DEMACRO: Defense against Malicious Cross-domain Requests

Sebastian Lekies[1], Nick Nikiforakis[2], Walter Tighzert[1], Frank Piessens[2], and Martin Johns[1]

[1] SAP Research, Germany
firstname.lastname@sap.com
[2] IBBT-DistriNet, KU Leuven, 3001 Leuven, Belgium
firstname.lastname@cs.kuleuven.be

Abstract. In the constant evolution of the Web, the simple always gives way to the more complex. Static webpages with click-through dialogues are becoming more and more obsolete and in their place, asynchronous JavaScript requests, Web mash-ups and proprietary plug-ins with the ability to conduct cross-domain requests shape the modern user experience. Three recent studies showed that a significant number of Web applications implement poor cross-domain policies allowing malicious domains to embed Flash and Silverlight applets which can conduct arbitrary requests to these Web applications under the identity of the visiting user. In this paper, we confirm the findings of the aforementioned studies and we design *DEMACRO*, a client-side defense mechanism which detects potentially malicious cross-domain requests and de-authenticates them by removing existing session credentials. Our system requires no training or user interaction and imposes minimal performance overhead on the user's browser.

1 Introduction

Since the release of the World Wide Web by CERN, the online world has dramatically changed. In this ever-expanding and ever-changing Web, old technologies give way to new ones with more features enabling developers to constantly enrich their Web applications and provide more content to users. This evolution of the Web is one of the main reasons that the Internet, once accessible by an elite few, is now populated by almost 2 billion users[3].

Two of the most popular platforms for providing enriched Web content are Adobe Flash and Microsoft Silverlight.Through their APIs, developers can serve data (e.g. music, video and online games) in ways that couldn't be traditionally achieved through open standards, such as HTML. The latest statistics show a 95% and 61% market penetration of Flash and Silverlight respectively, attesting towards the platforms' popularity and longevity [17].

Unfortunately, history and experience have shown that functional expansion and attack-surface expansion go hand in hand. Flash, due to its high market

[3] http://www.internetworldstats.com

penetration, is a common target for attackers. The last few years have been a showcase of "zero-day" Flash vulnerabilities where attackers used memory corruption bugs to eventually execute arbitrary code on a victim's machine [1].

Apart from direct attacks against these platforms, attackers have devised ways of using legitimate Flash and Silverlight functionality to conduct attacks against Web applications that were previously impossible. One of the features shared by these two platforms is their ability to generate client-side cross-domain requests and fetch content from many remote locations. In general, this is an opt-in feature which requires the presence of a policy configuration. However, in case that a site deploys an insecure wildcard policy, this policy allows adversaries to conduct a range of attacks, such as leakage of sensitive user information, circumvention of CSRF countermeasures and session hijacking. Already, in 2007 a practical attack against Google users surfaced, where the attacker could upload an insecure cross-domain policy file to Google Docs and use it to obtain cross-domain permissions in the rest of Google's services [18]. Even though the security implications of cross-domain configurations are considered to be well understood, three recent studies [13, 14, 9] showed that a significant percentage of websites still utilize highly insecure policies, thus, exposing their user base to potential client-side cross-domain attacks.

To mitigate this threat, we present *DEMACRO*, a client-side defense mechanism which can protect users against malicious cross-domain requests. Our system automatically identifies insecure configurations and reliably disarms potentially harmful HTTP requests through removing existing authentication information. Our system requires no training, is transparent to both the Web server and the user and operates solely on the client-side without any reliance to trusted third-parties.

The key contributions of this paper are as follows:

- To demonstrate the significance of the topic matter, we provide a practical confirmation of this class of Web application attacks through the analysis of two vulnerable high-profile websites.
- We introduce a novel client-side protection approach that reliably protects end-users against misconfigured cross-domain policies/applets by removing authentication information from potentially malicious situations.
- We report on an implementation of our approach in the form of a Firefox extension called *DEMACRO*. In a practical evaluation we show that *DEMACRO* reliably protects against the outlined attacks while only implying a negligible performance overhead.

The rest of this paper is structured as follows: Section 2 provides a brief overview of cross-domain requests and their specific implementations. Section 3 discusses the security implications of misconfigured cross-domain policies, followed by two novel real-world use cases in Section 4. Section 5 presents in detail the design and implementation of *DEMACRO*. Section 6 presents an evaluation of our defense mechanism, Section 7 discusses related work and we finally conclude in Section 8.

2 Technical background

In this section we will give a brief overview of client-side cross-domain requests.

2.1 The Same-Origin Policy

The Same-Origin Policy (SOP) [19] is the main client-side security policy of the Web. In essence, the SOP enforces that JavaScript running in the Web browser is only allowed access to resources that share the same origin as the script itself. In this context, the origin of a resource is defined by the characteristics of the URL (namely: protocol, domain, and port) it is associated with, hence, confining the capabilities of the script to its *own* application. The SOP governs the access both to local, i.e., within the browser, as well as remote resources, i.e., network locations. In consequence, a JavaScript script can only directly create HTTP requests to URLs that satisfy the policy's same-origin requirements. Lastly, note that the SOP is not restricted to JavaScript since other browser technologies, such as Flash and Silverlight, enforce the same policy.

2.2 Client-side Cross-Domain Requests

Despite its usefulness, SOP places limits on modern Web 2.0 functionality, e.g., in the case of Web mash-ups which dynamically aggregate content using cross-domain sources. While in some scenarios the aggregation of content can happen on the server-side, the lack of client-side credentials and potential network restrictions could result in a less-functional and less-personalized mash-up. In order to accommodate this need of fetching resources from multiple sources at the client-side, Flash introduced the necessary functionality to make controlled client-side cross-domain requests. Following Flash's example, Silverlight and newer versions of JavaScript (using CORS [25]) added similar functionality to their APIs. For the remainder of this paper we will focus on Flash's approach as it is currently the most wide spread technique [14]. Furthermore, the different techniques are very similar so that the described approach can easily be transferred to these technologies.

