VRIJE
UNIVERSITEIT
BRUSSEL

CLOUD COMPUTING

JSFlow and Caja - Two Web Security Approaches

Carlos Javier Rojas Castillo

August 26, 2020

Professor: Jens Nicolay, Assistant: Angel Luis Scull Pupo

Computer Science
Master in Software Languages and Software Engineering

Contents

1 Context

Nowadays, most of the web applications are *web mashups*, that is, a web application that combines custom content and third-party services into a new service[1]. The advantages of such development style is the ease and speed in how web applications are developed[1].

Unfortunately, web mashups come with security risks. The script inclusion mechanism, required to include third-part content into a host, gives the third-party content same privileges as the host. The guest code can for example access the *DOM*, perform network requests, redirect the host page, and so on. Hence, it becomes crucial to restrict the power of the guests, in order to cope with unauthorised actions of malicious or compromised services.

In this report, we discuss two different approaches towards securing web mashups: *Caja* and *JSFlow*. The first, is a *sandboxing* mechanism that creates a safe execution environment for the host through the use of a security model known as *object-capability-model*[2]. The latter builds from the perspective that *access control* can be circumvented. And introduces means to prevent the leak of sensitive information through an extended JavaScript interpreter known as *JSFlow*[3].

The remaining of the report is organised with the intention to clarify and compare both approaches. Section 2 introduces Caja and several use examples. Section 3 focus around *JSFlow* and test examples. Finally, section 4 provides a brief comparison of both.

2 Caja

2.1 Object Capability Language

Essentially, Caja (*capabilities attenuate JavaScript authority*) is the name given by the authors to a restricted JavaScript subset[2]. In this subset, some JavaScript constructs are deemed to be dangerous and no longer allowed (e.g. *eval* and *with*). The obtained subset is called Caja and is intentionally chosen to adhere to a security model known as *object-capability-model*.

In an *object-capability-model*, objects are isolated from each other. This with the intention to prevent them from affecting each other and their surroundings[2]. Additionally, the model suggests the use of *object references* as a way to grant authority to isolated objects. An object reference is simply a reference to an object that contains custom and restricted functionality. So that only references holders are able to use such functionality[2]. Hence, by defining object references and controlling who has access to it, the model limits the way how isolated objects affect each other and their surroundings. And consequently, limit their power to an essential minimum.

Regarding mashups security, developers use Caja and several helper components to enforce the model at client-side. More precisely, developers *sandbox* third-party content[4] and define for them object references with restricted authority. A process known as *taming objects*[2]. In the remaining, we briefly overview the helper components, as-well as, their role in the enablement of the model.

The enablement of the object-capability model involves the collaboration of two components namely *cajoler* and *caja.js* [2, 4]. The *cajoler* is a helper component that resides at the server side and is responsible for code transformation[2]. For a given program, the cajoler either: 1) rejects the program if it does not adhere to Caja constructs. Or 2) rewrites the program into a capability safe version of it, that is, a program that contains extra code required to enforce the model at runtime[2].

The second component, *caja.js*, is a JavaScript module loaded into client interpreter that gives access to a singleton object called *caja*. The object defines an API that permits to sandbox

guest content, as-well as, create custom object references. The module also enables a capability safe runtime environment enforced on cajoled programs.

To illustrate how a caja based web page is structured consider figure 1. The web page is only meant to illustrate a basic setup and in no means implements meaningful content. The numbers correspond with key aspects of caja and are further discussed.

