Declaratively Specifying Security Policies For Web Applications

Angel Luis Scull Pupo Jens Nicolay Elisa Gonzalez Boix

Sofware Languages Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel
{ascullpu,eljenso,egonzale}@vub.ac.be

Abstract

The complex architecture of browser technologies and dynamic characteristics of JavaScript make it difficult to ensure security in client-side web applications. Browser-level policies alone, such as Content Security Policy and Same-Origin Policy, are not sufficient because they are implemented inconsistently across browsers and can be bypassed. At the application level, however, there exists no specification language for expressing a wide range of security policies in a composable and reusable manner.

In this paper we develop a declarative language for encoding an combining security policies in the context of JavaScript web applications. We explore JavaScript's reflection capabilities to enforce these security policies dynamically. We validate our work by expressing common security policies encountered in the literature.

1. Introduction

Today web applications can be seen as mashups of JavaScript code and content from various sources. All JavaScript code included from different sources has the same privileges to access sensitive resources of the browser such as cookies, location, etc. This situation exposes web applications to security threats such as Cross Site Scripting, Cross Site Request Forgery, Sensitive Data Exposure, etc. [9, 13, 18]. Many efforts have been done at the browser level to mitigate these security threats. For example, the Content Security Policy (CSP) allows developers to inform the browser about the sources from which the application expects to load resources. The Same-Origin Policy (SOP) restricts the content a web page can access to resources of the same origin. Nevertheless, the implementation of SOP and CSP present inconsistencies across browsers vendors, and can be bypassed [3, 22]. As a result, browser-level efforts need to be complemented with application-level security policies.

In this paper, we present an application-level security mechanism for specifying and enforcing security policies. In particular, we aim to answer two research questions:

1. How can application-dependent security policies be expressed so that they are composable?
2. Can JavaScript's reflective capabilities be employed to enforce security policies without compromising their *transparency* and *tamper-proofness*? [1]

We discuss aspects about *transparency* and *tamper-proofness* as a key point for future development, but we do not aim cover them here.

2. Security policies for client side web applications

We consider an application-level security policy as a program property that must hold during the entire application's execution. Schneider et al. [24] classify security policies in three classes:

1. *access control policies* restrict what operations principals can perform on objects,
2. *information flow policies* restrict what principals can infer about objects from observing system behavior, and
3. *availability policies* restrict principals from denying others the use of a resource.

While the goal of our research is to cover those three types, our current prototype implementation only supports access control policies. Therefore we focus on access control security policies in this paper, and we will refer to them as security policies in the rest of the paper.

2.1 Expressing security policies

To express security policies, many approaches support *imperative* specifications. In this case, security policies are expressed either in JavaScript directly (like in JSand [1]) or in the implementation language of the JavaScript virtual ma-

[1] By transparency we refer to the ability of a security policy that not interfere with the target code it protects, and tamper-proofness means that security policies themselves should not be vulnerable to attacks.

Meta'16 October 30, 2016, Amsterdam, Netherlands

chine (C++ in Richards et al. [21]). These approaches offer the flexibility of a full-fledged general purpose language. However, it is the developer's responsibility to make sure that the policy is tamper-proof, and does not contain bugs or errors that result in new vulnerabilities. In addition, developers have to manually call the enforcement mechanism and perform the security checks themselves.

Alternatively, security policies can be written in a fully *declarative* manner [5, 7, 10]. Typically, declarative approaches embed type annotations or access paths to a program as policies. Heidegger et al. [7] implement access permission contracts using a DSL embedded inside code comments. Keil et al. [10] propose to express access control paths in a regular expression language. Such a language is completely declarative, allowing developers to express complex access control paths that automatically work in a transitive manner. Drossopoulou et al. [5] propose a *domain-specific language* (DSL) to express trust, risk, and security policies in open systems. Their definitions make use of logical predicates and assertions in an object capability language.

Finally, some works combine imperative and declarative specifications. Phung et al. [19, 20] support type enforcement policies which are declaratively specified as a list of types that are accepted as a function's arguments, while user-defined policies are still imperative. Their approach to express user-defined security policies is based on *Aspect-Oriented Programming* (AOP) [11]. Developers express security properties in an imperative way at the granularity of methods and properties of built-in objects. However, this can lead to security misconfigurations and inconsistencies that can be used in attacks. In addition, Phung et al. do not propose a mechanism to combine and reuse security policies. ConScript [15] is also based on AOP and proposes a browser-based aspect system that allows the creation of application-specific security policies. Policies are expressed in a declarative/imperative hybrid manner using advice functions like *around*. Security checks are written in an imperative manner while the code that will be checked (e.g `window.open`) is expressed in an aspect-oriented manner. Like Phung et al.'s work, ConScript does not provide any mechanism to combine policies.