2.3 An Opt-in Relaxation of the SOP

As we will illustrate in Section 3, a general permission of cross-domain requests would result in a plethora of dangerous scenarios. To prevent these scenarios, Adobe designed cross-domain requests as a server-side opt-in feature. A website that desires its content to be fetched by remote Flash applets has to implement and deploy a cross-domain policy which states who is allowed to fetch content from it in a white-list fashion. This policy comes in form of an XML file (`crossdomain.xml`) which must be placed at the root folder of the server (see Listing 1 for an example). The policy language allows the website to be very explicit as to the allowed domains (e.g. `www.a.net`) as well as less explicit through the use of wildcards (e.g. `*.a.net`). Unfortunately the wildcard can be used by itself, in which case all domains are allowed to initiate cross-domain requests and fetch content from the server deploying this policy. While this can be useful in case of well-defined public content and APIs, in many cases it can be misused by attackers to perform a range of attacks against users.

Listing 1 Exemplary crossdomain.xml file

```
<cross-domain-policy>
  <site-control
    permitted-cross-domain-policies="master-only" />
  <allow-access-from domain="a.net"/>
</cross-domain-policy>
```

2.4 Client-side cross-domain requests with Flash

Figure 1 gives an overview of how Flash conducts client-side cross-domain requests in a legitimate case. (The general scenario is equivalent for Silverlight and only differs in the name and the structure of its policy file). If the domain `a.net` would like to fetch data from the domain `b.net` in the user's authentication context, it has to include an applet file that implements cross-domain capabilities. This file can either present the fetched data directly or pass it on to JavaScript served by `a.net` for further processing. As already explained earlier, `b.net` has to white-list all domains that are allowed to conduct cross-domain requests. Therefore, `b.net` hosts a cross-domain policy named `crossdomain.xml` in it's root folder. (So the url for the policy-file is `http://b.net/crossdomain.xml`). If the Flash applet now tries to conduct a requests towards `b.net`, the Flash Player downloads the cross-domain policy from `b.net` and checks whether `a.net` is white-listed or not. If so, the request is granted and available cookies are attached to the request. If `a.net` is not white-listed the request is blocked by the Flash Player in the running browser.

3 Security Implications of Client-Side Cross-Domain Requests

In this section we present two classes of attacks that can be leveraged by an adversary to steal private data or to circumvent CSRF protection mechanisms.

3.1 Vulnerable Scenario 1: Insecure Policy

For this section we consider the same general setup as presented in section 2.4. This time however, `b.net` hosts personalized, access-controlled data on its domain and at the same time allows cross-domain requests from any other domain

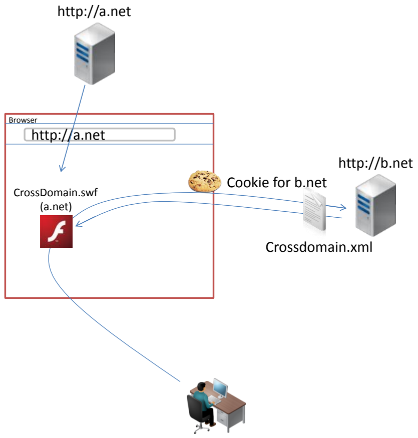

Fig. 1: General Use Case

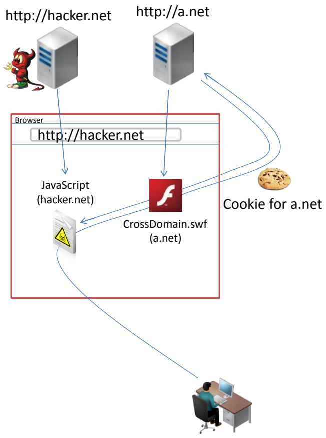

Fig. 2: Vulnerable Flash Proxy

by white-listing a wildcard ("*") in its cross-domain policy. As a result any Flash/Silverlight applet is allowed to conduct arbitrary requests towards `b.net` with the user's session cookie attached to it. Thus, an adversary is able to craft an applet file that can access the personalized, access-controlled data on `b.net`. The last step for the attacker is to lure users into visiting a website that embeds the malicious file. This could either be achieved through social networks, social engineering or through the abuse of other vulnerabilities such as cross-site scripting on vulnerable sites. The more popular the website hosting the insecure policy is, the more chances the attacker has that the users who end up visiting the malicious domain will provide him with authenticated sessions.

3.2 Vulnerable Scenario 2: Insecure Flash Proxies

As we have recently shown [10], an insecure cross-domain policy is not the only condition which enables adversaries to conduct the attacks outlined in Section 3.1: The second misuse case results from improper use of Flash or Silverlight applets. As stated in Section 2.4, an applet is able to exchange data with JavaScript for further processing. For security reasons, communication between JavaScript and Flash/Silverlight applets is also restricted to the same domain. The reason for this is that, as opposed to other embedded content such as JavaScript, embedded Flash files keep their origin. Consequently, JavaScript located on `a.net` cannot communicate with an applet served by `b.net` even if that is embedded in `a.net`. But, as cross-domain communication is also sometimes desirable in this setup, an applet file can explicitly offer communication capabilities to JavaScript served by a remote domain. Therefore, Flash utilizes a white-listing approach by offering the ActionScript directive `System.security.allowDomain(domain)`. With this directive, an applet file can explicitly allow cross-domain communication from a certain domain or white-list all domains by using a wildcard.