Note that, setup figure 1 is also the one that we follow in examples of section 2.2. Additionally, for the well-functioning of the solution, the module caja.js and cajoler program need to be accessible i.e. stored in the app server. For simplicity, we use the resources provided by Google servers[1].

```
<head>
    <!-- (1) Load caja module-->
    <script type="text/javascript" src="http://caja.appspot.com/caja.js"></script>
</head>
<body>
    <div id="guest"></div>
    <script type="text/javascript">
        //(2) setup cajoler server
        caja.initialize({ cajaServer: 'http://caja.appspot.com/'})
        //(3) configure caja with a DOM element and network policy
        var dom = document.getElementById("guest");
        var np = { rewrite: (uri)=> { /*policies*/ }}
        caja.load(dom, np, (frame) => {
            //(4) Tame objects

            //(5)Cajole guest and provide API
            frame.code(guestURL, 'text/html')
                .api(/*API for guest*/)
                .run()
        })
    </script>
</body>
```

Figure 1: A basic setup to enable a caja based solution at client-side

In figure 1, (1) illustrates how the *caja.js* module is retrieved through script inclusion. Once the module is loaded, the singleton object is available, which allows execution of the remaining code. In (2), we configure the singleton object with the address of the cajoler server. In (3), we tell caja where to insert the guest content (*dom*) and what network policies (*np*) to apply on the guest. The *dom* and *np* can also be set to *undefined* to respectively indicate that the guest is not embedded nor has access to the network. In (4), we insert code that creates *tamed objects*[2], that is, objects with limited and customised authority. For simplicity, the taming code is omitted and left for later discussion. Finally, (5) asks the cajoler to cajole the guest content (retrievable from address *guestURL*) and to use the tamed objects as allowed API to the guest.

[1]Module *caja.js* can be retrieved through a script inclusion where the src attribute is set to `http://caja.appspot.com/caja.js`.
The *cajoler* is enabled through configuration option on the singleton *caja* object and can be set to `http://caja.appspot.com/`

2.2 Caja Test Examples

At the end of the previous section, we saw how to create a basic setup in order to use a caja based solution (figure 1). In this section, we continue building upon the basic setup, that is, we focus on the taming process and how it helps achieve some security goals. For this, we introduce several examples discussed in the remaining of the section. To follow along, please consult installation guide *README*.

For the following examples, we start three different web applications: *host*, *guest1* and *guest2*. The *host* is a web mashup that loads services from the two others. Depending on the example, the host either uses services of both or not. And depending on the service used, we impose different security conditions.

Complete Isolation

In this example, guest1 is a joke service used by host to display a joke to its end users. Guest1 provides the joke in HTML format, which is inserted into a specific DOM element inside the guest. The host knows that the guest does not require any JS in order to function correctly. The host can therefore safely cajole guest1 and provide it zero authority. Figure 2a illustrate the caja code needed for this.

```
const EMBED = document.getElementById('guest')
const NO_NETWORK = undefined;
const guestURL = 'http://localhost:3030/caja/example1_guest.html'
caja.load(EMBED, NO_NETWORK, (frame) => {
    frame.code(guestURL, 'text/html').run()
})
```

Figure 2a: caja configured to cajole guest *guestURL* without network access and embed it in DOM element *EMBED*

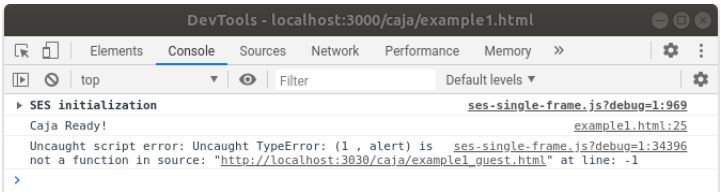

Figure 2b: developer tool message warning about the execution of an unexisting function called *alert* from whitin guest content *example1_ guest.html*

The first two arguments of *load* (figure 2a), respectively tell caja where to insert the cajoled content (*EMBED*) and the policy around network access. In our situation, we set the policy to *undefined* which means that no network access is allowed. Additionally, to show that the isolated content has no authority, the guest attempts to access *alert* function, which is rejected by Caja (figure 2b). The developer tool logs an error due to the use of a non-existent function *alert*. This is because we did not provide the guest with the capability to use such function.

Note that if the same code is executed whitin an *IFrame*, the fail on the *alert* function does not occur. The IFrame manages to isolate the given content but does not impose any restrictions upon functionality[1]. In our situation, when using IFrame, the joke service can access previously rejected function and other functionality.

