Biting the hand that serves you: A closer look at client-side Flash proxies for cross-domain requests

Martin Johns and Sebastian Lekies[⋆]

SAP Research Karlsruhe
`(martin.johns|sebastian.lekies)@sap.com`

Abstract. Client-side Flash proxies provide an interface for JavaScript applications to utilize Flash's cross-domain HTTP capabilities. However, the subtle differences in the respective implementations of the same-origin policy and the insufficient security architecture of the JavaScript-to-Flash interface lead to potential security problems. We comprehensively explore these problems and conduct a survey of five existing proxy implementation. Furthermore, we propose techniques to avoid the identified security pitfalls and to overcome the untrustworthy interface between the two technologies.

1 Introduction

Over the period of the last decade, Web applications have exposed an ever growing emphasis on sophisticated client-side functionality. In many places, the request-response-render Web of the early days has made way for rich JavaScript clients that utilize AJAX-driven communication and dynamic user interfaces. However, as the evolution of the Web browser's native capabilities did not always kept pace with the rapid innovation of the application's demands, plug-in technologies, such as Adobe Flash [1] filled the gap and provided advanced features which were missing in the browsers.

However, the security policy implemented by browser plug-ins does not always exactly match the security model of the Web browser. In certain areas, already subtle deviations can lead to security implications which are hard to handle.

In this paper we examine how the difference in handling the same-origin policy in respect to cross-domain JavaScript and cross-domain Flash lead to unexpected consequences. For this purpose, we explore the field of client-side Flash proxies for cross-domain requests. After covering the technological basis of client-side HTTP requests (Sec. 2), we explore a little known security flaw that can occur when cross-domain Flash applets interact with adversary controlled JavaScript (Sec. 3). To substantiate our observation, we examine five

[⋆] This work was in parts supported by the EU Project WebSand (FP7-256964), http://www.websand.eu.

publicly available client-side Flash proxies (see Sec. 3.3). Furthermore, we show how to overcome the identified issues. First, we solve the specific problem of providing a secure and flexible client-side Flash proxy (Sec. 4). Secondly, we tackle the general problem of implementing Flash-based functionality that reliably makes origin-based security decisions even when interacting with untrustworthy JavaScript (Sec. 5). After discussing related work (Sec. 6) we finish with a conclusion (Sec. 7).

2 Client-side cross-domain HTTP requests

2.1 Technical background

In general, the same-origin policy (SOP) [26] is the main security policy for all active content that is executed in a Web browser within the context of a Web page. This policy restricts all client-side interactions to objects which share the same origin. In this context, an object's origin is defined by the URL, port, and protocol, which were utilized to obtain the object. While this general principle applies to all active client-side technologies (e.g., JavaScript, Java applets, Flash, or Silverlight), slight variations in the implementation details exist. Please refer to [35] for further reference.

Based on the SOP, the initiation of network connections from active content in the browser is restricted to targets that are located within the origin of the requesting object[1]. This means, a JavaScript that is executed in the browser in the context of the origin `http://www.example.org` is only permitted to generate HTTP requests (via the XMLHttpRequest-object [31]) to URLs that match this origin. The same general rule exist for Flash, Silverlight, and Java.

However, in the light of increasing popularity of multi-domain, multi-vendor application scenarios and ever growing emphasis on client-side functionality, a demand for browser-based *cross-domain* HTTP requests arose (e.g., in the field of Web2.0 Mashups), as such request have the ability to mix data and code from more then one authorization/authentication context (see Sec. 2.2).

Following this demand, the ability to create cross-domain HTTP requests from within the browser has been introduced by Adobe Flash [1]. To avoid potential security implication (see Sec. 2.5), the designated receiver of the HTTP request has to explicitly allow such requests. This is done through providing a `crossdomain.xml`-policy file [2] which whitelists all domains that are allowed to send browser-based cross-domain HTTP request.

Following Flash's example, Silverlight and Java have also introduced capabilities for browser-based cross-domain HTTP requests. To enforce similar 'recipient-opt-in' policies as in Flash's model, Silverlight and Java imitate the `crossdomain.xml` mechanism of requiring policy files.

[1] NB: This restriction only applies to HTTP requests that are created by active code. HTML elements, such as `IMG` or `IFrame` are unaffected and can reference cross-domain objects.

2.2 Use cases for client-side cross-domain HTTP requests

The need for client-side cross-domain requests is not immediately obvious. Alternatively, the Web application could offer a server-side proxying service that fetches the cross-domain content on the server-side, instead of requiring the client-side browser to issue the cross-domain requests. Such a service can be accessed by the client-side code via standard, SOP-compliant methods. This technique is offered for example by Google's Gadget API [11].

However, this technique is only applicable in cases in which the Web application's server-side code is able to access the requested content, e.g, when requesting publicly available internet resources.

Server-side proxies might not be applicable whenever the access to the requested content is restricted. Examples for such scenarios include situations in which the requested content is unavailable to the server because of network barriers (e.g., an internet Web application interacting with intranet data) or cases in which the demanded information is only available in the current authentication context of the user's Web browser (e.g., based on session cookies, see Sec. 2.5 for details). Such use cases can only be realized with client-side cross-domain capabilities.

2.3 The current move towards native browser capabilities

After a period of perceived stagnation of Web browser development and innovation, the recent years have shown rapid progression of browser technologies. This movement was initially lead by the Web Hypertext Application Technology Working Group (WHATWG [33]) and has now been picked up by the W3C and IETF.

A significant part of the work of these groups (and associated Web browser vendors) is the adoption of capabilities, that have successfully been introduced by browser plug-in, and their addition to the browser's native HTML/JavaScript functionality. Examples for such capabilities include audio/video playback and expanded networking capabilities.