We have shown that these wildcards are also misused in practice: Several popular off-the-shelf cross-domain Flash proxies include such wildcard directives, and thus, allow uncontrolled cross-domain JavaScript-to-Flash communication. If such a Flash applet offers cross-domain network capabilities and at the same time grants control over these capabilities to cross-domain JavaScript, an attacker can conduct requests in the name of the website serving the applet file.

Figure 2 shows the general setup for this attack. An adversary sets-up a website `hacker.net` that includes JavaScript capable of communicating with a Flash applet served by `a.net`. This applet includes a directive that allows communication from JavaScript served by any other domain. Thus, the attacker is able to instruct the Flash applet to conduct arbitrary requests in the name of `a.net`. If JavaScript from `hacker.net` now conducts a request towards `a.net` via the vulnerable applet, the request itself is not happening cross-domain as `a.net` is the sender as well as the receiver. Therefore, the Flash Player will grant any request without even checking if there is a cross-domain policy in place at `a.net`. Consequently, the attacker can conduct cross-domain requests and read the response as if `a.net` would host a wildcard cross-domain policy. Furthermore, the adversary is also able to misuse existing trust relationships of other domains towards `a.net`. So, if other domains white-list `a.net` in their cross-domain policy, the attacker can also conduct arbitrary cross-domain requests towards those websites by tunneling them through the vulnerable proxy located on `a.net` (please refer to [10] for details concerning this class of attacks).

3.3 Resulting malicious capabilities

Based on the presented use and misuse cases we can deduce the following malicious capabilities that an attacker is able to gain.

1. *Leakage of Sensitive Information:* As an adversary is able to conduct arbitrary requests towards a vulnerable website and read the corresponding responses, he is able to leak any information that is accessible via the HTML of that site including information that is bound to the user's session id. Thus, an attacker is able to steal sensitive and private information [8].
2. *Circumvention of Cross-Site Request Forgery Protection:* In order to protect Web applications from cross-site request forgery attacks, many websites utilize a nonce-based approach [4] in which a random and unguessable nonce is included into every form of a Web page. A state changing request towards a website is only granted if a user has requested the form before and included the nonce into the state changing request. The main security assumption of such an approach is that no one else other than the user is able to access the nonce and thus, nobody else is able to conduct state changing requests. As client-side cross-domain requests allow an adversary to read the response of a request, an attacker is able to extract the secret nonce and thus bypass CSRF protections.

3. *Session Hijacking:* Given the fact that an adversary is able to initiate HTTP requests carrying the victim's authentication credentials, he is essentially able to conduct a session hijacking attack (similar to the one performed through XSS vulnerabilities). As long as the victim remains on the Web page embedding the malicious applet, the attacker can chain a series of HTTP requests to execute complex actions on any vulnerable Web application under the victim's identity. The credentials can be used by an attacker either in an automated fashion (e.g. a standard set of requests towards vulnerable targets) or interactively, by turning the victim in an unwilling proxy and browsing vulnerable Web applications under the victim's IP address and credentials (see Section 6.1).

3.4 General Risk Assessment

Since the first study on the usage of cross-domain policies conducted by Jeremiah Grossman in 2006 [7], the implementation of cross-domain policies for Flash and Silverlight applets is becoming more and more popular. While Grossman repeated his experiment in 2008 and detected cross-domain policies at 26% of the top 500 websites, the latest experiments show that the adoption of policies for the same set of websites has risen to 52% [14]. Furthermore, the amount of wildcard policies rose from 7% in 2008 up to 11% in 2011. Those figures clearly show that client-side cross-domain requests are of growing importance.

Three recent studies [9, 13, 14] investigated the security implications of cross-domain policies deployed in the wild and all came to the conclusion that cross-domain mechanisms are widely misused. Among the various experiments, one of the studies [14] investigated the Alexa top one million websites and found 82,052 Flash policies, from which 15,060 were found using wildcard policies in combination with authentication tracking and, thus, vulnerable to the range of attacks presented in Section 3.

4 Real-World Vulnerabilities

To provide a practical perspective on the topic matter, we present in this Section two previously undocumented, real-world cases that show the vulnerabilities and the corresponding malicious capabilities. These two websites are only two examples of thousands of vulnerable targets. However, the popularity and the large user base of these two websites show that even high profile sites are not always aware of the risks imposed by the insecure usage of client-side cross-domain functionality.

4.1 Deal-of-the-day Website: Insecure wildcard policy[4]

The vulnerable website features daily deals to about 70 million users world-wide. At the time of this writing, it was ranked on position 309 of the Alexa Top Sites.

[4] anonymized for publication

When we started investigating cross-domain security issues on the website, a crossdomain.xml file was[5] present, which granted any site in the WWW arbitrary cross-domain communication privileges (see Listing 2). This policy can be seen as a worst case example as it renders void all restrictions implied by the Same-Origin Policy and any CSRF protection. On the same domain under which the policy was served, personal user profiles and deal registration mechanisms were available. Hence, an attacker was able to steal any information provided via the HTML user interface. As a proof-of-concept we implemented and tested an exploit which was able to extract any personal information[6]. Furthermore, it was possible to register a user for any deal on the website as CSRF tokens included into every form of the website could be extracted by a malicious Flash or Silverlight applet.

Listing 2 The website's crossdomain.xml file

```
<cross-domain-policy>
    <site-control permitted-cross-domain-policies="all" />
    <allow-access-from domain="*" />
    <allow-http-request-headers-from domain="*" headers="*" />
</cross-domain-policy>
```

4.2 Popular sportswear manufacturer: Vulnerable Flash proxy[7]

As discussed in Section 3, even without a wildcard cross-domain policy, an attacker is able to conduct arbitrary cross-domain requests under certain circumstances. For this to be possible, a website needs to host a Flash or Silverlight file which is vulnerable to the second misuse case presented in Section 3.2.