Interaction with the host - Network disabled

This example illustrates the ability to define a controlled collaboration between the guest code and the host. The host uses the guest for its ability to determine passwords strengths. More specifically, the guest is a JavaScript function that for a given password checks whether the password has a certain length. The guest also provides feedback at each password verification. And when the password is strong, it attempts to leak it through an *XMLHttpRequest* (see *caja/example2_ guest.js*).

From the perspective of the host, the guest has no reason to access the network and only requires access to the password and a DOM element. In caja, this translates to an *undefined* network policy, as-well as, a tamed object (called *pswdService*) that provides the guest with the password an the ability to write in the DOM (figure 3a). From *pswdService*, the guest uses methods *register* and *feedback* to respectively register a callback and display password feedback into the host. The callback is invoked by the host each time a new password is available.

```
var listener;
var pswdService = {
    register: (l) => { listener = l; },
    feedback: (fb) => {
        document.getElementById("feedback").innerHTML = fb
    }
}
caja.markReadOnlyRecord(pswdService);
caja.markFunction(pswdService.register);
caja.markFunction(pswdService.feedback);
var tamedService = caja.tame(pswdService);
```

Figure 3a: object *pswdService* is an API object that first undergoes a taming process before providing it to the guest

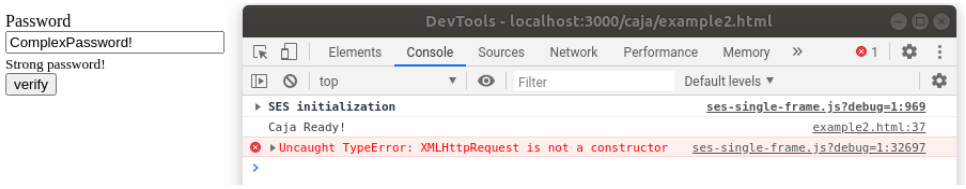

Figure 3b: On the left: an input field containing a user password and the button that triggers the password strength verification. On the right: developer tool message informing about the attempt to construct a non-existent object called *XMLHttpRequest*.

Before we provide *pswdService* object to the guest, we first need to *tame* it i.e. tell caja what entries of the object are usable and how to restrict access on it. For this, we use *markReadOnlyRecord* to mark *pswdService* as only readable content and *markFunction* to enable access of both functions. Then, we trigger the taming process by invoking method *tame* on object *pswdService*. At this point, the taming result can safely be given to the guest, which extends it with the ability to use both functions (*register* and *feedback*). Hence, any other operation is disallowed to the guest. This is for example illustrated in figure 3b, where caja successfully prevents the leak of the password.

```
const MAX_REQUESTS = 2
var requests_made = 0
const NETWORK_POLICY = {
    rewrite: (uri) => {
        if (requests_made < MAX_REQUESTS &&
            /http:\/\/localhost:3030/.test(uri)) {
            requests_made += 1;
            return uri;
        }
        return undefined
    }
};
```

```
var hostAPI = {
    newXtr: () => {
        if (requests_made < MAX_REQUESTS) {
            requests_made += 1;
            return new XMLHttpRequest();
        }else{
            logOnDom("XMLHttpRequest failed");
        }
    },
}
```

Figure 4: On the left: a network policy that only allows two network requests to the same origin. On the right: object API tamed and provided to the guest to give it access to *XMLHttpRequest* objects.

Controlled Network Access

In this example, we show that it is possible to give limited network access to a guest. For this example, we reuse previous joke service and allow it to perform two network requests. The first, to retrieve a joke and the second to retrieve an image.

The host introduces a network policy specially crafted to fit such requirement (figure 4 left). The network policy is invoked each time the guest access the network and its result determines whether the request is accepted. In our situation, a request is accepted if 1) the current request does not exceed the maximum amount of allowed requests and 2) the request is towards the guest domain. Additionally, in order to perform network requests, we also provide the guest with the capability to create *XMLHttpRequest* objects. For this, we tame object *hostAPI* (figure 4 right) and provide it to guest as API. Method *newXtr* returns *XMLHttpRequest* objects if the maximum amount of allowed requests is not exceed.