This development leads to the prediction that in the future such native capabilities will reduce the current dependency on browser plug-ins. First indicators of this trend are the mobile Web browsers of the iOS and Windows Phone Seven platforms that completely rely on native technologies and do not provide support for browser plug-ins.

In the course of expanding JavaScript's networking stack, the ability to create cross-domain HTTP requests have been added. JavaScript's native implementation is called Cross-origin Resource Sharing (CORS) [32]. CORS utilizes JavaScript's XMLHttpRequest-object for this purpose. Instead of following Flash's example of utilizing policy files, CORS utilizes HTTP response headers to allow or deny the requests. Only if the HTTP response carries an allowing HTTP header, the received data is passed on to the calling script. This behavior allows much more fine-grained access control compared to the established policy-based mechanisms.

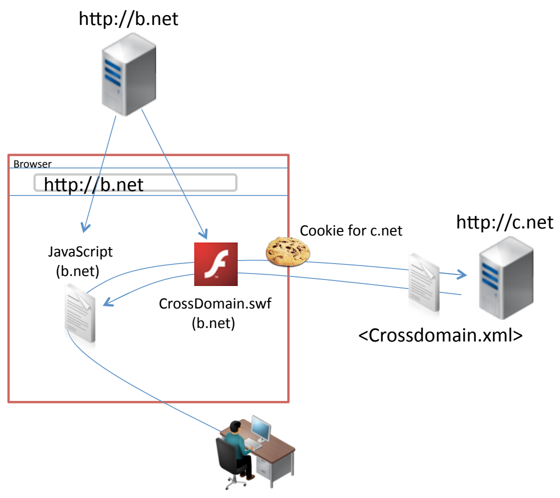


Fig. 1: Client-side cross-domain Flash proxy

2.4 Client-side cross-domain Flash-proxies for legacy browsers

Given the move towards native JavaScript capabilities in modern browsers and the expected fading importance of browser plug-ins (see above), it becomes increasingly important for Web applications to support CORS for scenarios which utilize client-side cross-domain requests. As a consequence, for the time being developers will have to implement two variants of the same cross-domain request functionality in parallel: A version that utilizes Flash or Silverlight for legacy browsers that do not yet provide CORS and a CORS version for modern browsers that either do not support plug-ins, such as most mobile browsers, and for browsers in which plug-ins have been disabled (e.g., for security reasons). Based on experiences with the longevity of old browser variants, such as Internet Explorer 6, it can be expected that this transitional phase will last a considerable amount of time.

To ease this development, client-side Flash-proxies have been introduced which export Flash's cross-domain capabilities to JavaScript (see Fig. 1). These proxies are small, single-purpose Flash applets combined with a small JavaScript library that together provide an interface to the enclosing page's JavaScript for initiating HTTP requests to cross-domain targets and passing the corresponding HTTP response back to the calling script. Figure 2 shows the interaction pattern between the JavaScript library and the Flash applet.

To further ease the use of these proxy libraries, extensions for popular JavaScript frameworks, such as JQuery or Dojo, are provided that import the functionality into the respective framework. In most cases this is done in a fashion that makes its usage almost transparent to the developer, i.e., through exchanging the framework's networking methods with the corresponding functions of the proxy library.

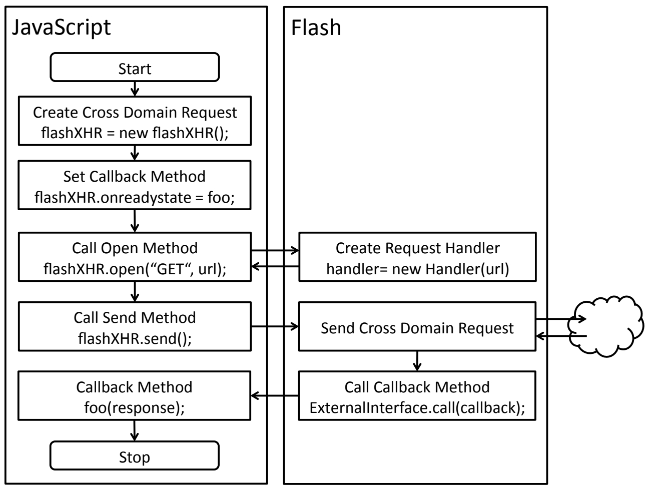

JavaScript
Flash
Start
Create Cross Domain Request
flashXHR = new flashXHR();
Set Callback Method
flashXHR.onreadystate = foo;
Call Open Method
flashXHR.open("GET", url);
Create Request Handler
handler= new Handler(url)
Call Send Method
flashXHR.send();
Send Cross Domain Request
Callback Method
foo(response);
Call Callback Method
ExternalInterface.call(callback);
Stop

Fig. 2: Interaction between JavaScript and the Flash proxy

In a survey we were able to identify five different published Flash proxy libraries which provide the described functionality. See Table 3.3 for a brief overview.

2.5 Security implications of client-side cross-domain HTTP requests

Before we discuss the potential security problems introduced by client-side Flash proxies in Section 3, we briefly revisit the general potential security problem that can arise in connection with client-side cross-domain HTTP requests.

Attacker model: From now on, we consider the following scenario: The victim visits a Web site `a.net`, which is under the control of the attacker. This Web site is granted the right to create cross-domain requests to a second Web site `b.net`, for instance because `b.net` has an over-allowing crossdomain.xml policy file that whitelists ("*") all foreign domains. Furthermore, the victim currently possesses an authenticated session state with `b.net`, i.e., in the victim's browser a session cookie for this domain exists.

This setting allows the attacker to create arbitrary HTTP requests to `b.net` from within the victim's browser and read the corresponding HTTP responses. The victim's cookies for `b.net`, including the authenticated session cookies, are attached to these requests automatically by the browser, as the cookies' domain matches the target domain of the outgoing HTTP request. These requests are received and handled by `b.net` in the same fashion as regular requests coming from the victim, hence, they are interpreted in the victim's current authentication context.