We found such a vulnerable flash proxy on a Web site of a popular sportswear manufacturer that offers an online store for its products. Although the website's cross-domain policy only includes non-wildcard entries, it hosts a vulnerable Flash proxy which can be misused to circumvent the restrictions implied by the Same-Origin Policy.

Besides leaking private data and circumventing CSRF protections, the vulnerability can be exploited even further by an attacker to misuse existing trust relationships of the sportswear manufacturer with other websites. As the vulnerable Flash proxy enables an adversary to conduct client-side cross-domain requests in the name of the company, other websites which white-list the sportswear manufacturer's domain in their cross-domain policies are also exposed to attacks. During our tests, we found 8 other websites containing such a white-list entry.

[5] The vulnerability has been reported and fixed in the meantime.

[6] Notice: We only extracted our own personal information and, hence, did not attack any third person

[7] anonymized for publication

5 Client-Side Detection and Mitigation of Malicious Cross-Domain Requests

In Section 3.4 we showed that plug-in-based cross-domain techniques are widely used in an insecure fashion and thus users are constantly exposed to risks resulting from improper configuration of cross-domain mechanisms (see Section 3 for details). In order to safeguard end-users from these risks we propose *DEMACRO*, a client-side protection mechanism which is able to detect and mitigate malicious plug-in-based cross-domain requests.

5.1 High-level Overview

The general mechanism of our approach functions as follows: The tool observes every request that is created within the user's browser. If a request targets a cross-domain resource and is

caused by a plugin-based applet, the tool checks whether the request could potentially be insecure. This is done by examining the request's execution context to detect the two misuse cases presented in Section 3: For one, the corresponding cross-domain policy is retrieved and checked for insecure wildcards. Furthermore, the causing applet is examined, if it exposes client-side proxy functionality. If one of these conditions is met, the mechanism removes all authentication information contained in the request. This way, the tool robustly protects the user against insecurely configured cross-domain mechanisms. Furthermore, as the request itself is not blocked, there is only little risk of breaking legitimate functionality.

While our system can, in principle, be implemented in all modern browsers, we chose to implement our prototype as a Mozilla Firefox extension and thus the implementation details, wherever these are present, are specific to Firefox's APIs.

5.2 Disarming potentially malicious Cross-Domain Requests

A cross-domain request conducted by a plug-in is not necessarily malicious as there are a lot of legitimate use cases for client-side cross-domain requests. In order to avoid breaking the intended functionality but still protecting users from attacks, it is crucial to eliminate malicious requests while permitting legitimate ones. As described in Section 3.1 the most vulnerable websites are those that make use of a wildcard policy and host access-controlled, personalized data on the same domain; a practice that is strongly discouraged by Adobe [2]. Hence, we regard this practice as an anti-pattern that carelessly exposes users to high risks. Therefore, we define a potentially malicious request as one that carries access credentials in the form of session cookies or HTTP authentication headers towards a domain that serves a wildcard policy. When the extension detects such a request, it disarms it by stripping session cookies and authentication headers. As the actual request is not blocked, the extension does not break legitimate application but only avoids personalized data to appear in the response.

Furthermore, *DEMACRO* is able to detect attacks against vulnerable Flash proxies as presented in Section 3.2. If a page on `a.net` embeds an applet file served by `b.net` and conducts a same-domain request towards `b.net` user credentials are also stripped by our extension. The rationale here is that a Flash-proxy would be deployed on a website so that the website itself can use it rather than allowing any third party domain to embed it and use it.

5.3 Detailed Detection Algorithm

While *DEMACRO* is active within the browser it observes any request that occurs. Before applying actual detection and mitigation techniques, *DEMACRO* conducts pre-filtering to tell plugin- and non-plugin-based requests apart

. If a plugin-based request is observed, *DEMACRO* needs to check whether the request was caused by a Silverlight or a Flash Applet, in order to download the corresponding cross-domain policy file

. With the information in the policy file *DEMACRO* is now able to reveal the nature of a request by assessing the following values:

1. `Embedding Domain:` The domain that serves the HTML document which embeds the Flash or Silverlight file
2. `Origin Domain:` The domain that serves the Silverlight or Flash file and is thus used by the corresponding plug-in as the origin of the request
3. `Target Domain:` The domain that serves the cross-domain policy and is the target for the request
4. `Cross-domain whitelist:` The list of domains (including wildcard entries) that are allowed to send cross-domain requests to the target domain. This information is received either from the Silverlight or Flash cross-domain policy.

Depending on the scenario, the three domains (1,2,3) can either be totally distinct, identical or anything in between. By comparing these values *DEMACRO* is able to detect if a request was conducted across domain boundaries or if a vulnerable proxy situation is present. For the former, the extension additionally checks whether the policy includes a wildcard. If such a potentially malicious situation is detected the extension removes existing HTTP authentication headers or session cookies from the request. Figure 3 summarizes our detection algorithm.

In the remainder of this section, we provide technical details how *DEMACRO* handles the tasks of request interception, plugin identification, and session identifier detection.

Requests interception and redirect tracing: In order to identify plug-in-based requests, *DEMACRO* has to examine each request at several points in time. Firefox offers several different possibilities to intercept HTTP requests, but none of them alone is sufficient for our purpose. Therefore, we leveraged the capabilities of the `nsIContentPolicy` and the `nsIObserver` interfaces.