2.2 Enforcing security policies

Execution monitoring (EM) is a common technique to enforce security policies. EM ensures that a target program satisfies a certain specified property, and can notify or halt the system when this property is violated [24].

An EM mechanism can be implemented as part of the target system by relying on code instrumentation [20, 28] or VM modifications [15, 21]. However, VM modifications have as a drawback that they limit the portability of security policies (as they need to be re-implemented and customized for each browser), and code instrumentation results in performance impact.

Alternatively, EM can be achieved by observing and modifying the target system on the fly by employing JavaScript's reflective capabilities. As a result, policies then can only be applied at the object level; primitive values cannot be fully traced by the current JavaScript Proxy API. This in turn may limit the transparency and tamper-proofness of the enforcement mechanism.

2.3 Problem Statement

Although much research has focused on imperative specifications of security policies, declarative security policies have a number of advantages. First, declarative specifications offer a well-defined interface for expressing policies, constraining developers to a particular way of defining a policy which may lead to less error-prone code. However, they are less flexible than imperative specifications as they usually require policy developers to use new notations to express their policies, needing additional support for enforcing them in an engine, parser, or compiler. Studied works rely on new languages to express security policies obtaining more freedom in expressiveness but forcing developers to learn a new language. Indeed they lack the facilities of the host language.

We claim that these observations motivate the need for a new declarative domain-specific language (DSL) that combines the flexibility of imperative policies with the ease of development provided by more declarative solutions. In this paper we present GUARDIA, an *internal DSL* [8] for JavaScript for declaratively specifying security policies for client side web applications. We aim to distill fundamental security policies from the wide range of solutions describe above, in order to incorporate them in GUARDIA as building blocks that can then be composed to express more complex policies. We design GUARDIA's enforcement mechanism in a modular way so that we can experiment with different enforcement solutions. In the current prototype implementation, we employ ECMAScript 2015 [27] meta-level facilities to develop an EM to enforce security policies, and we discuss a design toward an hybrid approach in which code instrumentation is combined with JavaScript proxies to improve the transparency and tamper-proofness of policies.

3. GUARDIA: An internal DSL for Declaratively Specifying Security Policies

In this section we describe GUARDIA, a novel declarative DSL to express security policies for client side web applications written in JavaScript. GUARDIA provides a predefined set of fundamental policies that can be composed to build more complex ones. Fundamental policies alleviate the developer of the burden of correctly writing security policies, and the built-in composability mechanisms provides the flexibility of imperative specifications. We will first explain how to define security policies in GUARDIA, and we describe the main building blocks to create policies and combine them, and finally how to deploy them on JavaScript ob-

jects. Section 4 will explain other relevant constructs through coding examples.

In order to enforce security policies, GUARDIA uses execution monitoring of security-relevant invocations. The execution of the target program halts if one of these security-relevant invocations does not comply with one of the policies. We follow a similar attack model adopted by Phung et al. in [20] and assume that (1) the code loaded by the web browser is injected with malicious code, and (2) malicious code has access to all resources of the browser. A security relevant-invocation can be a call to a method or a function, or access to an object property. For example, a call to the *window.open* method could be used for malicious purposes, and therefore its invocation is considered security-relevant.

3.1 Implementation

In this work, we employ the traps defined in the ECMAScript Proxy API [16]. We use proxies to change the semantics of a subset of operations on certain JavaScript objects to enforce security policies by intercepting the relevant invocations. As such, checks in GUARDIA are limited by the parameters that are accepted by the proxies' traps.

The idea is to wrap objects with proxies to uphold the security invariants. To build a proxy object, a target object and a handler should be provided, as Listing 1 shows. The proxy's constructor returns a wrapper of the target object. This wrapper intercept all read and write operations intended upon the object, enforcing security policies before funneling the invocation to the target object.

```
var objProxy = new Proxy(obj,{
    get: function(tar,prop,rec){
      //security policy enforcement
        return Reflect.get(tar,prop,rec);
    }
    set: function(tar,prop,val,rec){
      //security policy enforcement
      Reflect.set(tar,prop,val,rec)
    }
  })
```

Listing 1. Example of a Javascript proxy

A security policy in our implementation is a JavaScript object that specifies a number interception points used by the enforcement mechanism to hook in the adequate traps.