Resulting malicious capabilities: We can deduct the several potential attack vectors, based on the scenario discussed above.

Leakage of sensitive information [13]: The adversary can obtain all information served by `b.net` which the victim is authorized to access by simply requesting the information via HTTP and forwarding the corresponding HTTP responses to the attacker's server.

Circumvention of Cross-site Request Forgery protection [27]: The security guarantee of nonce-based Cross-site Request Forgery (CSRF) protection [7] is based on the assumption that the attacker is not able to obtain the secret nonce which is required for the server to accept the request. These nonces are included in the HTML of `b.net`. As the adversary is able to read HTTP responses from `b.net`, he can request the page from `b.net` that contains the nonce, extracting the nonce from the page's source code, and using it for the subsequent HTTP request.

Session hijacking [23]: As discussed, the adversary has the ability to create arbitrary HTTP requests that carry the victim's authentication credentials and read the corresponding HTTP responses. In consequence, this enables him to conduct attacks that are, for most purposes, as powerful as session hijacking attacks which are conducted via cookie stealing: As long as the targeted application logic remains in the realm of the accessible domain (in our case `b.net`), the attacker can chain a series of HTTP requests to execute complex actions on the application under the identity of the user, regardless of CSRF protection or other roadblocks. This can happen either in a fully automatic fashion, as for instance XSS worms [18, 21] function, or interactive to allow the attacker to fill out HTML forms or provide other input to his attacks. Frameworks such as BeEF [5] or MalaRIA [23] can be leveraged for the interactive case.

3 Abusing client-site cross-domain Flash proxies

In this section, we explore a little known[2] security problem that is caused by the way Web browsers handle the SOP in respect to cross-domain Flash applets. Hence, from now on, we limit the scope of the discussion to JavaScript and Flash.

3.1 Subtle differences in the SOP

Both JavaScript files and Flash applets can be included in foreign Web pages of arbitrary origin due to the HTML's transparent handling of cross-domain locations in tags, such as `script`, `object`, or `embed` (see Listing 1). However, the specifics how the SOP is applied to such cross-domain active content differs [35]:

Cross-domain JavaScript code inherits the origin of the enclosing domain. It is handled as if it was a native component of the enclosing Web page and looses all ties to its original origin.

[2] During investigating the topic, we only encountered one single blog post discussing the issue [29]. Besides that, there appears to be no awareness.

Listing 1 Cross-domain inclusion of script and Flash code

```
<!-- HTML source of a Web page served by a.net -->
[...]
<!-- cross-domain JavaScript -->
<script src="http://b.net/somescript.js" />

<!-- cross-domain Flash -->
<object [...]>
  <param name="movie" value="http://b.net/flash_proxy.swf" />
  <param name="quality" value="high" />
  <embed src="http://b.net/flash_proxy.swf" [...]></embed>
</object>
```

Opposed to this, cross-domain Flash applets keep the origin from which the Flash's swf-file was retrieved. As a consequence, all security relevant actions are restricted or granted in the context of the Flash's original origin. In particular, this means that a browser's decision if an outgoing HTTP request will be permitted, is made based on checking the destination URL of the request against the original origin of the Flash and **not** based on the origin of the enclosing Web page.

Consequently, in cases in which a Flash applet provides a public JavaScript interface, this interface can be used by scripts running in the context of the enclosing Web page to cause actions which are executed under the cross-domain origin of the Flash applet.

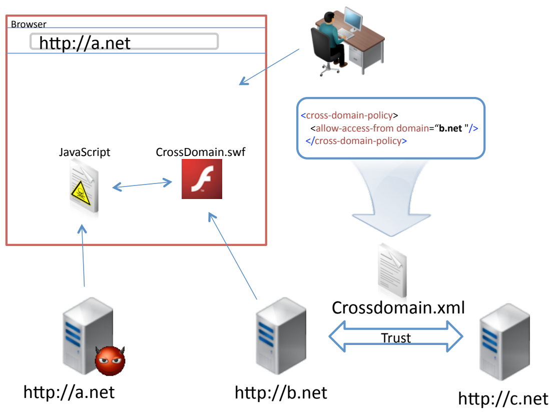
Please note: To provide a public JavaScript interface which can be abused in the outlined fashion, the Flash applet has to release the exported methods for external domains using the `allowDomain()`-directive [4]. The value of this directive is a list of domains which are permitted to access the declared interface. However, in the case of general purpose Flash libraries it is common practice to whitelist all domains (using the wildcard mechanism `allowDomain("*")`), as at the time of development the programmer cannot know on which domains the library will be used (see also Sec. 3.4).

In the remainder of this section we explore attack vectors that result from the consequences of the mismatching SOP implementations in the context of client-side Flash proxies.

3.2 Attack vectors

Scenario: In this section we consider a scenario that is similar to the general misuse case which was discussed in Section 2.5. The victim accesses a Web page from the domain `a.net` which is under the control of the attacker. Hence, the adversary is able to execute JavaScript in the context of `a.net` and import content from other domains via HTML tags, including cross-domain Flash applets.

In addition to `a.net`, two more domains exist: `b.net` and `c.net` (see Fig. 3). The application hosted on `c.net` exports cross-domain services to `b.net` and, hence, provides a `crossdomain.xml` policy file that allows requests coming from `b.net`. To access these services, the application running on `b.net` utilizes a client-side Flash proxy, as described in Section 2.4.



```
<cross-domain-policy>
  <allow-access-from domain="b.net "/>
</cross-domain-policy>
```

Fig. 3: Scenario

Attack variant 1 - **Transitivity of trust:** The adversary's script on `a.net` can utilize `b.net`'s Flash proxy to create HTTP requests. Requests to locations that whitelist `b.net` in their `crossdomain.xml` policy are permitted, as the browser's security decision is made in respect to the origin of the Flash. This way, the attacker can conduct the attacks outlined in Section 2.5 targeting `c.net`, even though he has no direct control over the domain `b.net` (see Fig.4).