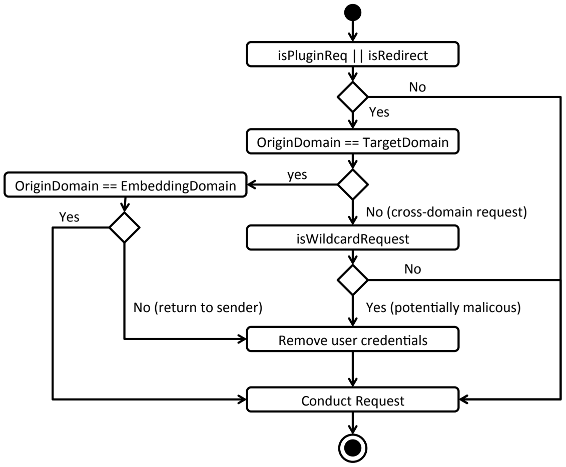
isPluginReq || isRedirect
No
Yes
OriginDomain == TargetDomain
yes
OriginDomain == EmbeddingDomain
No (cross-domain request)
Yes
isWildcardRequest
No
No (return to sender)
Yes (potentially malicous)
Remove user credentials
Conduct Request

Fig. 3: Detection and Mitigation Algorithm

The `nsIContentPolicy` interface offers a method called `shouldLoad` which is called each time an HTTP request is initiated and before the actual `HTTPChannel` object is created[8]. Thereby, the method returns a boolean value indicating whether a request should be conducted by Firefox or not. Since we do not want to block a request but only modify its header fields, this method cannot fully serve our purpose. But as it is the only method that receives the url of the webpage and the DOM object that caused the request, we need to intercept page requests here and detect the origin of a request. A request originating from either a `HTMLObjectElement` or from a `HTMLEmbedElement` is categorized as a plug-in-based request.

The `nsIObserver` interface offers the `observe` method which is called at three different points in time:

1. `http-on-modify-request`: Called each time before an HTTP request is sent.
2. `http-on-examine-response`: Called each time before the response is passed back to the caller.
3. `http-on-examine-cache-response`: Called instead of `http-on-examine-response` when the response is completely read from cache.

Thereby, the `observe` method receives an HTTPChannel object as a parameter which can be used to modify request as well as response header fields. If the extension detects a potentially malicious request, it can thus disarm it by stripping existing session information in `Cookie` fields and by removing `Authentication` header fields.

To prevent an attacker from hiding cross-domain requests behind local redirects, the extension also needs to keep track of any redirect resulting from a

[8] the HTTPChannel object is used to conduct the request and read the response

plug-in-based request. This is also done in the `observe` method at the `http-on-examine-response` event. If a 3xx status code of a plug-in-based request is detected, the redirect location will be stored for examination of follow-up requests.

During experimentation with *DEMACRO* we noticed that Web applications tend to initiate a new session if an existing session identifier is not present in a user's cookie headers. More precisely, if a session identifier never reaches the application, the application emits a `Set-Cookie` header which includes a new session identifier. If this header reaches the user's browser it will override existing cookies with the same name for the corresponding domain and therefore the user's authenticated session is replaced by an unauthenticated one. As this can obviously lead to undesired side-effects and possible denial of service attacks, *DEMACRO* additionally examines each response of potentially malicious requests and removes `Set-Cookie` headers before allowing the response to be interpreted by the browser.

Plug-in identification: In order for *DEMACRO* to investigate the correct cross-domain policy, our system must detect whether the underlying request was caused by a Silverlight or by a Flash applet. Since the HTTP request itself does not carry any information about its caller, we developed a mechanism for Firefox to distinguish between Flash and Silverlight requests.

As stated above, the only point in time where we have access to the request-causing DOM element is the call of the `shouldLoad` method in the `nsIContentPolicy` interface. But, due to the fact that Silverlight and Flash files can both be embedded into a page by using either an `HTMLObjectElement` or an `HTMLEmbedElement`, we need to examine the exact syntax used to embed those files for each element. By testing standard and less-standard ways of embedding an applet to a page, we resulted to the detection mechanism shown in Listing 3. In case the detection mechanism fails, the extension simply requests both policies, in order to prevent an attacker who is trying to circumvent our extension by using an obscure method to embed his malicious files.

Web Framework	**Name of Session variable**
PHP	phpsessid
ASP/ASP.NET	asp.net_sessionid aspsessionid
JSP	x-jspsessionid jsessionid

Table 1: Default session naming for the most common Web frameworks

Session-Cookie detection: As described earlier, it is necessary to differentiate between session information and non-session information and strip the

Listing 3 `Object` and `Embed` detection (pseudocode)

```
function detectPlugin(HTMLElement elem){

  var type = elem.getAttribute("type");
  var data = elem.getAttribute("data");
  var src  = elem.getAttribute("src");

  switch(type.startsWith){
    case "application/x-silverlight": return flash;
    case "application/x-shockwave-flash": return silverlight;
    default:
  }

  if(data=="data:application/x-silverlight")
    return silverlight;
  if(data.endsWith(".swf")) return flash;

  switch(src.endsWith){
    case ".swf": return flash;
    case ".xap": return silverlight;
    default:
  }

  return -1;
}
```

former while preserving the latter. The reasoning behind this decision is that while transmitting session identifiers over applet-originating cross-domain requests can lead to attacks against users, non-session values should be allowed to be transmitted since they can be part of a legitimate Web application's logic.

DEMACRO utilizes a techniques initially described by Nikiforakis et al. [16] and Tang et al. [23] that attempts to identify session identifiers at the client-side. The approach consists of two pillars. The first one is based on a dictionary check and the second one on measuring the information entropy of a cookie value:

The dictionary check is founded on the observation that well known Web frameworks use well-defined names for session cookies - see Table 1. By recognizing these values we are able to unambiguously classify such cookies as session identifiers. Furthermore, in order to detect custom naming of session identifiers, we characterize a value as a session cookie if it's name contains the string "sess" and if the value itself includes letters as well as numbers and is more than ten characters long. We believe that this is a reasonable assumption since all the session identifiers generated by the aforementioned frameworks fall within this categorization.