A security policy object defines two types of interception points which monitor security-relevant read and write operations, such as a method invocation or the assignment to a property in the target object, respectively. Developers can register listeners to monitor these read and write operations. The listing 2 shows how to define policy object that denies a read operation on the *open* property/method. We specify each field that could have a policy. The *whenRead* and *whenWrite* fields take an array of predicates that are evaluated on each read or write operation. Similarly, the *readListeners* and *writeListeners* fields take an array of listeners (closures) that are notified on each read or write operation. Each registered listener is a JavaScript object that contains a *notify* function.This function receives as parameters the dynamic information related to the actual invocation. The *notify* function is executed each time a property is accessed or a method is invoked.

```
var pol = {
    whenRead: [Deny(['open'])],
    whenWrite: [...],
    readListeners: [{
      notify: (tar, prop, rec, args) => {
        //update some state...
      }
    }]
    writeListeners: [...]
}
```

Listing 2. Example of a policy definition

All fields of policy definition object receives arrays of objects. Those arrays are iterated to execute the registered behavior in the corresponding trap. For example, `whenRead` field register all predicates that must be triggered when a read operation occurs on the proxy object. Listing 3 shows how all registered predicates in `whenRead` are executed. This code snipped is part of the handler that is used to build secured proxy objects. Before executing the read operation, each policy is checked by means of `filter` method. The execution throws an error if one them returns false.

```
  ...
  get: function(tar, prop, rec){
    for(let pred of whenRead){
      if(!pred.filter(tar, prop, rec))
        throws new Error("Not allowed");
    }
    ...
    return Reflect.get(tar, prop, rec);
  }
  ...
```

Listing 3. Example of a proxy trap implementation

3.2 Security Policy Deployment

Security policies must be deployed on objects. In our approach, the result is a proxy object that acts as a handler of the target object. Listing 4 shows how to deploy a policy for the *window* object.

```
...
var winProxy = installPolicy(pol).on(window);
```

Listing 4. Policy deployment example

To construct the proxy we first build the handler that implements *get* and *set* traps. These traps trigger the checks (see Listing 3) registered in the policy. If the result of the check is true, then the trap notifies the listeners and then forwards execution to the target object. Otherwise, program execution is halted.

3.3 Basic Security Policies

Elemental components of a security policy have been distilled into a number of *traits* [23] that are then composed

to define a security policy. This allows developers to further compose the provided policies to build more complex ones.

Following Schneider's security policy definition in [24], the most basic type of policy is a predicate that decides whether an invocation is valid or not. Listing 5 shows such a trait which requires a *filter* method that should evaluate to a boolean value.

```
var TBase = Trait({
    filter: Trait.required
});
```

Listing 5. Basic policy interface

`TBase` represents the most abstract policy which is extended to provide the set of elementary policies supported by GUARDIA. Listing 6 shows how we extended `TBase` in the `TAllow` trait to support policies that permit only certain executions on a target object. As the listing shows, `TBase` is composed with a new trait that provides the behavior of the filter method. `TAllow` is materialized by the function `Allow`, that takes as parameter an array of strings and returns an object that implements `TAllow`. The resulting object contains the behavior of the policy. We do not provide a trait for a *deny* concept, because it can be expressed in terms of `TAllow`.

```
var TAllow = Trait.compose(TBase, Trait({
    filter: function (tar, prop, rec, args) {
        return contains(this.allowed, prop);
    }
}));

function Allow(properties) {
    var allPropPolicy = Trait.create({}, TAllow);
    allPropPolicy.allowed = properties;
    return allPropPolicy;
}
```

Listing 6. Implementation of Allow concept

GUARDIA must not only allow to develop simple policies, but more elaborated policies or combined policies as well. To this end we provide three *higher-order* policies that can be used to combine policies based on the three traditional logical operators. GUARDIA provides them as three JavaScript functions:

- the `Not` function receives as parameter a policy object A and returns a policy object B that negates the behavior of A. We use this building block to reify the *Deny* concept.
- the `And` function returns a policy that behaves as a logical and of the policies provided as parameters.
- the `Or` function returns a policy that evaluates to true if one of the policies given as parameter returns true.