To show the restricted network access, we trigger a third network requests in the guest code, which is successfully detected and prevented. This thanks to the branch conditional of the if statement in *hostAPI*. Note that in the host page, the error is illustrated with a red font.

Services Collaborate

In this example, we show that it is possible to support service collaboration in a controlled manner. In our situation, the host uses two services to help a user create a unique post title. The first service (*verify service*) verifies whether a given post title is already in use. If so, it displays a message indicating this and asks the second service (*suggestion service*) to propose a title. The *verify service* also provides a number to the *suggestion service* that represents the amount of verification attempts made until now. When asked, the *suggestion service* displays a title and the received number. The proposed title supposedly inspires the user in finding an unique title. The guests of the resulting host page are displayed in figure 5a.

To define the communication between both guests, we introduce a specifically designed object that mediates their communication (figure 5b). The object is tamed as previous examples and introduces three methods as API. The *quantityListener* is only provided to suggestion service and is used to register a callback. The *setQuantity* is only provided to verify service and used to communicate the quantity of verification attempts. The third method is discussed below.

At first glans, both services work as expected. The verification service displays an informative message and shares the verification quantity to the suggestion service. The latter, displays the

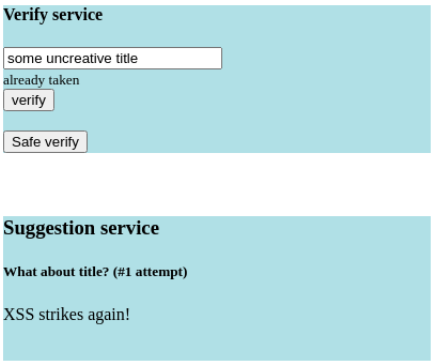

Figure 5a: Two guests services cajoled whitin the host page. On the bottom: the *suggestion service* that suggests a potential title and displays the quantity of attempts. On top: the *verify service* that for a given title supposedly verifies the title existence. Top button uses *setQuantity* and lower button *setQuantitySafe* of object in figure 5b. Top button leaks the title at the second attempt.

```
var shared = {
    cb: null,
    quantityListener: (cb_) => {
        this.cb = cb_
    },
    setQuantity: (t_) => { this.cb(t_) }, //unsafe
    setQuantitySafe: (t_) => {
        if (typeof t_ === 'number') {
            this.cb(t_)
        }
    },
};
```

Figure 5b: object used to control communication between both guest services. The use of function *setQuantity* allows for unsafe communication.

quantity along with a suggestion title (figure 5a). However, after the second verification, the verification service shares an object containing a title instead of the regular quantity. This a problem if we only want them to share the quantity. And this is specially a problem if the title is sensitive content. The issue is related to method *setQuantity*. More specifically, the method does not verify what content is being communicated, which allows the leak of the title. A safe alternative to *setQuantity* is *setQuanitySafe*, which is triggered when clicking on *"safe verify"* button. The improved method successfully prevents the share of non-number content by simply controlling what kind of content is being exchanged.

The intentional flaw of *setQuantity* is meant to illustrate that disposing of cajoled content is not enough to design a secure mashup. Developers, still need to be careful in how they design the authority meant for cajoled contents.