The second pillar is based on the fact that session identifiers are long random strings. Thus their entropy, i.e., the number of bits necessary to represent them, is by nature higher than non-random strings. *DEMACRO* first compares a cookie variable's name with its dictionary and if the values are not located it then calculates the entropy of the variable's value. If the result exceeds a certain threshold (acquired by observing the resulting entropy of session identifiers generated by the PHP programming framework), the value is characterized as a session identifier and is removed from the outgoing request.

This session identifier technique works without the assistance of Web servers and offers excellent detection capabilities with a false negatives rate of ~3% and a false-positive ratio of ~0.8% [16].

6 Evaluation

6.1 Security

In this section we present a security evaluation of *DEMACRO*. In order to test its effectiveness, we used it against MalaRIA [15], a malicious Flash/Silverlight exploit tool that conducts Man-In-The-Middle attacks by having a user visit a malicious website. MalaRIA tunnels attacker's requests through the victim's browser thus making cross-domain requests through the victim's IP address and with the victim's cookies. We chose Joomla, a popular Content Management System, as our victim application mainly due to Joomla's high market penetration [5]. Joomla was installed on a host with a wild-card cross-domain policy, allowing all other domains to communicate with it through cross-domain requests.

Our victim logged in to Joomla and then visited the attacker's site which launched the malicious proxy. The attacker, situated at a different browser, could now initiate arbitrary cross-domain requests to our Joomla installation. Without our countermeasure, the victim's browser added the victim's session cookie to the outgoing requests, thus authenticating the attacker as the logged-in user. We repeated the experiment with *DEMACRO* activated. This time, the plug-in detected the cross-domain requests and since the Joomla-hosting domain implemented a weak cross-domain policy, it stripped the session-identifier before forwarding the requests. This means that while the attacker could still browse the Joomla website through the victim's browser, he was no longer identified as the logged-in user.

Apart from the security evaluation with existing and publicly available exploits, we also implemented several test-cases of our own. We implemented Flash and Silverlight applets to test our system against all possible ways of conducting cross-domain requests across the two platforms and we also implemented several vulnerable Flash and Silverlight applets to test for the second misuse case (Section 3.2). In all cases, *DEMACRO* detected the malicious cross-domain requests and removed the authentication information. Lastly we tested our system against the exploits we developed for the real-world use cases (See Section 4) and were able to successfully prevent the attacks in both cases.

Request Type	**#Requests**
Non-Cross-Domain	77,988 (98.6%)
Safe Cross-Domain	414 (0.52%)
Unsafe Cross-Domain	
Without Cookies	387 (0.49%)
With Session Cookies	275 (0.34%)
With Non-Session Cookies	29 (0.05%)
Total	79,093 (100%)

Table 2: Nature of requests observed by DEMACRO for Alexa Top 1k websites

6.2 Compatibility

In order to test *DEMACRO*'s practical ability to stop potentially malicious cross-domain requests while preserving normal functionality, we conducted a survey of the Alexa top 1,000 websites. We used the Selenium IDE [9] to instrument Firefox to automatically visit these sites twice. The rationale behind the two runs is the following: In the first run, *DEMACRO* was deactivated and the sites and ad banners were populating cookies to our browser. In the second run, *DEMACRO* was enabled and reacting to all the insecure cross-domain requests by stripping-off their session cookies that were placed in the browser during the first run. The results of the second run are summarized in Table 2.

In total, we were able to observe 79,093 HTTP requests, of which 1,105 were conducted by plug-ins across domain boundaries. 691 of the requests were considered insecure by *DEMACRO* and thus our system deemed it necessary to remove any session cookies found in these requests. Of the 691, approximately half of them did not contain cookies thus these requests were not modified. For the rest, *DEMACRO* identified at least one session-like value in 275 requests which it removed before allowing the requests to proceed.

In order to find out more about the nature of the insecure requests that *DEMACRO* modified, we further investigated their intended usage: The 275 requests were conducted by a total of 68 domains. We inspected the domains manually and discovered that almost half of the requests where performed by Flash advertising banners and the rest by video players, image galleries and other generic flash applications. We viewed the websites first with *DEMACRO* de-activated and then activated and we noticed that in all but one cases, the applications were loading correctly and displaying the expected content. The one case that did not work, was a Flash advertisement that was no-longer functional when session cookies were stripped away from its requests.

One can make many observations based on the aforementioned results. First of all, we observe that the vast majority of requests do not originate from plugins which experimentally verifies the commonly-held belief that most of the Web's content is served over non-plugin technologies. Another interesting observation is

[9] `http://seleniumhq.org/projects/ide/`

1,500 C.D. requests	Firefox	FF & DEMACRO	Overhead/req.
JavaScript	27.107	28.335	0.00082
Flash	184	210	0.00173

Table 3: Best and worst-case microbenchmarks (in seconds) of cross-domain requests

that 50% of the cross-domain plugin-originating requests are towards hosts that implement, purposefully or accidentally, weak cross-domain policies. Finally, we believe that the experiments show that *DEMACRO* can protect against cross-domain attacks without negatively affecting, neither the user's browsing experience nor a website's legitimate content.

6.3 Performance

Regardless of the benefits of a security solution, if the overhead that its use imposes is too large, many users will avoid deploying it. In order to evaluate the performance of our countermeasure we measured the time needed to perform a large number of cross-domain requests when a) issued by JavaScript and b) issued by a Flash applet.