Listing 7 shows the implementation of the And function that returns a policy that behaves as a logical `and` operator.

```
function And(...policies) {
  return Trait.create({},
  Trait.compose(TBase, Trait({
    filter: function (t, p, r, args) {
      for (var p of policies) {
        if (!p.filter(t, p, r, args)) {
          return false;
        }
      }
      return true;
    }
  })))
}
```

Listing 7. Implementation of And concept

So far, we are able to design policies that allow or deny the invocation of methods or the access to the properties of a target object. But they are not sufficient to express control flow policies like *no dynamic iframe creation* in which the execution of the *document.createElement(tag)* must halt *only* when the value of the tag attribute is equal to *iframe*. To this end, GUARDIA provides the `ParamAt` function that returns a policy which can be used to check if a specific parameter of a method invocation hold some property. Listing 8 shows the implementation of `ParamAt` function. The first parameter is a function that will be called by the `filter` method. This function must return a boolean value. The second parameter is the index of the parameter of the current invocation. The last parameter is an arbitrary value used by the function provided to `ParamAt`. The result object of `ParamAt` is composed with `TParamAt` trait that implement the filter function required for predicates. In this particular case the implementation of the filter function applies the provided function to the specified parameters.

```
var TParamAt = Trait.compose(TBase, Trait({
    idxParam: Trait.required,
    otherParam: Trait.required,
    opFn: Trait.required,

    filter: function (tar, prop, rec, args) {
        return this.opFn(args[this.idxParam],
                this.otherParam);
    }
}));

function ParamAt(fn, idx, other) {
    var bProto = {};
    bProto.idxParam = idx;
    bProto.otherParams = other;
    bProto.operatorFn = fn;
    return Trait.create(bProto, TParamAt);
}
```

Listing 8. Implementation of ParamAt concept

Many security policies require to take into account the program's state in order to determine whether a certain property holds. Function `StateFnParam` allows the application of a function to check the arguments of the actual security-related invocation. Listing 12 shows an example of its usage.

4. GUARDIA in Action

In order to show how developers can encode security policies in GUARDIA we now show the implementation of eight

security policies documented in previous works [15, 20, 28]. For each policy, we first briefly explain the policy and type of attack it prevents, and then discuss the code in GUARDIA.

Policy #1: Limit popup window construction

Like Kikuchi et al. [12] and Meyerovich et al.[15], we limit the number of attempts to open a popup window. Also, as suggested by Phung et al. [20], we check that the new window has a location and status bar. We extended the invariant of the policy by checking that the URL is in a whitelist. The policy registers a listener that increments a counter each time a window is open. Later the predicate verifies that the first parameter is in a whitelist of URLs. It also verifies that the second parameter contains a location and status bar. We use the `StateLessThan` function to check if a certain variable state is less than a value.

```
var lstnr = {
  notify: function (tar, name, rec, args) {
    if (name === 'open') {
      var winOpCnt = ac.getState('winOpenCount');
        if (winOpCnt) {
          winOpCnt += 1;
          ac.setState('winOpenCount', winOpCnt);
        } else {
          ac.setState('winOpenCount', 1);
        }
      }
    }
}
var contains = (a, b) => {
                  return  a.indexOf(b) != -1
        }
var limitWin = Or(
        And(
            Allow(['open']),
            ParamInList(0,urls),
            ParamAt(contains,1,
                        'location=yes'),
            ParamAt(contains,1,
                          'status=yes'),
            StateLessThan('winOpenCount',3)),
        Not(Allow(['open'])))

var proxy = installProxy({whenRead:[limitWin]
                          readListeners:[lstnr]})
                         .on(window)
```

Listing 9. Limit popup window construction policy

Policy #2: Prevent resource abuse

Client-side resource abuse in JavaScript can adversely affect user experience to the point that the application becomes unusable [20]. There exist certain methods in the DOM API that can be exploited for this kind of attack like `prompt and alert` [3, 15, 20]. Listing 10 shows how to create a policy that prevents resource abuse of the methods `prompt`, `alert` and `confirm` in GUARDIA. At each invocation, the policy checks the name of the property being accessed. If the property is one of the specified by the policy, then the invocation is denied.

```
var noResAbuse = Deny(['alert','prompt',
                                  'confirm'])
var proxy = installPolicy({whenRead:[noResAbuse]})
                                  .on(window)
```

Listing 10. Prevent modal dialogues abuse policy

Policy #3: Disallow dynamic *iframe* creation

This policy aims to solve attacks that can happen by restoring built-in methods from another page as stated in [20]. Listing 11 shows how to build such a policy by negating the combination of a `Allow` policy with a `ParamAt` which applies the function `equals` to the first invocation parameter of `createElement` function and 'iframe'.

```
var equals = (a, b) => { return a === b }
var pol = not(and(
          Allow(['createElement']),
                    ParamAt(equals, 0, 'iframe')))
var proxy = installPolicy({whenRead: pol})
                  .on(document)
```

Listing 11. Disallow dynamic iframe creation policy

Note that in contrast to Phung et al. in [20], the developer is not aware of the enforcement mechanism, and does not need to manually enforce the policy and as such, the code of the security policy is less exposed to coding errors.