3 JSFlow

3.1 Dynamic Information Flow Control

JSFlow builds from the perspective that any access control mechanism (including caja) can be circumvented [3, 5]. Therefore, any security guarantee enforced by the access control mechanism is no longer valid. The reason is that access control prevents the access of some content as long as it is enabled. Once circumvented, there is no mechanisms that further protects the sensitive information. With this in mind, Heding et al.[3] propose *JSFlow* i.e. an extended JS interpreter that enforces *secure information flow* at runtime. The idea behind the interpreter is to identify sources of sensitive information, track the flow of that data, and if necessary prevent operations upon them[3, 5]. For this, the interpreter heavily relies on the use of a *dynamic information flow analysis*. In such analysis the information flow is tracked at runtime, which fits good in the dynamic nature of JS[3]. Additionally, each encountered value is associated to a *security label*. The security label defines a degree of confidentiality imposed upon the value. For example, a value marked as *public* is not as sensitive as a value marked as *secret*. Hence, when JSFlow evaluates operations upon values, the associated label determines whether the operation is allowed or not. Furthermore, security labels are also customizable i.e. users are allowed to associate self-defined labels to newly declared values. This allows to define user specific security policies around information access[3, 5]. In section 3.2 this aspect of JSFlow is further explored.

In general, JSFlow is responsible for the correct update and propagation of data labels. For this, it has to face several challenges and complexities caused by the dynamic nature of JavaScript[6]. While the remaining explores some issues, section 3.2 focus around specific use cases of JSFlow.

Explicit Flow
If we assume the existence of variables $x = 2$ and $y = 3$, respectively labelled as *public* and *sensitive*. An assignment of the form $x = 2 + y$ should not result in variable x labelled as *public*, but instead marked as *sensitive*. Otherwise, it becomes very easy to leak sensitive information through an assignment. This example, demonstrates that an analysis has to be sensitive to *explicit flows* of data[3].

Implicit Flow
Assume the following code line (retrieved an adapted from the authors [3]) **var x=0; if (y) x=1;**, where y is a sensitive boolean and x is public. Even-thought the content of y is not explicitly assigned to x, we still manage to derive it through control flow. If at the end of this computation, x is 0, y must have been false, otherwise true. This example, shows that JSFlows also has to be sensitive to *implicit flows* i.e. information flow dependant on control flow[3].

To solve this problem, the authors introduce an extra security label called *program counter* (pc). The pc is changed at each branching condition and set to the same confidential degree as the guard expression[3]. In the previous example, when JSFlow reaches the sensitive branching condition y, it assigns to *pc* a *sensitive* label. So that, when the assignment is performed, JSFlow rejects the operation by terminating the program.

Non-Interference & TINI
In general, information flow control solutions should satisfy a security property known as *noninterference*[3]. The property declares that public output should not depend on sensitive content. Meaning that, if the sensitive content is the only content changed across all executions of a program, the public content should remain the same across all executions[7]. Hence, public output

does not leak sensitive content in presence of *noninterference*.

To enable *noninterference* JSFlow employs *TINI* (termination insensitive noninterference), which is, a security model that ensures *noninterference* for programs that terminate execution[7]. Unfortunately, TINI does not prevent leak of sensitive data for programs that do not terminate execution. For example, consider following listing retrieved from the authors tutorial page[7].

```
1  /* Assuming h is a secret variable of type Number defined elsewhere.
2   * Also assume the attacker can observe the output done by print. */
3
4   for(var i = Number.MIN_VALUE; i <= Number.MAX_VALUE; i++) {
5      print(i);
6      if( i == h) {
7          while (true){}
8      }
9   }
10 }
```

Listing 1: A non-terminating program that leaks sensitive data when stuck in the while-loop [7]

In this example, it is possible for an attacker to leak the information through the print statement (line 5). The approach is to create a public variable i that increments from the minimum to the maximum possible value a number can have. At each increment, the attacker prints the number (allowed because the number is public) and then tests for equality with sensitive integer h (line 6). If equal the program jumps into a while-loop that never ends. Hence, the last printed integer is equal to the sensitive integer h.

3.2 JSFlow Test Examples

As intended by the authors, JSFlow can be used as a Firefox extension[8, 3]. The extension successfully replaces the Firefox JavaScript engine with their custom interpreter and this to control information flow on web pages[3].

In the following, we use the Firefox extension to enable IFC on two different test examples. Similar to Caja, the examples introduce a host (i.e. a web mashup) that uses functionality provided from guests. To follow along, please consult *README* installation guide.