JavaScript Cross-domain requests: This experiment presents the minimum overhead that our extension will add to a user's browser. It consists of an HTML page which includes JavaScript code to fetch 1,500 images from a different domain than the one the page is hosted on. Both domains as well as the browsing user are situated on the same local network to avoid unpredictable network inconsistencies. The requests originating from JavaScript, while cross-domain, are not part of the attack surface explored in this paper and are thus not inspected by our system. The experiment was repeated 5 times and the first row of Table 3 reports the time needed to fetch all 1,500 images with and without our protecting system. The overhead that our system imposes is 0.00082 seconds for each cross-domain request. While this represents the best-case scenario, since none of the requests need to be checked against weak cross-domain policies, we believe that this is very close the user's actual everyday experience where most of the content served is done so over non-plugins and without crossing domain boundaries, as shown in Section 6.2.

Flash Cross-domain requests: In this experiment we measure the worst-case scenario where all requests are cross-domain Flash-originating and thus need to be checked and processed by our system. We chose to measure "Flash-Gallery" [10], a Flash-based image gallery that constructs its albums either from images on the local disk of the webserver or using the public images of a given user on `Flickr.com`. Cross-domain accesses occur in the latter case in order for the applet to fetch the necessary information of each image's location and finally the image itself. A feature that made us choose this applet over other Flash-based image galleries is its pre-emptive loading of all available images before the user

[10] `http://www.flash-gallery.org/`

requests them. Thus, the applet will perform all cross-domain requests needed without any user interaction.

To avoid the network inconsistencies of actually fetching 500 images from Flickr, we implemented the necessary subset of Flickr's protocol to successfully provide a list of image URIs to the Flash applet, in our own Web application which we hosted on our local network. Using our own DNS server, we resolved `Flickr.com` to the host of our Web application instead of the actual Web service. This setup, allowed us to avoid unnecessary modifications on the client-side, i.e. the Flash platform, our plug-in and the Flash applet, and to accurately measure the imposed worst-case overhead of our solution. According to the current protocol of `Flickr.com`, an application first receives a large list of image identifiers. For each identifier, the applet needs to perform 3 cross-domain requests. One to receive information about the image, one to fetch the URIs of different image sizes and finally one to fetch the image itself. We configured our Web service to return 500 image identifiers which in total correspond to 1,500 cross-domain requests. Each transfered image had an average size of 128 Kilobytes. Each experiment was repeated 5 times and we report the average timings.

The second row of Table 3 reports the results of our experiment. Without any protection mechanisms, the browser fetched and rendered all images in 184 seconds. The reason that made these requests so much slower than the non-protected JavaScript requests of Section 6.3 is that this time, the images are loaded into the Flash-plugin and rendered, as part of a functioning interactive image gallery on the user's screen. With our system activated, the same task was accomplished in 210 seconds, adding a 0.00173 seconds overhead to each plugin-based cross-domain request in order to inspect its origin, the policy of the remote-server and finally perform any necessary stripping of credentials.

It is necessary to point out that this overhead represents the upper-bound of overhead that a user will witness in his every-day browsing. In normal circumstances, the majority of requests are not initiated by Flash or Silverlight and thus we believe that the actual overhead will be closer to the one reported in the previous section. Additionally, since our experiments were conducted on the local network, any delay that *DEMACRO* imposes affects the total operation time much more than requests towards remote Web servers where the round-trip time of each request will be significantly larger.

7 Related Work

One of the first studies that gave attention to insecure cross-domain policies for Flash, was conducted by Grossman in 2006 [7]. At the time, 36% of the Alexa top 100 websites had a cross-domain policy and 6% of them were using insecure wildcards. Kontaxis et al. [13] recently reported that now more than 60% of the same set of websites implement a cross-domain policy and the percentage of insecure wildcard policies has increased to 21%. While we [14] used a more conservative definition of insecure policies, we also came to the conclusion that the cross-domain traffic through Flash and Silverlight is a real problem.

To the best of our knowledge this paper presents the first countermeasure towards this increasingly popular problem. The nature of the problem, i.e. server-side misconfigurations resulting to poor security, allows for two categories of approaches. The first approach is at the server-side, where the administrator of a domain configures the cross-domain policy correctly and thus eliminates the problem all together. While this is the best solution, it a) depends on an administrator to realize the problem and implement a secure policy and b) needs to be repeated by all administrators in all the domains that use cross-domain policies. Practice has shown that adoption of server-side countermeasures can be a lengthy and often incomplete process [27]. For these reasons we decided to focus our attention on the client-side where our system will protect the user regardless of the security provisions of any given site.

Pure client-side security countermeasures against popular Web application attacks have in general received much attention due to their attractive "install once, secure all" nature. Kirda et al. [12] attempt to stop session hijacking attacks conducted through cross-site scripting (XSS) [26] at the client side using a proxy which blocks requests towards dynamically generated URIs leading to third-party domains. Nikiforakis et al. [16] and Tang et al. [23] tackle the same problem through the identification of session identifiers at the client-side and their subsequent separation from the scripts running in the browser. Vogt et al. [24] also attempt to prevent the leakage of session identifiers through the use of static analysis and dynamic data tainting, however Russo et al. [20] have shown that the identifiers can still be leaked through the use of side channels.

Moving on to Cross-Site Request Forgeries, Johns and Winter [11] propose a solution where a client-side proxy adds tokens in incoming URLs (based on their domains) that bind each URL with their originating domain. At each outgoing request, the domain of the request is checked against the originating domain and if they don't match, the requests are stripped from their credentials. De Ryck et al. [21] extend this system, by moving it into the browser where more context-information is available. Shahriar and Zulkernine [22] propose a detection technique where each cross-domain request is checked against the visibility of the code that originated it in the user's browser. According to the authors, legitimate requests will have originated from visible blocks of code (such as a visible HTML form) instead of hidden code (an invisible auto-submitting form or JavaScript code). None of the above authors consider cross-domain requests generated by Flash and Silverlight.