Policy #4: Disabling page redirects after `document.cookie` read events

Cookies are commonly used by web servers to store data regarding to user session. If an attacker is allowed to make a request after reading information stored in cookies, this could cause leakage of valuable information [12, 15, 20]. There are different ways to make a request to an external site, but here we present a policy that disallows changing the `location` property of the window to avoid such an attack.

Listing 12 shows how to construct such a policy by combining a listener(lines 1 to 4) and the predicate of the policy (lines 5 to 10). In the predicate, any attempt to change the `location` triggers the execution of `StateFnParam` that verifies if `cookieRead` is false. Otherwise, it is not allowed to change the `location`. Lines 10 to 13 install the policy.

```
1  var lstnr = {
2    notify: (t,p,r,a) => {
3      if(p === 'cookie'){
4        setState('cookieRead',true) }}}
5  var noRedirect = Or(
6    And(   Allow(['location']),
7           StateFnParam(equals,'cookieRead',false)),
8    Not(Allow(['location']))
9  )
10 installPolicy({
11   whenWrite: [noRedirect],
12   readListeners: [lstnr]
13 }).on(window)
```

Listing 12. Disabling foreign links after cookies access

Policy #5: Allowing redirections for a whitelist of URLs

Both Pungh et al. [20] and Meyerovich et al. [15], propose a policy to prevent redirection to another web site by means of changing the location property of the *window* and *document* objects. Similarly, we can prevent leakage of information by changing the source location of images, forms, frames, and iframes. Listing 13 illustrates this policy in GUARDIA. Redirections and setting of source locations are allowed only for URLs that are contained in a whitelist.

```
var urls = ['http://google.com',
            'http://facebook.com']
var whtList = Or(
      And(Allow(['location']),
          ParamInList(0,urls))),
          Deny('location'))

installPolicy({whenRead:[whtList]})
                    .on(document.location);

var pol = Or(And(
      Allow(['src']),
      ParamInList(0, url)),
      Not(Allow(['src'])));

installPolicy({whenRead:[pol]})
                    .onAll(document.images);
```

Listing 13. Allowing redirections for a whitelist of URLs

Policy #6: Restrict `XMLHttpRequest` to secure connections and whitelist urls

Phung et al. [20] prevents impersonation attacks using the *XMLHttpRequest* object by restricting its *open* method to whitelist urls. Meyerovich et al. [15] proposes a policy that enforces an HTTPS request when user and password arguments are supplied to the *open* method. Here we implement a security policy that compose these approaches.

```
var startsWith = (a,b) => {return a.startsWith(b)}
var isHTTPS = StateFnParam(1,startsWith, 'HTTPS')
var pol = Or(And(
        Allow(['open']),
        ParamInList(1,urls),
        isHTTPS,
        Not(ParamAt(equals,3,undefined)),
        Not(ParamAt(equals,4,undefined))
        ),
            And(
        Allow(['open']),
        ParamInList(1,urls),
        Not(isHTTPS)  ),
        Not(Allow(['open']))));
XMLHttpRequest = installPolicyCons(pol,
                                   XmlHttpRequest);
```

Listing 14. Restrict XMLHttpRequest requests to HTTPS and whitelist urls

Policy #7: Allowing a whitelist cross-frame messages

Cross-origin communication using `window.postMessage` can lead to attacks such as Cross Site Scripting and Denial of Service. The policy below is intended to prevent these kind of attacks by checking that the origin URL of the message is white-listed. The predicate of the policy verifies, by means of `ParamInList`, that the second parameter of the invocation of `postMessage` is contained in a whitelist of URLs. If this is not the case, then the invocation of `postMessage` is denied.

```
var urls = ['http://google.com',
                        'http://facebook.com']

var whtList = Or(And(Allow(['postMessage']),
      ParamInList(1,urls))), Deny('postMessage'))
var proxy = installPolicy({whenRead: [whtList]})
                                    .on(window)
```

Listing 15. Allowing a whitelist cross-frame messages

Policy #8: Disallow *string* arguments to *setInterval* and *setTimeout* functions

This policy aims to disallow the execution of arbitrary code as described in [15]. Functions *setTimeout* and *setInterval* can accept a closure or string as callback argument. As such, these functions can be abused to run malicious code.