Attack variant 2 - **Return to sender:** In addition to creating requests to all domains that list `b.net` in their `crossdomain.xml` policies, the adversary is also able to make requests from `a.net` to `b.net` itself, even in cases in which `b.net` does not provide a `crossdomain.xml` policy at all (see Fig.5). The reason again is the fact that the Flash applet's actions are permitted on the basis of its own origin (`b.net`) and not on the actual origin of requests (`a.net`).

Summary: In consequence, through merely hosting a susceptible client-side Flash proxy, a Web application can undermine both its own security as well as the security of all Web applications that express trust in this application's domain via their `crossdomain.xml` policy.

3.3 Survey of published Flash proxies

In order to assess how wide spread the issues are, we conducted a survey to identify readymade client-side Flash proxies. We were able to identify five dif-

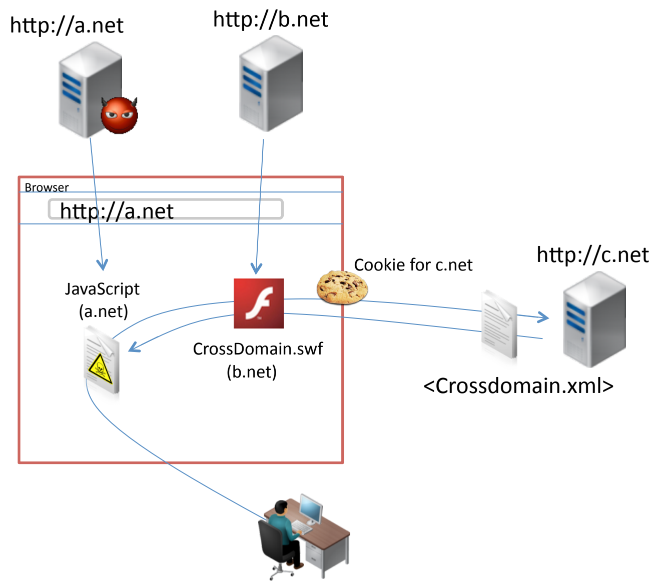


Fig. 4: Attack variant 1 - Transitivity of trust

ferent publicly available implementations: flXHR [30],SWFHttpRequest [34], FlashXMLHttpRequest [8], CrossXHR [24], and F4A[3].

Experimental setup: To check wether the found proxies expose the suspected vulnerability, we set up the network layout that is shown in Figure 3. The attacking JavaScript and its enclosing Web page were hosted on the domain `a.net`. The domain `b.net` hosted the five swf-files and on `c.net` a `crossdomain.xml` policy was set, which whitelisted only `b.net`. After writing code to connect the attacker script with the proxies for the five individual interfaces, we tested if a script executed under the origin of `a.net` is able to access text-files which are hosted on the other two domains.

Results: Out of the five examined implementation, three were vulnerable (see Table 3.3). For the two proxies which were not vulnerable the attack didn't work out, because these two libraries do not utilize the `allowDomain(*)` directive, which is needed to expose the SWF's interface to JavaScript originating from different domains. Two of the three vulnerable proxies (CrossXHR and F4A) were exploitable on the first try. The case of flXHR was more interesting, as the author of the proxy apparently was aware of the security problem and took measures to mitigate it (he even wrote a blog post about the problem [29]).

[3] The development of F4A seems to have ended. We included it in this survey, as it is still in productive use, e.g., by sites such as `nike.com`

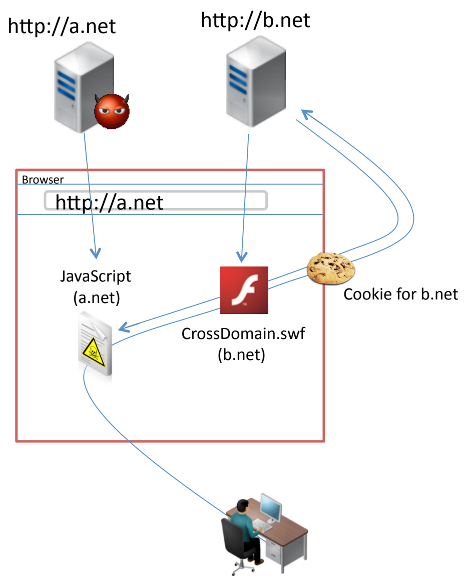


Fig. 5: Attack variant 2 - Return to sender

Case study - the flXHR proxy: flXHR is the only implementation that attempts to defend directly against the in Section 3.2 identified attack vectors. Before initiating a request, a flXHR proxy tries to compare its own origin with the origin of the interfacing JavaScript. If these two origin values do not match, flXHR checks the `crossdomain.xml` file of the request's target domain manually, if the origin of the JavaScript is whitelisted in the policy. This is even done, when the target of the request matches the origin of the Flash itself.

The problem that arises from this solution is how the origin-value of the interfacing JavaScript is obtained: As Flash does not provide such a mechanism natively, the only possibility to accomplish this task is to use the `ExternalInterface` API [3] to call a JavaScript function. In the examined case, flXHR obtains the value through calling the function `window.location.href.toString()`.

However, the JavaScript of the enclosing Web page is completely controlled by the adversary and, thus, all public JavaScript functions can be overwritten

Name	**Year**	**Source**	**JS-libs**	**Vulnerable**
flXHR [30]	2010	No	1,2,3,4	Yes, despite of countermeasures
SWFHttpRequest [34]	2007	Yes	1,2,3	No
FlashXMLHttpRequest [8]	2007	No	3	No
CrossXHR [24]	2010	Yes	1,2	Yes
F4A	unknown	No	-	Yes

Legend:
Year: Year of last recorded activity
Source: Is source code available?
JS-libs: Available plug-in for popular JavaScript frameworks: 1) JQuery, 2) Prototype, 3) Dojo, 4) Mootools

with function wrappers [22]. With the method presented in Listing 2, an attacker can fake the location of his page and make flXHR believe that the enclosing origin matches its own.