No Network Leak

In this example, we reconsider the password service as introduced in section 2.2. The service assists a user in determining whether his password is strong enough (figure 6a). In our case, the guest is a JS function that for a given password checks whether the length is greater than a minimum. Furthermore, the service provides feedback on the strength of the password and attempts to leak the password through an image tag.

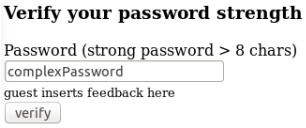

Figure 6a: the input field that captures a user password and the *verify* button that triggers password verification guest code.

When a user inserts a strong password and then clicks on the *verify* button. The user is presented with a dialog box as illustrated in figure 6b. The dialog box warns the user about

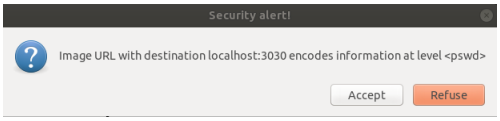

Figure 6b: Dialog box emitted by JSFlow to warn about the leak attempt of the password through an *image URL*

an attempt to leak sensitive content through an image URL. At this point the user can either allow or reject such operation. The dialog box illustrates the successful detection of potential dangerous operations on sensitive content. The reason why JSFlow manages to detect the invalid operation, is because the authors consider all content coming from an input field as sensitive [3]. Hence, when the password value reaches the public image request the dialog box is triggered.

Note that in the source code (*jsflow/example1.html*), we also associate a custom label *pswd* to the password value. This is possible thanks to newly introduced JSFlow construct *lbl*. The construct helps to define custom security policies through the introduction of custom labels. Even-though in this example the *pswd* label was not necessary in preventing the leak of the password. It is still interesting to know that *lbl* exists.

Partial Data Leak

In this example, we reconsider the *joke service* and allow it to access public information defined for a particular user i.e. username *user123*. The username is used by the guest to generate a custom joke and to display it into the host page. Additionally, the guest is also supposed to retrieve an image and insert it into the host page (which fails for reasons mentioned below). The host also contains the email address of the user i.e. *user123@jsfow*, which is assumed to be sensitive, thus, not accessible to the guest.

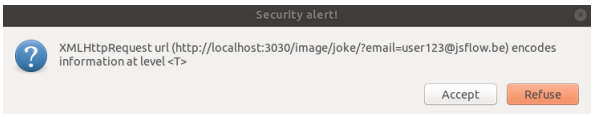

Figure 7: Dialog box emitted by JSFlow to warn about the leak attempt of the email through an *XMLHttpRequest* object

To trigger the joke service, the host invokes guest function *loadJokeService* (see *jsflow/example2_guest.js*) with three arguments. The arguments respectively provide to the function: 1) a DOM id on where to insert the joke, 2) a DOM id on where to insert an image and 3) the username. Using the three arguments, the guest performs two different network requests. The first retrieves the custom joke and inserts the result in the DOM. The latter retrieves the image, but fails due to a security policy violation. More precisely, the guest attempts to leak sensitive content (i.e. email) while performing the network request. This is detected by JSFlow and results in an alert box. Equally to previous example, the user can either allow or reject such network request (figure 7).

4 Strengths & Limitations

In this section, we briefly compare Caja and JSFlow based on their strengths and weaknesses. One issue with Caja is the restrictions that it imposes on the integration of third-party services[4]. More precisely, the allowed Caja subset forces third-party services to adapt their existent/future content. In contrast JSFlow, manages to apply their IFC monitor without the need for a rewrite. As reported by Hedin et al. [3], JSFlow tracks information flow in several third-party services (e.g. *google analytics*) and even discovers unauthorised leaks[3].

In the few examples that we saw, Caja requires more lines of code to achieve the same security outcome as JSFlow. In the password example, we tamed an object and disabled network access to enable safe use of the password service. In JSFlow, the same is achieved with no effort. This because, the input fields are considered to be sensitive by design [3]. Hence, no additional code is needed to prevent their leak.