Listing 16 shows how we express a policy to restrict the execution of these functions to closures. In the policy below the execution of `setTimeout` and `setInterval` is permitted only if the first parameter of the invocation is a function.

```
var pol = Or(
     And(Allow(['setTimeout','setInterval']),
         ParamWithType(0, 'function')),
     Not(Allow(['setTimeout','setInterval'])))

var prox = installPolicy({whenRead: pol})
                                    .on(window)
```

Listing 16. Disallow string arguments policy

5. Evaluation and Discussion

We evaluated GUARDIA by expressing 13 different security policies extracted from literature (cf. section 2) [6, 15, 20, 28]. Table 1 provides an overview of all policies that have been expressed in GUARDIA. The table is an adaptation from the table presented in Bielova [3] that we extended with the type of attack that each policy aims to prevent. We have already shown in the previous section the implementation of 8 of those security policies as representative examples of each type of attack. They are marked in the the table with their corresponding number. As the table shows GUARDIA is able to cover all policies analyzed in the related work.

In the remainder of this section we discuss the main features and limitations of our approach.

5.1 DSL vs. GPL

In this work, we explore an internal DSL for expressing security policies in contrast to creating a standalone declarative

Attack type	Security policy	HV [6]	Yu et al. [28]	Phung et al. [20]	ML [15]	GUARDIA
Forgery	Limited number of popup windows opened **(P1)**	✓	✓	✓	✓	✓
Forgery	No popup windows without location and status bar			✓		✓
Resource abuse	Prevent abuse of resources like modal dialogues **(P2)**	✓		✓	✓	✓
Restoring built-ins from frames	Disallow dynamic iframe creation **(P3)**			✓	✓	✓
Information leakage	Disabling page redirects after `document.cookie` read **(P4)**	✓	✓	✓	✓	✓
Information leakage	Allowing redirections for a whitelist of URLs **(P5)**			✓	✓	✓
Information leakage	Restrict `XMLHttpRequest` to secure connections and whitelist URLs **(P6)**				✓	✓
Information leakage	Disallow setting of `src` property			✓		✓
Information leakage	Disallow setting of `location` property			✓	✓	✓
Impersonation	`XMLHttpRequest` is restricted to HTTPS connections **(P6)**				✓	✓
Impersonation / Information leakage	Disallow open and send methods of XHR object			✓	✓	✓
Man in the middle	*postMessage* can only send to the origins in a whitelist **(P7)**				✓	✓
Run arbitrary code*(fix)	Disallow *string* arguments to *setInterval* & *setTimeout* **(P8)**				✓	✓

Table 1. Comparison of approaches in security policies. Policy numbers **P1** to **P8** refer to the policies discussed in section 4.

programming language like in [5], or expressing policies as a library in a full-fledged GPL. This choice is motivated by the fact that we aim to provide basic and extensible building blocks distilled from the domain of security policies.

In some cases, expressing a policy in GUARDIA is more verbose than using an imperative approach, but we argue that our language has the benefit of preventing code duplication and coding errors as it incorporates a number of patterns and offers them in well-defined building blocks. In addition, in imperative approaches like [15, 20] developers mix the code of security policies with the enforcement mechanism, decreasing code maintainability. Another advantage of implementing GUARDIA as a DSL is that the security policy specification language can be decoupled from the enforcement mechanism that monitors the program execution. As mentioned in section 3.3, the security policy language interacts with the enforcement mechanism by a well-defined interface that provides information regarding the target object, the property being accessing, and the parameters of the actual call.

GUARDIA only supports access control policies for now, but we are currently working on extending it to support information flow control policies as well. In particular we have conducted the first implementation steps for supporting dynamically taint analysis [2, 4, 25].

5.2 Transparency

Like [20, 24], we aim to achieve transparency in the sense that the behavior of a target object in which a policy is installed should not affected by GUARDIA's EM mechanism. We rely on Van Cutsem and Miller's [26] *invariant enforcement* mechanism to uphold the invariants of target objects. The transparency of our solution is thus tied to the transparency of JavaScript proxies. First experiments to investigate how proxies behave in a real environment on different browsers and using libraries like JQuery, has shown some issues. For example, JQuery presents errors when we wrap methods of the window or document objects. To date, we do not know if JQuery poses an strong invariant or if it is an issue in the implementation of the Proxy API. We are currently experimenting with alternative enforcement mechanisms based on the combination of proxies and code instrumentation [4] in order to improve the transparency and as well tamper-proofness of the security policies.