As we will discuss further in Section 5.1, with the `ExternalInterface` API it is not possible to fix this problem, because it only allows to call public JavaScript functions by name and not by reference. Therefore, it is not possible to call a function which cannot be overwritten by an adversary. In Section 5.2 we describe our approach towards ensuring that the received location value has not been tampered with.

Listing 2 Subverting flXHR's protective measures

```
self = new function(){}
self.location = new function(){};
self.location.href = new function() {
    this.toString = function(){
        return "http://b.net";
    };
};
```

3.4 Analysis

After conducting the practical vulnerability testing, we further investigated the cause of the problem via analysis of the source code (when available) or via decompilation of the binaries.

In all three vulnerable cases, the cause of the exposed insecurity is an over-allowing internal policy in respect to interaction with external JavaScript: All three vulnerable applets used the directive `allowDomain("*")` within their code (see Sec. 3.1). Even the author of the flXHR proxy decided to do so, although he was aware of the resulting potential security problems.

The reason for this design decision is inherent in the purpose of the applets: They are built to function as pre-built, drop-in solutions which can be used without modifications, following the design of general purpose programming libraries.

Without an `allowdomain()` directive, the Flash would only function with JavaScript of an origin that exactly matches the origin of the Flash file. This behavior is bound to cause problems with Web application configurations that utilize more than one (sub-)domain. Already enabling the Web application to be accessed using both the plain domain (e.g., `http://example.org`) and the www-subdomain (`http://www.example.org`) potentially breaks the proxy's functionality for one of the two alternatives.

4 Methods to provide secure client-side Flash proxy functionality

In this section we explore the safe inclusion of a client-side Flash proxy in a Web application. For this, we propose two approaches: First, we discuss how to apply the general method of CSRF protection to securely include a prefabricated, potentially vulnerable Flash proxy (see Sec. 4.1). Secondly, we show how to build a custom proxy that is safe against the attack without giving up to much of the flexibility of the existing solutions (see Sec. 4.2).

4.1 Secure inclusion via CSRF protection

Some of the already existing Flash proxies (e.g. flXHR) are matured software and provide well tested support code, such as plug-ins for popular JavaScript frameworks. For this reason it might be desirable to use them regardless of the identified security issues.

To do so securely, the application has to ensure that the proxy is only used within its foreseen environment, i.e., the proxy is indeed included in a Web page that belongs to the application. This can be done by adapting the nonce-based schema of CSRF-protection [7]: Instead of serving the applet's swf-file as a static binary, the code is delivered by a server-side script. This script verifies that the requesting URL contains a secret nonce which is tied to the requesting browser's session cookie, e.g., in the form of a URL parameter (see Listing 3 for an example). Only if the received nonce matches the expected value, the SWF is delivered to the browser. As the adversary cannot obtain such a nonce for the victim's current browser context, he is not able to create the attacking page and, hence, is unable to execute the attack.

Listing 3 CSRF protected delivery of Flash code (PHP example)

```
<?php
if ($_GET["anti_csrf_nonce"] == $_SESSION["nonce"]){
   $swf = [...]   // binary data of the .swf file
   header("Content-type: application/x-shockwave-flash");
   echo $swf;
} else {
   ... // Generate 500 internal server error
}
?>
```

4.2 Flexibly restricting the Flash-to-JavaScript interface

In order to enable HTML-to-SWF communication, Flash uses the `External-Interface` API to expose functionality to JavaScript. Cross-domain scripting from JavaScript to Flash, however, is forbidden by default. In cases where cross-domain communication is needed the directive `flash.system.Security .allowDomain("http://c.net")` can be used inside an SWF to grant scripting access from c.net.

As explained in Section 3.4 this fact causes problems for general purpose, pre-built solutions which are supposed to be usable without modifications. Thus, these libraries often use allowDomain('*'), which grants cross domain scripting access from all domains to the SWF. However, using this directive has severe security implications as the SWF is now vulnerable to the attack vectors described in Section 3.2.

As it is not feasible to disable cross-domain scripting on SWF files for general purpose libraries, we see two possibilities on how to avoid the use of `allowDomain("*")`: Hardcode the set of trusted domains or dynamically parametrize `allowDomain`-based on a separate configuration file.

The first solution requires the user to hardcode all domains he would like to grant cross-domain access to in the source code of the swf-file. This solution has two major shortcomings. For one, the user needs to have the ability to compile the source code into a working swf-file. Furthermore, he needs to do this each time he wants to update his access grants.

Therefore, we propose an alternative solution which uses a separate configuration file and dynamically grants access to all domains specified in this file. With the help of this file a user can change his configuration at any time without the need of recompiling the SWF. As this solution provides the needed flexibility and does only grant cross-domain access to trusted domains and not to the general public it is well suited for general purpose libraries.

Implementation: To examine the feasibility of this approach, we successfully created a slightly modified version of the open source crossXHR [24] proxy. On instantiation, the modified proxy now requests an `alloweddomains.xml` file from the proxy's original host (this location information can be obtained without requiring interaction with the potentially untrustworthy interfacing JavaScript). The set of domain-values contained in the received file are passed to the `allowDomain` call.

5 Securely offering public Flash interfaces

The methods described in Section 4 are sufficient to securely provide a Web application with a client-side Flash proxy. However, the underlying problem remains unsolved: How can we publicly provide a general purpose Flash applet that reliably enforces security restrictions based on the origin of the JavaScript which the applet interacts with?