The code difference becomes more obvious in the example where we restrict the joke service to two network requests. The *newXtr* method actually results in a tamed *XMLHttpRequest* object. The tamed object exposes to the guest, only *XMLHttpRequest* properties (e.g. *readyState*, *ready*, etc.) needed to perform the network request. On the code level, this means that we have to perform one tame method invocation per used *XMLHttpRequest* property. In summary, the taming of the *XMLHttpRequest* object illustrates that the problem becomes worse once we integrate more advanced libraries (e.g. *JQuery*).

As mentioned, JSFlow builds from the perspective that access control can be circumvented[3]. And once circumvented, previously unauthorised resources are once again usable. This problem is demonstrated by Michal Bentkowski[9]. He shows that it is possible to escape Caja through the exploit of a new language construct. More precisely, when the *AsyncFunction constructor* was introduced i.e. *Object.getPrototypeOf(async function(){}).constructor* [10]. Caja did not yet take the construct into account. Resulting in an exploitable scenario as illustrated in figure 8b. The attack code is shown in figure 8a.

```
<script>
    <!--
    (async function(){}).constructor('alert(1)')();
  </script>
```

Figure 8a: Guest content that enables Caja escape through async constructor exploit[9]

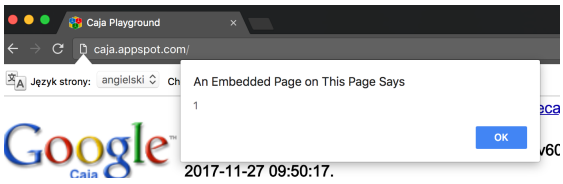

Figure 8b: Successful use of unauthorised alert function by a supposedly cajoled guest. Exploit possible thanks to code in figure 8a [9]

Compared to Caja, JSFlow is certainly not production ready. As observed by Scull Pupo et al.[11], JSFlow is bound to a certain JavaScript version and browser environment. Which certainly impacts their applicability in a more realistic production setting[11].

References

[1] P. De Ryck, M. Decat, L. Desmet, F. Piessens, and W. Joosen, "Security of web mashups: A survey," in *Information Security Technology for Applications*, T. Aura, K. Järvinen, and K. Nyberg, Eds., Berlin, Heidelberg, 2012, pp. 223–238.

[2] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Safe active content in sanitized javascript," *Google, Inc., Tech. Rep*, 2008.

[3] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "Jsflow: Tracking information flow in javascript and its apis," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 1663–1671.

[4] S. Van Acker and A. Sabelfeld, *JavaScript Sandboxing: Isolating and Restricting Client-Side JavaScript*. Cham: Springer International Publishing, 2016, pp. 32–86. [Online]. Available: https://doi.org/10.1007/978-3-319-43005-8_2

[5] D. Hedin and A. Sabelfeld, "Web application security using jsflow," in *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, 2015, pp. 16–19.

[6] D. Hedin, L. Bello, and A. Sabelfeld, "Information-flow security for javascript and its apis," *Journal of Computer Security*, vol. 24, no. 2, pp. 181–234, 2016.

[7] "Tutorial," http://www.jsflow.net/tutorial.html, accessed on: 2020-08-12 [Online].

[8] J. Magazinius, D. Hedin, and A. Sabelfeld, "Architectures for inlining security monitors in web applications," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2014, pp. 141–160.

[9] "Yet another google caja bypasses hat-trick," https://blog.bentkowski.info/2017/11/yet-another-google-caja-bypasses-hat.html, accessed on: 2020-08-12 [Online].

[10] "Asyncfunction," https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/AsyncFunction, accessed on: 2020-08-12 [Online].

[11] A. L. S. Pupo, L. Christophe, J. Nicolay, C. De Roover, and E. G. Boix, "Practical information flow control for web applications," in *International Conference on Runtime Verification*. Springer, 2018, pp. 372–388.