5.3 Tamper-proofness

Tamper-proofness is achieved by making security policies secured so that they are not vulnerable to attacks. This is quite challenging in JavaScript and specially, when employing JavaScript proxies as the basis of the enforcement mechanism, which cannot intercept operations with primitive values. Primitive operations (e.g addition, multiplication, etc.) and comparisons in JavaScript use implicit type conversions which can be exploited to bypass the security mechanism. In the remainder of this section we discuss three attacks which can compromise the tamper-proofness of GUARDIA.

Redefinition of `toString` and `valueOf` functions. Listing 17 shows a code snippet on how an attacker could provide an object that at policy evaluation that redefines the `toString` function to bypass a whitelist policy which restricts access to 'bad' URLs. To avoid this problem we adopt the same approach of Phung et al. [20] by converting all policy parameters to primitive values and only using the converted values in target invocations. A mechanism to protect built-in objects in order to protect against undesired behavior and state is work in progress.

```
var liarObj = {
  value : 'good',
  toString : function(){
    var result = this.url
    this.value = 'bad'
    return result;
  }
}
console.log(liarObj.toString()) //good
console.log(liarObj.toString()) //bad
```

Listing 17. Example of ToString redefinition

Function aliasing. Our language relies on the names of functions and properties to validate their invocation. Relying on names to ensure security is an easy way of restricting access to certain functionalities or data. However, in JavaScript it is easy to create function aliases because functions are objects allocated on the heap. For example, `window.open` function can be aliased with `myFun` by assignment: `myFun = window.open`. An attack can used the aliased function to circumvent security policy enforcement. This potential risk is discussed by Phung[20] and Meyerovich [15].

To prevent such risks associated with aliasing, we rely on the fact that the deployment of security policies must be realized *before* any other code that can create aliases of target objects is executed. As such, all aliases are created to the security policy and the target object is never exposed to client code. Alternative ways to prevent function aliasing is a point for future work.

Prototype poisoning. An attacker could take advantage of the Javascript's prototype inheritance chain to compromise GUARDIA tamper-proofness. Since every JavaScript object is created in an extensible and configurable state, properties can be freely added and modified at any point of the object's lifetime. An attacker can at any time change the prototype of an object and use the inheritance chain to then bypass a security policy. As described in in [14] there are different attacks related to prototype poisoning: built-in subversion, global setter subversion, and policy object subversion.

GUARDIA cannot currently deal with this kind of attacks. In future work, we aim to explore ECMAScript 5 new primitives that allow to strengthen and protect objects from unintended changes [17, 26]. An object can become *non-extensible* so that it is not possible add new properties to the object, and *non-configurable* so that any attempt to change non-configurable properties fails. We can employ these two primitives for preventing changes on the target object which would conflict with the property expressed in a security policy installed on them. Note, however, this implies trading transparency (since the target object will not behave as if there was no security policy installed on it) in favor of tamper-proofness for dealing with prototype poisoning attacks.

6. Conclusion

We presented an approach to declaratively specify security policies for web applications. We introduced GUARDIA, a DSL for expressing composable security policies that combines the flexibility of imperative specification languages with the ease of development provided by more declarative solutions. Security policies in our approach are enforced by wrapping target objects with proxies that intercept security-relevant invocations. To prevent developers from making mistakes in the enforcement of the policies, GUARDIA is agnostic of the underlying enforcement mechanism. To date we are not aware of the existence of a similar approach in the context of web applications. To evaluate our approach, we implemented 13 security policies found in related work, and found that GUARDIA is capable of expressing all of them.

Acknowledgments

This research is funded by the Secloud Secure-IT Strategic Platform project of the Innoviris.brussels research agency.

References

[1] P. Agten, S. Van Acker, Y. Brondsema, P. H. Phung, L. Desmet, and F. Piessens. Jsand: Complete client-side sandboxing of third-party javascript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 1–10, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1312-4. doi: 10.1145/2420950.2420952. URL http://doi.acm.org/10.1145/2420950.2420952.

[2] T. H. Austin, T. Disney, C. Flanagan, T. H. Austin, T. Disney, and C. Flanagan. *Virtual values for language extension*, volume 46. ACM, Oct. 2011.