5.1 Problem: An untrustworthy interface

Flash's provided option to interact with it's surrounding container is via the `ExternalInterface.call()` method [3]. This method takes the name of one public function and an arbitrary number of arguments as parameters. If the surrounding container is an HTML page, this method invokes a JavaScript function and returns the value provided by the function.

As all public JavaScript functions can be overwritten [22], an adversary is able to manipulate any value `call()` could receive, even if the called method is a native JavaScript method such as `eval()` or `window.location.href.toString()` (see Sec. 3.3). Thus, data which is received by ExternalInterface.call cannot be trusted. As discussed in [22], in most browsers a wrapped JavaScript function can be restored to its original state by calling a `delete` on the function. Hence, in theory it should be possible to circumvent all actions an adversary has taken by calling `delete` before each usage of a native function. But unfortunately, as `delete` is a JavaScript keyword and not a function, it cannot be used as a parameter for `ExternalInterface.call()`, rendering this potential solution infeasible.

5.2 Solution: Redirection to fragment identifier

As discussed above, a Flash applet's capabilities to interact with the enclosing HTML/JavaScript environment are severely limited and for most parts not outfitted to counter JavaScript wrappers which have been set up by the adversary. Hence, data retrieved from JavaScript cannot be trusted. Nevertheless, Flash needs to rely on JavaScript to obtain the URL of the enclosing page, in order to make certain security sensitive decisions.

The idea of the proposed countermeasure is to utilize redirects to fragment identifiers: If a browser window redirects itself to a URL containing a fragment identifier (also know as a local anchor), the browser automatically scrolls to the corresponding location in the displayed document after the page-loading process has terminated. However, if the URL of the currently displayed Web page already equals the target of the redirect, the page-loading step is skipped and the local anchor is accessed directly. This behavior can be utilized for validating the correctness of an URL which was retrieved by Flash's external interface mechanism:

After receiving the supposed URL of the enclosing page, the flash applet appends a fragment identifier to the URL and instructs the browser to redirected to the resulting location.

If the originally received URL was correct, i.e., it is indeed the URL of the enclosing Web page, this redirect does not cause the browser to do an actual reload. Instead, only the scroll position of the page is changed to the location of the local anchor, if a matching anchor exists in the page. If no matching anchor exist, the redirect causes no effects at all[4]. In any case, the Flash remains active in the page and can conduct the initiated action (e.g., initiating the cross-domain request).

[4] We tested this behavior with the following browsers: Firefox, Internet Explorer, Google Chrome, Opera, Safari

However, if the URL received from JavaScript was manipulated, it differs from the correct URL. Thus, the browser will redirect the user to the page the attacker's JavaScript claims to have as an origin and the Flash applet stops executing. E.g., in the attack scenario described in Section 3.3, the redirect would cause the browser window to navigate away from `a.net` towards `b.net`.

One remaining weakness of this scheme is, that the attacker could stop the browser from redirecting, using JavaScript event handlers, such as `window.onbeforeunload`. Such an action is not directly noticeable by the Flash applet and, hence, is indistinguishable from the browser's behavior in the correct case.

In order to prevent such attempts, we include a random nonce (large and unguessable) into the fragment identifier (see Listing 4). Before executing the security sensitive action the Flash applet requests the enclosing URL from JavaScript for a second time and compares the received URL to the URL saved by the Flash applet (includes the nonce) before. If the redirect was prevented, the attacker is not able to identify the random number and, thus, Flash can detect that the redirect was stopped and reject the call. If the redirect was successful and the URL was not tampered with, Flash is able to verify that the two nonces and the URLs are equal and, thus, can conclude that the first URL received from JavaScript was indeed correct (see Listing 5).

Listing 4 Redirect to fragment (sketch)

```
var location: String =
        ExternalInterface.call("self.location.href.toString");
var rand = String(Math.floor(Math.random()*RANDOM_MAX_SIZE));
this.newLocation = location + "#nonce" + rand";
var request: URLRequest = new URLRequest(this.newLocation);
navigateToURL(request,"_self");
```

Listing 5 Verifying the fragment identifier after the redirect (sketch)

```
var location: String =
    ExternalInterface.call("self.location.href.toString");

if(this.newLocation = location){
    newLocation = null;
    objects[id].send(content);
}else{
    newLocation = null; //invalidate to prevent brute forcing
    [...]               // die
}
```

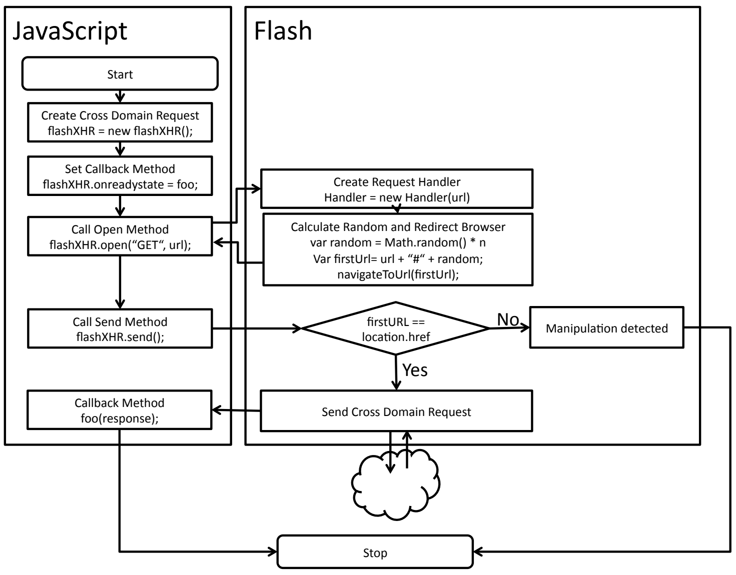

JavaScript
Flash
Start
Create Cross Domain Request
flashXHR = new flashXHR();
Set Callback Method
flashXHR.onreadystate = foo;
Call Open Method
flashXHR.open("GET", url);
Create Request Handler
Handler = new Handler(url)
Calculate Random and Redirect Browser
var random = Math.random() * n
Var firstUrl= url + "#" + random;
navigateToUrl(firstUrl);
Call Send Method
flashXHR.send();
firstURL ==
location.href
No
Manipulation detected
Yes
Callback Method
foo(response);
Send Cross Domain Request
Stop

