because the model considers accessing a reference and executing some methods as the only way to threaten a system. However, there are other interesting security properties that are related to the actual computation of programs.

An example of a security property based on actual computation—which is actually checked by most browsers—is to ensure that a third-party script does not degrade the interactive experience of the user: if a script takes too long to execute, the browser suspends its execution and asks the user whether to simply abort the script. While such a restriction is beyond the realm of capabilities, it is straightforward to define with ZAC:

```
function nInstructionsRestriction(n){
  return {
    nInstructions: 0,
    rule:   function(event){
      return ++nInstructions > n;
    },
    action: function(event){
      if(confirm("This script is running for too long. Abort it?"){
        throw "Cannot execute more than " + n + " instructions";
      }
      nInstructions = 0;
      return event.proceed();
    }
  }; }
```

For simplicity, we use instruction count rather than actual time. The point here is just to give an idea of the wide range of policies that can be expressed using dynamic AOP as a foundation for defining restrictions.

V. Conclusion

The ubiquity of the JavaScript language and the myriad of ways in which it is being used turn access control into a crucial element for a safe web experience. In this article we presented ZAC, a lightweight practical library for access control in JavaScript comprising very interesting features which permit to express security properties not enforceable with other approaches, such as per-object restrictions and bounded-time execution. ZAC's implementation is based on aspect orientation, which, apart from constituting a solid ground currently under active development, brings many practical benefits in terms of expressiveness.

Availability. ZAC is available at `http://pleiad.cl/aspectscript/zac`, and AspectScript at `http://pleiad.cl/aspectscript`.

References

[1] AdSafe. `http://www.adsafe.org`.

[2] Caja. `http://code.google.com/p/google-caja`.

[3] John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6):1029–1052, 2004.

[4] John Clements, Ayswarya Sundaram, and David Herman. Implementing continuation marks in javascript. In *Proceedings of the Workshop on Scheme and Functional Programming*, 2008.

[5] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), October 2001.

[6] FBJS. `http://wiki.developers.facebook.com/index.php/FBJS`.

[7] Cédric Fournet and Andrew D. Gordon. Stack inspection: theory and variants. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(3):360 – 399, 2003.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, 3rd edition*. Addison-Wesley, 2005.

[9] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[10] ECMA International. *ECMAScript Language Specification. ECMA-262*. ECMA, 5 edition, December 2009.

[11] JSON. `http://www.json.org/`.

[12] Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[13] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Trans. Web*, 1(3):11, 2007.

[14] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, pages 168–179, Brussels, Belgium, April 2008. ACM Press.

[15] Rodolfo Toledo, Paul Leger, and Éric Tanter. AspectScript: Expressive aspects for the Web. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 13–24, Rennes and Saint Malo, France, March 2010. ACM Press.

[16] Dan Wallach and Edward Felten. Understanding Java stack inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–63, 1998.