[3] N. Bielova. Survey on JavaScript security policies and their enforcement mechanisms in a web browser. *The Journal of Logic and Algebraic Programming*, 82(8):243–262, Nov. 2013.

[4] L. Christophe, E. G. Boix, W. De Meuter, and C. De Roover. Linvail - A General-Purpose Platform for Shadow Execution of JavaScript. *SANER*, pages 260–270, 2016.

[5] S. Drossopoulou, J. Noble, and M. S. Miller. Swapsies on the internet: First steps towards reasoning about risk and trust in an open world. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, PLAS'15, pages 2–15, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3661-1. doi: 10.1145/2786558.2786564. URL http://doi.acm.org/10.1145/2786558.2786564.

[6] O. Hallaraker and G. Vigna. *Detecting malicious JavaScript code in Mozilla*. IEEE, 2005.

[7] P. Heidegger, A. Bieniusa, and P. Thiemann. Access permission contracts for scripting languages. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 111–122, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103671. URL http://doi.acm.org/10.1145/2103656.2103671.

[8] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196–es, Dec. 1996.

[9] X. Jin, T. Luo, D. G. Tsui, and W. Du. Code injection attacks on html5-based mobile apps. *CoRR*, abs/1410.7756, 2014. URL http://arxiv.org/abs/1410.7756.

[10] M. Keil and P. Thiemann. Efficient dynamic access analysis using javascript proxies. In *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pages 49–60, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2433-5. doi: 10.1145/2508168.2508176. URL http://doi.acm.org/10.1145/2508168.2508176.

[11] G. Kiczales. Aspect-oriented programming. In *27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 730–730. IEEe.

[12] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov. JavaScript Instrumentation in Practice. In *Programming Languages and Systems*, pages 326–341. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[13] S. Lekies, B. Stock, M. Wentzel, and M. Johns. The unexpected dangers of dynamic javascript. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 723–735, Washington, D.C., Aug. 2015. USENIX Association. ISBN 978-1-931971-232. URL https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lekies.

[14] J. Magazinius, P. H. Phung, and D. Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In *Informatics*, pages 239–255. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[15] L. A. Meyerovich and B. Livshits. ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser. In *2010 IEEE Symposium on Security and Privacy*, pages 481–496. IEEE, 2010.

[16] M. S. Miller and T. Van Cutsem. *Proxies: design principles for robust object-oriented intercession APIs*, volume 45 of *design principles for robust object-oriented intercession APIs*. ACM, Dec. 2010.

[17] M. D. Network. MDN object, 2016. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object.

[18] G. K. Pannu. A Survey on Web Application Attacks. *IJCSIT) International Journal of Computer Science and ...*, 2014.

[19] P. H. Phung and L. Desmet. A two-tier sandbox architecture for untrusted javascript. In *Proceedings of the Workshop on JavaScript Tools*, JSTools '12, pages 1–10, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1274-5. doi: 10.1145/2307720.2307721. URL http://doi.acm.org/10.1145/2307720.2307721.

[20] P. H. Phung, D. Sands, and A. Chudnov. Lightweight self-protecting javascript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-394-5. doi: 10.1145/1533057.1533067. URL http://doi.acm.org/10.1145/1533057.1533067.

[21] G. Richards, C. Hammer, F. Zappa Nardelli, S. Jagannathan, and J. Vitek. Flexible access control for javascript. *SIGPLAN Not.*, 48(10):305–322, Oct. 2013. ISSN 0362-1340. doi: 10.1145/2544173.2509542. URL http://doi.acm.org/10.1145/2544173.2509542.

[22] H. Saiedian and D. Broyle. Security vulnerabilities in the same-origin policy: Implications and alternatives. *Computer*, 2011.

[23] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable Units of Behaviour. In *ECOOP 2003 – Object-Oriented Programming*, pages 248–274. Springer Berlin Heidelberg, Berlin, Heidelberg, July 2003.

[24] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1): 30–50, Feb. 2000.

[25] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ. In *the 2009 ACM SIGPLAN conference*, pages 87–97, New York, New York, USA, 2009. ACM Press.

[26] T. Van Cutsem and M. S. Miller. Trustworthy Proxies. In *ECOOP 2013 – Object-Oriented Programming*, pages 154–178. Springer Berlin Heidelberg, Berlin, Heidelberg, July 2013.

[27] A. Wirfs-Brock. Ecmascript 2015 language specification, 2015.

[28] D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. *ACM SIGPLAN Notices*, 42(1):237–249, Jan. 2007.