Fig. 6: Interaction pattern between untrusted JavaScript and a secured Flash applet

Implementation: To verify our hypothesis, we implemented the outlined protection mechanism, again using the open source proxy crossXHR [24] as the basis of our implementation. Please refer to Figure 6 for further details on the modified interaction pattern between the Flash applet and surrounding JavaScript. Our implementation successfully defied both the general attacks (see Sec. 3.2) as well as the function wrapper trick (see Sec. 3.3 and Listing 2) while continuing to provide its designated functionality in legitimate use cases.

Limitation: The proposed countermeasure might cause problems with Web applications that utilize fragment identifiers to reflect their current application-state in the URL. Web applications that rely heavily on AJAX-driven interaction with their server-side, traditionally expose problems when it comes to creating hyperlinks which directly lead to specific parts of the application. For this reason, such applications occasionally append the identifier of the currently displayed application unit in the form of a fragment identifier to the URL (e.g. `http://twitter.com/#!/datenkeller`). If such applications do not foresee that the proposed countermeasure extends the window's URL (including the current fragment identifier) with the random nonce, erroneous behavior might result. However, this problem is easily avoided by the application. The appended nonce is clearly marked. Thus, removing it before processing the application's own fragment identifier is straight forward.

6 Related work

The recent history has shown that the evolution of the browser's client-side capabilities is often accompanied by the introduction of new security problems. In this section we list documented cases that relate to the issues discussed in this paper.

Issues with Flash's cross-domain request mechanism: The potential security issues with Flash's cross-domain capabilities have received some attention from the applied security community [28, 10]. To assess the potential attack surface, Grossman examined in 2008 the policy files of the Alexa 500 and Fortune 500 websites [12]. He found that at this point in time 7% of the examined websites had a policy that granted every domain full access via the `*`-wildcard. Public documentation of real issues were given by, e.g., Rios [25] who compromised Google's mail service GMail by attaching a forged `crossdomain.xml` to an email, and by Grossman [13] who accessed private information on `youtube.com` by uploading a swf-file to a host which was whitelisted in YouTube's policy. In 2010, the tool MalaRIA (short for 'Malicious Rich Internet Application') [23] was released. The tool provides a graphical user interface to interactively conduct session hijacking attacks, as outlined in Section 2.5.

Further flaws of Flash's client-side networking: Besides cross-domain aspects, Flash's handling of client-side HTTP requests exposed further security shortcomings (which have been resolved in the meantime): For one, Flash allowed in the past to add arbitrary HTTP headers to outgoing requests, leading to issues such as referrer spoofing or cross-site scripting [20]. Furthermore, it was shown that Flash's handling of client-side networking was susceptible to DNS rebinding attacks [19, 17].

Security problems with modern browser features: The introduction of client-side cross-domain requests is not the only modern browser feature that was the cause of security problems. Huang et al. [16] recently discovered that the current design of the Web socket protocol [15] in the presence of transparent Web proxies can be abused to conduct DNS cache poisoning attacks. Furthermore, Barth et al. [6] exposed a shortcoming in the `postMessage`-API [9] for asynchronous communication between frames which allowed, under certain circumstances, to compromise the confidentiality of the inter-frame message exchange. Finally, Heyes et al. [14] demonstrated how advanced features of cascading style sheets (CSS) can be utilized to create various information leaks.

7 Conclusion

As we have shown in this paper, the current movement to drop reliance on browser plug-ins in favor of applications that take full advantage of modern browser features might expose unexpected security pitfalls. There is a high probability that for practical reasons, in this transition phase, Web applications might utilize a hybrid model for client-side code, which is mainly based on native JavaScript code and uses Flash (or other plug-in technologies) solely for fallback solutions on legacy browsers. However, the subtle difference in the respective implementations of the same-origin policy together with the insufficient security architecture of the interfaces between the two technologies lead to complex security issues.

In this paper, we have demonstrated how the class of client-side Flash proxies for cross-domain HTTP requests is affected by this general problem. Furthermore, we proposed two separate countermeasures that allow the specific case of Flash HTTP proxies to be handled. These protective measures are not limited to the explored application scenario but can be applied to securely handle the general case – whenever Flash functionality is exposed to potentially untrustworthy JavaScript. Thus, the discussed techniques can be used to provide generic drop-in components for legacy browsers to securely support the transitional hybrid model.

8 Acknowledgments

We would like to thank Kyle Simpson for feedback and valuable insight in the development of flXHR.

References

1. Adobe Coperation. Adobe flash. [online] `http://www.adobe.com/products/flash/flashpro/`.
2. Adobe Systems Inc. Cross-domain policy file specification. `http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html`, January 2010.
3. Adobe Systems Incorporated. flash.external ExternalInterface . ActionScript 3.0 Reference for the Adobe Flash Platform, `http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/external/ExternalInterface.html`, accessed in January 2011, December 2010.
4. Adobe Systems Incorporated. flash.system Security. ActionScript 3.0 Reference for the Adobe Flash Platform, `http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/system/Security.html`, accessed in January 2011, December 2010.
5. Wade Alcorn et al. Browser Exploitation Framework (BeEF. [software], `http://code.google.com/p/beef/`, accessed in January 2011, 2011.
6. Adam Barth, Collin Jackson, and John C. Mitchel. Securing Frame Communication in Browsers. In *USENIX Security*, page 1730, 2008.
7. Jesse Burns. Cross Site Request Forgery - An introduction to a common web application weakness. Whitepaper, `https://www.isecpartners.com/documents/XSRF_Paper.pdf`, 2005.
8. Julien Couvreur. FlashXMLHttpRequest: cross-domain requests. [software], `http://blog.monstuff.com/archives/000294.html`, accessed in January 2011, 2007.
9. Ian IanHickson (ed.). HTML - Living Standard. WHATWG working draft, `http://www.whatwg.org/specs/web-apps/current-work/`, 2010.