Client-side defense mechanisms have also been used to protect a user's online privacy. Egele et al. [6] designed a client-side proxy which allows users to make explicit decisions as to which personal information gets transmitted to third-party social network applications. Beato et al. propose a client-side access-control system for social networks, where the publishing user can select who will get access to the published information [3].

8 Conclusion

In this paper we have shown that the increasingly popular problem of insecure Flash/Silverlight cross-domain policies is not just an academic problem, but a real one. Even high profile sites carelessly expose their users to unnecessary risks by relying on misconfigured policies and plugin applets. In order to protect security aware users from malicious cross-domain requests we propose a client-side detection and prevention mechanism, *DEMACRO*. *DEMACRO* observes all requests that occur within the user's web browser and checks for potential malicious intent. In this context, we consider a request to be potentially harmful, if it targets a cross-domain resource on a Web server that deploys an insecure wildcard policy. In such a case, *DEMACRO* disarms potentially insecure cross-domain requests by stripping existing authentication credentials. Furthermore, *DEMACRO* is able to prevent the vulnerable proxy attack in which a vulnerable Flash application is misused to conduct cross-domain requests under a foreign identity. We examine the practicality of our approach, by implementing and evaluating *DEMACRO* as a Firefox extension. The results of our evaluation suggest that our system is able to protect against malicious cross-domain requests with a negligible performance overhead while preserving legitimate functionality.

Acknowledgments: This research was done with the financial support from the Prevention against Crime Programme of the European Union, the IBBT, the Research Fund KU Leuven, and the EU-funded FP7 projects NESSoS and WebSand.

References

1. Adobe. Adobe - security bulletins and advisories.
2. Adobe Systems Inc. Cross-domain policy file specification. `http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html`, January 2010.
3. F. Beato, M. Kohlweiss, and K. Wouters. Scramble! your social network data. In *Proceedings of the 11th Privacy Enhancing Technologies Symposium (PETS)*, 2011.
4. J. Burns. Cross Site Request Forgery - An introduction to a common web application weakness. Whitepaper, `https://www.isecpartners.com/documents/XSRF_Paper.pdf`, 2005.
5. Water and Stone: Open Source CMS Market Share Report, 2010.
6. M. Egele, A. Moser, C. Kruegel, and E. Kirda. Pox: Protecting users from malicious facebook applications. In *Proceedings of the 3rd IEEE International Workshop on Security in Social Networks (SESOC)*, pages 288 –294, 2011.
7. J. Grossman. crossdomain.xml statistics. `http://jeremiahgrossman.blogspot.com/2006/10/crossdomainxml-statistics.html`.
8. J. 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.

9. D. Jang, A. Venkataraman, G. M. Swaka, and H. Shacham. Analyzing the Cross-domain Policies of Flash Applications. In *Proceedings of the 5th Workshop on Web 2.0 Security and Privacy (W2SP)*, 2011.
10. M. Johns and S. Lekies. Biting the hand that serves you: A closer look at client-side flash proxies for cross-domain requests. In *Proceedings of the 8th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2011.
11. M. Johns and J. Winter. RequestRodeo: Client Side Protection against Session Riding. In *Proceedings of the OWASP Europe 2006 Conference*, 2006.
12. E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-Side Solution for Mitigating Cross Site Scripting Attacks. In *Security Track of the 21st ACM Symposium on Applied Computing (SAC)*, April 2006.
13. G. Kontaxis, D. Antoniades, I. Polakis, and E. P. Markatos. An empirical study on the security of cross-domain policies in rich internet applications. In *Proceedings of the 4th European Workshop on Systems Security (EUROSEC)*, 2011.
14. S. Lekies, M. Johns, and W. Tighzert. The state of the cross-domain nation. In *Proceedings of the 5th Workshop on Web 2.0 Security and Privacy (W2SP)*, 2011.
15. Malaria - i'm in your browser, surfin your webs. `http://erlend.oftedal.no/blog/?blogid=107`, 2010.
16. N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *Proceedings of the 3rd International Symposium on Engineering Secure Software and Systems (ESSoS)*, 2011.
17. Rich internet application (ria) market share. `http://www.statowl.com/custom_ria_market_penetration.php`.
18. B. B. Rios. Cross domain hole caused by google docs. `http://xs-sniper.com/blog/Google-Docs-Cross-Domain-Hole/`.
19. J. Ruderman. The Same Origin Policy. [online], `http://www.mozilla.org/projects/security/components/same-origin.html` (01/10/06), August 2001.
20. A. Russo, A. Sabelfeld, and A. Chudnov. Tracking Information Flow in Dynamic Tree Structures. In *14th European Symposium on Research in Computer Security (ESORICS'09)*, 2009.
21. P. D. Ryck, L. Desmet, T. Heyman, F. Piessens, and W. Joosen. CsFire: Transparent Client-Side Mitigation of Malicious Cross-Domain Requests. In *Proceedings of 2nd International Symposium on Engineering Secure Software and Systems (ESSoS '10)*, pages 18–34, 2010.
22. H. Shahriar and M. Zulkernine. Client-side detection of cross-site request forgery attacks. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 358 –367, 2010.
23. S. Tang, N. Dautenhahn, and S. T. King. Fortifying web-based applications automatically. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, 2011.
24. P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS '07)*, 2007.
25. W3C. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`.
26. The Cross-site Scripting FAQ. `http://www.cgisecurity.com/xss-faq.html`.
27. Y. Zhou and D. Evans. Why Aren't HTTP-only Cookies More Widely Deployed? In *Proceedings of 4th Web 2.0 Security and Privacy Workshop (W2SP '10)*, 2010.