10. Stefan Esser. Poking new holes with Flash Crossdomain Policy Files. [online], `http://www.hardened-php.net/library/poking_new_holes_with_flash_crossdomain_policy_files.html`, Accessed in January 2011, October 2006.
11. Google inc. Google Gadgets API: Working with Remote Content. [online], `http://code.google.com/apis/gadgets/docs/remote-content.html`, accessed in January 2011.
12. Jeremiah Grossman. Crossdomain.xml Invites Cross-site Mayhem. [online], `http://jeremiahgrossman.blogspot.com/2008/05/crossdomainxml-invites-cross-site.html`, Accessed in January 2011, May 2008.
13. Jeremiah Grossman. I used to know what you watched, on YouTube. [online], `http://jeremiahgrossman.blogspot.com/2008/09/i-used-to-know-what-you-watched-on.html`, Accessed in January 2011, September 2008.
14. Gareth Heyes, Eduardo Vela Nava, and David Lindsay. CSS: The Sexy Assassin. Talk at the Microsoft Blue Hat conference, `http://technet.microsoft.com/en-us/security/cc748656`, October 2008.
15. Ian Hickson. The Web Sockets API. W3C Working Draft WD-websockets-20091222, `http://www.w3.org/TR/2009/WD-websockets-20091222/`, December 2009.
16. Lin-Shung Huang, Eric Y. Chen, Adam Barth, Eric Rescorla, and Collin Jackson. Transparent Proxies: Threat or Menace? Whitepaper, `http://www.adambarth.com/experimental/websocket.pdf`, 2010.
17. Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting Browsers from DNS Rebinding Attack. In *Proceedings of the 14th ACM Conference on Computer and Communication Security (CCS '07)*, October 2007.
18. Samy Kamkar. Technical explanation of the MySpace worm. [online], `http://namb.la/popular/tech.html`, accessed in January 2011, October 2005.
19. Kanatoko. Anti-DNS Pinning + Socket in Flash. [online], `http://www.jumperz.net/index.php?i=2&a=3&b=3`, (19/01/07), January 2007.
20. Amit Klein. Forging HTTP Request Headers with Flash ActionScript. Whitepaper, `http://www.securiteam.com/securityreviews/5KP0M1FJ5E.html`, July 2006.
21. Benjamin Livshits and Weidong Cui. Spectator: Detection and Containment of JavaScript Worms. In *Usenix Annual Technical Conference*, June 2008.
22. Jonas Magazinius, Phu H. Phung, and David Sands. Safe wrappers and sane policies for self protecting JavaScript. In Tuomas Aura, editor, *The 15th Nordic Conference in Secure IT Systems*, LNCS. Springer Verlag, October 2010. (Selected papers from AppSec 2010).
23. Erlend Oftedal. Malicious rich internet application (malaria). [software], `http://erlend.oftedal.no/blog/?blogid=107`, accessed in January 2011, April 2010.
24. Boris Reitman. CrossXHR - a Cross-Domain XmlHttpRequest drop-in-replacement. [software], `http://code.google.com/p/crossxhr/wiki/CrossXhr`, accessed in January 2011, Feburary 2010.
25. Billy Rios. Cross Domain Hole Caused By Google Docs. [online], `http://xs-sniper.com/blog/Google-Docs-Cross-Domain-Hole/`, Accessed in January 2011, 2007.
26. Jesse Ruderman. The Same Origin Policy. [online], `http://www.mozilla.org/projects/security/components/same-origin.html` (01/10/06), August 2001.
27. Chriss Shiflett. Cross-Domain Ajax Insecurity. [online], `http://shiflett.org/blog/2006/aug/cross-domain-ajax-insecurity`, Accessed in January 2011, August 2006.

28. Chriss Shiflett. The Dangers of Cross-Domain Ajax with Flash. [online], `http://shiflett.org/blog/2006/sep/the-dangers-of-cross-domain-ajax-with-flash`, Accessed in January 2011, September 2006.
29. Kyle Simpson. (new) Adobe Flash Player security hole found, flXHRs response. [online], `http://www.flensed.com/fresh/2008/08/adobe-flash-player-security-hole/`, accessed in January 2011, August 2008.
30. Kyle Simpson. flXHR - Cross-Domain Ajax with Flash. [software], `http://flxhr.flensed.com/`, accessed in January 2011, 2010.
31. Anne van Kesteren. The XMLHttpRequest Object. W3C Working Draft, `http://www.w3.org/TR/XMLHttpRequest`, April 2008.
32. Anne van Kesteren (Editor). Cross-Origin Resource Sharing. W3C Working Draft, Version WD-cors-20100727, `http://www.w3.org/TR/cors/`, July 2010.
33. Web Hypertext Application Technology Working Groug (WHATWG). Welcome to the WHATWG community. [online], `http://www.whatwg.org/`, accessed in January 2011, 2011.
34. Jim R. Wilson. SWFHttpRequest Flash/Ajax Utility. [software], `http://jimbojw.com/wiki/index.php?title=SWFHttpRequest_Flash/Ajax_Utility`, accessed in January 2011, December 2007.
35. Michal Zalewski. Browser Security Handbook. Whitepaper, Google Inc., `http://code.google.com/p/browsersec/wiki/Main`, (01/13/09), 2008.